White-box cryptography

Implementing cryptographic algorithms in hostile environments.

Mattias Beimers - 3672565

November 12, 2014

Abstract

When cryptographic algorithms are executed in hostile environments, a straightforward implementation is insecure as the key can be read directly from memory. White-box cryptography aims to provide secure implementations even when the attacker has full control over the execution environment. In this paper we look at the theory and practice of a white-box AES implementation and demonstrate an application in which we replace a hardware key (dongle) with a software only white-box to check a software license.

1 Introduction

Modern cryptographic algorithms, such as AES, (Triple-)DES and RSA, are created to protect a certain message, or *plain text*, by encrypting it into a so called *cipher text*. These algorithms are designed to act like a *black-box*: the attacker has access to the inputs and outputs of the algorithm, but cannot observe or influence the computation. An example of this situation is an email: The attacker does not see what happens on the computers that send or receive the message, he can only listen at a router in between.

In practice however a cryptographic module is more like a *grey-box*: an attacker can get side channel information or even influence the computation. An example of a grey-box is a smart card. An attacker can measure information on, for instance, the power consumption or the timing of the execution. He can even inject faults by targeting specialized lasers on circuits of a chip. These kind of side channel attacks can potentially be used to gain knowledge about the used key [12, 13, 14, 15]. The safety of practical applications often does not only depend on the algorithm, but also on its implementation.

This leads us to the topic of this paper: *white-box cryptography*. In the white-box attack context an attacker can observe and modify every step in the computation. The classical example of this situation is digital rights management (DRM), where the end-user itself has incentives to break the protection of his media while having access to a decompiler or a debugger.

Another example of the white-box attack context is a mobile phone. Trends in payment move rapidly from paying with a smart card to paying with a mobile phone. Not all phones have a secure element (chip), or the access to the secure element is not restricted to the banking app only. This led to our internship project at UL Transaction Security (UL TS) where we want to find out what white-box cryptography is, how secure it is and for what applications it can be used.

To study the implementation of cryptographic algorithms in the white-box attack context we will look at the implementation as proposed by Chow et al. [2] in section 2. In section 3 we present a proof of concept for UL TS in which we seek to replace a (hardware) dongle with a white-box implementation to protect licensed software. In section 4 we will look at possible attacks on this demo and the white-box implementation in general. Afterwards we will evaluate some applications of white-box cryptography in practice in section 5 and finally we conclude in section 6.

2 The Chow et al. implementation

2.1 Introduction on white-box literature

In 2002 Chow, Eisen, Johnson and van Oorschot introduced the concept of white-box cryptography in their papers that proposed an implementation of DES [3] and AES [2] (see also Muir's tutorial [1]). Their technique of implementing a cryptographic algorithm as a network of encoded lookup tables is the standard way of creating a safe white-box implementation of a block cipher. Although they have been broken (by Wyseur et al. [5] (DES) and Billet et al. [4] (AES)), new ideas to mitigate the weaknesses have risen [16, 18, 20] and are broken again [17, 19, 21], these techniques are still the state of the art in the public literature.

In the rest of this section we will show the technical details on the white-box AES version. We chose AES over DES for several reasons. First of all because of the mathematical nature of AES as opposed to the statical nature of DES, secondly because AES is more durable than (Triple-) DES and lastly because AES is almost always used in practice (the size of an AES implementation

is more than 4 times as small as a Triple DES implementation) [3, 2]. We implement AES with a 128 bits key, but we could implement AES with other key sizes in a similar way.

2.2 Constructing a table based AES

The goal of 'white-boxing' a cryptographic algorithm is to make the implementation as strong as a black-box. One way of doing this is to create a lookup table that for every possible input block gives the corresponding output block. This is not possible of course, as that table would be approximately 5×10^{30} GB large ($2^{128} \cdot 16$ bytes). However, we can represent AES as a series of smaller lookup tables. Note that the key will be hard-wired in these lookup tables. This is a good thing, because the white-box algorithm is supposed to run securely on compromised platforms and therefore the key should never be present in memory. We will show how we implement AES using only lookup tables and then we will show how to protect these tables with internal and external encodings. But first we will give a modified description of AES.

2.2.1 Moving the ShiftRows operation

To implement AES as a series of lookup tables we use the alternative AES description on the right side of table 1. This description is equivalent to the traditional AES description on the left, but has the nice property that the ShiftRows operation is at the front. To get to this alternative description we move the first AddRoundKey in to the for loop and move the ninth AddRoundKey out of the for loop. This does not influence the result of the AES computation. Likewise we swap the ShiftRows and SubBytes operations and then the ShiftRows and the AddRoundKey operations (after modifying the round key by applying ShiftRows to it) while we keep the resulting AES computation unchanged.

2.2.2 An initial idea

Lets ignore the MixColumns operation for now. To immplement AES without the MixColumns operation, we can create lookup tables $T^*_{x,y,r}$ that map one byte of the state matrix of round r, to a byte of the state matrix of round $r+1$. This is displayed in figure 1. Such a (lookup) table $T^*_{x,y,r}$ consists of the shifted AddRoundKey operation and the SubBytes operation. The ShiftRows operation is applied by choosing which byte of the state matrix goes to which table (and therefore to which byte of the resulting state matrix). This is the

$state \leftarrow plaintext$	$state \leftarrow plaintext$
AddRoundKey($state, k_0$)	**for** $r = 1$ to 9 **do**
for $r = 1$ to 9 **do**	ShiftRows($state$)
SubBytes($state$)	AddRoundKey($state$, ShiftRows(k_{r-1}))
ShiftRows($state$)	SubBytes($state$)
MixColumns($state$)	MixColumns($state$)
AddRoundKey($state, k_r$)	**end for**
end for	ShiftRows($state$)
SubBytes($state$)	AddRoundKey($state$, ShiftRows(k_9))
ShiftRows($state$)	SubBytes($state$)
AddRoundKey($state, k_{10}$)	AddRoundKey($state, k_{10}$)
$ciphertext \leftarrow state$	$ciphertext \leftarrow state$

Table 1: Two ways to describe AES

reason why we use the AES description with the ShiftRows operation in the front.

Of course, we do not know the value of the state matrix in advance, so we store the outcome of these operations for all possible values of a byte:

$$\forall b \in \{0, \ldots, 255\} : T^*_{x,y,r}(b) = \text{SubBytes}(\text{AddRoundKey}'_{x,y,r-1}(b)),$$

$$where\ \text{AddRoundKey}'_{x,y,r}(b) = b \oplus\ k_r(ShiftRows(x, y)).$$

Notice that we now can compute the same result as a normal AES implementation without the MixColumns operation, but the key bytes are never present in memory. The only things that are visible during execution are the inputs and outputs to the lookup tables.

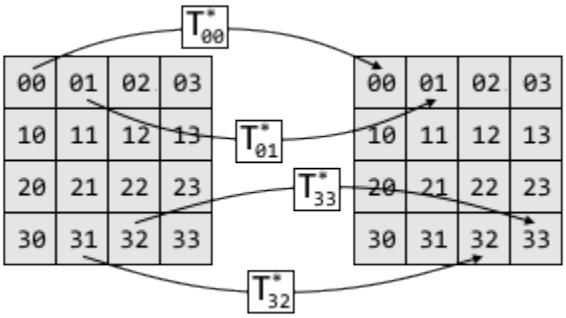

Figure 1: Tables that map one byte of the state matrix to another.

2.2.3 The actual tables

Although this works for the last round (with the addition of an extra AddRoundKey operation), this does not work for the first 9 rounds. This is because the output of the MixColumns operation does not depend on one input byte, but on all the bytes in the same column.

Let b_0, b_1, b_2, b_3 be result of applying the shifted AddRoundKey and SubBytes operations to the first, sixth, eleventh and sixteenth byte of the state matrix as input to a round r. These bytes form, after the ShiftRows operation, the first column of the state matrix before the MixColumns operation.

Applying the MixColumns operation gives the following equation:

$$\begin{pmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{pmatrix} \begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{pmatrix} = b_0 \begin{pmatrix} 02 \\ 01 \\ 01 \\ 03 \end{pmatrix} \oplus b_1 \begin{pmatrix} 03 \\ 02 \\ 01 \\ 01 \end{pmatrix} \oplus b_2 \begin{pmatrix} 01 \\ 03 \\ 02 \\ 01 \end{pmatrix} \oplus b_3 \begin{pmatrix} 01 \\ 01 \\ 03 \\ 02 \end{pmatrix}$$

This means that we can split up a matrix multiplication into 4 vector multiplications and 3 XOR operations. Each vector multiplication depends only on one byte of the input. So if we define

$$\begin{aligned} mc_0(b_0) &= b_0 \cdot (02, 01, 01, 03)^T \\ mc_1(b_1) &= b_1 \cdot (03, 02, 01, 01)^T \\ mc_2(b_2) &= b_2 \cdot (01, 03, 02, 01)^T \\ mc_3(b_3) &= b_3 \cdot (01, 01, 03, 02)^T \end{aligned}$$

then it follows that $state_{col_0} = mc_0(b_0) \oplus\ mc_1(b_1) \oplus\ mc_2(b_2) \oplus\ mc_3(b_3)$.

We now have come to the point where we can create tables $Ty^*_{x,y,r}$ that receive only one byte as input and that outputs the four bytes of the matrix multiplication with this byte. Each of these tables represents first applying the shifted AddRoundKey operation, then the SubBytes operation, and finally a multiplication with a row vector of the MixColumns matrix. Again, we store the outcome of these operations for all possible values of a byte:

$$\forall b \in \{0, \ldots, 255\} : Ty^*_{x,y,r}(b) = mc_y(\text{SubBytes}(\text{AddRoundKey}'_{x,y,r-1}(b))).$$

To calculate the resulting state bytes we need to apply a XOR operation, which we can also implement with a table: $XOR^*(b_0, b_1) = b_0 \oplus\ b_1$. To save space the XOR^* table does not map two bytes to a byte, but it maps two nibbles (four bits) to a nibble.

Right now one XOR^* table will satisfy, however, in the next section we will add encodings to protect the tables. Therefore we use six XOR^* tables for each Ty^* table.

The last round of AES does not have a MixColumns operation, so we can use the tables from the previous paragraph (but with an additional AddRoundKey operation):

$$\forall b \in \{0, \ldots, 255\} : T^*_{x,y}(b) = \text{AddRoundKey}_{x,y,10}(\text{SubBytes}(\text{AddRoundKey}'_{x,y,9}(b))).$$

As a summary, the flow of one byte through the network of tables representing all the AES rounds is depicted in figure 2.

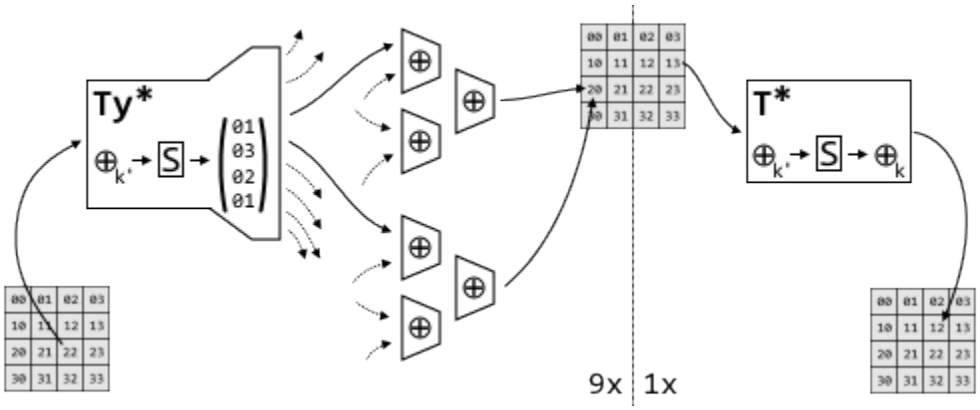

Figure 2: The flow of a byte through the table-network.

2.3 Protecting the tables

We want to protect the table network by applying encodings and Mixing Bijections. Even though the table network from the previous paragraph allows computing the AES rounds without ever having key bytes appear in memory, the key is quite easy to recover from these tables.

To extract the key from this table network an attacker could input zeros to the first round (consisting of Ty^* and XOR^* tables) and then undo the MixColumns and SubBytes by giving that result as input to a MixColumns inverse, the SubBytes inverse and the ShiftRows inverse. The result of this computation is $AddRoundKey(0, k_0)$, which outputs the AES key.

Another way an attacker could get the key is to brute force a table: since a Ty^* table only depends on a single byte of the round key, the attacker can just create up to 256 of such tables, one for every possible byte of the key, until it is equal to the Ty^* table in the code. The byte used to create this table is the byte of the key.

2.3.1 Internal encodings

To protect the tables we apply random bijections to the computation and cancel the bijection in the next table. Consider three tables A, B and C. The protected versions of these tables become $A' = f \circ A$, $B' = g \circ B \circ f^{-1}$ and $C' = C \circ g^{-1}$, where f and g are randomly chosen bijective functions.[1] Even though the tables have changed, the resulting computation of the three tables does not change, because each encoding cancels out in the next table:

$$C' \circ B' \circ A' = (C \circ g^{-1}) \circ (g \circ B \circ f^{-1}) \circ (f \circ A) = C \circ B \circ A.$$

We use this trick to encode all our tables. Because the input and output of the XOR^* tables are nibbles, and not bytes, we use two encodings per table (and two inverses from the previous table). One encoding for the first four bits, and the other encoding for the last four bits. We create such an encoding by filling an array with values from 0 to 15 (this is a 4-bit identity S-box) and then shuffling that array.

A summary of the resulting network is shown in figure 3.[2]

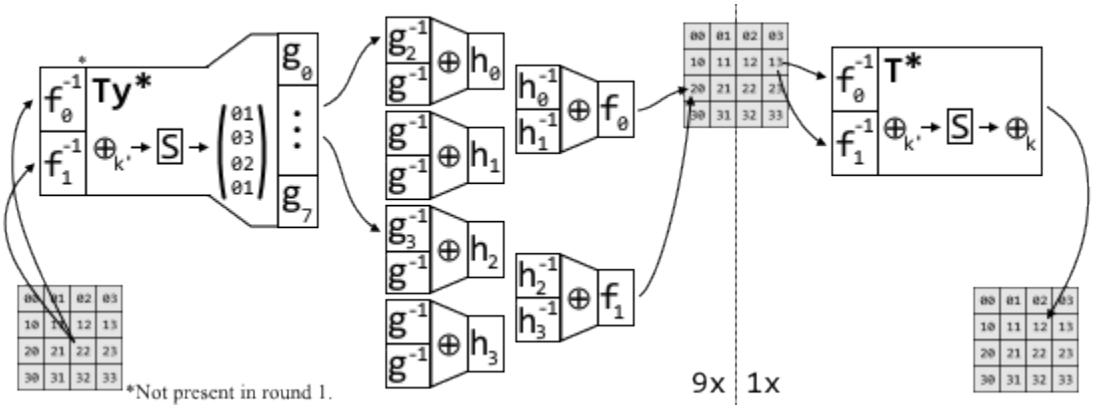

Figure 3: The flow of a byte through the table-network with encodings.

Each of these encodings can be seen as a randomly generated S-box (and with high probability a non-linear one). If we see a table (or a group of tables) as a mini cipher on their own, than these encodings are applied to the tables to achieve *confusion*: it makes sure the relation between the input and output of the tables are not related to the key bytes they are trying to hide. For a

[1] $(g \circ f)(x) = g(f(x))$.

[2] For each round we use different encodings, and the first encodings or a round (i.e. f_0^{-1} are the inverses of the last encodings from the previous round (i.e. f_0).

single table the confusion effect is perfect, it leaks no information about the key. You can see this by fixing an arbitrary table A, and check for a given key byte if you can find encodings such that the table with these properties is equal to A. Since there are such encodings for every key byte a table is perfectly secure. This is called *local security*. However, even though a single table leaks no information about the key, several tables together might leak other information (i.e. about the encodings). Therefore we also want to achieve *diffusion*: changing one bit of the input results in a change of approximately half of the bits of the output.

2.3.2 Mixing Bijections

To achieve *diffusion* for the tables we apply additional functions: invertible linear transformations, also called *mixing bijections*. We create such matrices by filling a square matrix with random bits and repeat that until it is invertible (a more sophisticated way is described by Xiao and Zhou [7]). After each of the MixColumns operations we first multiply with a 32x32 bits mixing bijection M before we apply the encodings and give it to the XOR tables. We also apply another 8x8 bits mixing bijection L^{-1} right after the encodings and before the AddRoundKey operation in the Ty tables. We use the same matrix M for the entire column (four per round) and we use a different matrix L^{-1} for each byte in the state matrix (sixteen per round). Note that the L^{-1} matrices are not the inverses of the matrix M, but they are separate mixing bijections.[3]

To undo the effect of the M multiplication after the XOR tables and to apply the correct transformations for the L^{-1} matrices, we create a new set of tables. Because the input to these tables (the output from the XOR tables) is 8 bits, and we want to multiply with a 32x32 bits matrix, we use the same trick we used for the MixColumns operation in paragraph 2.2.3, except that we split on the vector instead of the matrix and we multiply in GF(2) instead of GF(2^8):

$$M^{-1}\begin{pmatrix} b_0 \\ b_1 \\ b_2 \\ b_3 \end{pmatrix} = M^{-1}\begin{pmatrix} b_0 \\ 0 \\ 0 \\ 0 \end{pmatrix} \oplus M^{-1}\begin{pmatrix} 0 \\ b_1 \\ 0 \\ 0 \end{pmatrix} \oplus M^{-1}\begin{pmatrix} 0 \\ 0 \\ b_2 \\ 0 \end{pmatrix} \oplus M^{-1}\begin{pmatrix} 0 \\ 0 \\ 0 \\ b_3 \end{pmatrix}$$

[3]The L^{-1} matrices are called L^{-1} (*'L-inverse'*) to follow the naming of the encodings in figure 3, but this does not change anything as the inverse of an invertible linear transformation is again an invertible linear transformation.

Now if we define

$$M_0^{-1}(b_0) = M^{-1} \cdot (b_0, 0, 0, 0)^T,$$
$$M_1^{-1}(b_1) = M^{-1} \cdot (0, b_1, 0, 0)^T,$$
$$M_2^{-1}(b_2) = M^{-1} \cdot (0, 0, b_2, 0)^T,$$
$$M_3^{-1}(b_3) = M^{-1} \cdot (0, 0, 0, b_3)^T$$

and

$$L = \begin{pmatrix} L_0 & 0 & 0 & 0 \\ 0 & L_1 & 0 & 0 \\ 0 & 0 & L_2 & 0 \\ 0 & 0 & 0 & L_3 \end{pmatrix}$$

we can create our new tables $MB = g \circ L \circ M_i^{-1} \circ f^{-1}$, where f^{-1} and g are internal encodings. Finally we add the required XOR tables to finish our white-box implementation of AES.

2.3.3 External encodings

The input to the first round and the output of the last round of the implementation are not protected, therefore we add *external encodings*. These encodings change the input and output, so we are no longer computing something equivalent to AES. For external encodings F^{-1} and G we compute $G \circ AES \circ F^{-1}$. These encodings have to be undone outside the white-box. One could, for example, apply the input encodings on a server that sends the input and undo the output encodings elsewhere in the application. Although applying or undoing encodings elsewhere in the application does not add any theoretical security, it makes sure that an attacker cannot isolate the white-box, but would have to analyse the entire program.

The type of the external encodings is free to choose. In the original paper the authors suggest 128x128 bits mixing bijections, which requite additional tables to the network [2]. We could also use a series of parallel encodings like we used for the internal encodings, or a combination of the two. In our implementation for the demo in section 3 we chose to just x-or with a byte array.

2.3.4 Overview

Here we give an overview of the four final types of tables with their definitions:

1. The **Ty** table. This table contains a key addition, a substitution and a multiplication with the mix columns matrix. In the first round, the

input (8 bits) is encoded with an external encoding, in the other rounds it is encoded with internal encodings and an 8x8 bits mixing bijection. The output (32 bits) is encoded with a 32x32 bits mixing bijection and internal encodings. There are 144 Ty-tables in total.

$$Ty_{x,y,r}(b) = g(M \cdot \vec{v_y} \cdot (\text{SubBytes} \circ \text{AddRoundKey}'_{x,y,r-1} \circ L^{-1} \circ f^{-1})(b))$$

2. The **T** table. This table contains a key addition, a substitution and another key addition. The input (8 bits) is encoded with internal encodings and an 8x8 bits mixing bijection. The output (8 bits) is encoded with an external encoding. In total there are 16 T-tables.

$$T_{x,y}(b) = (g_{external} \circ \text{AddRoundKey}_{x,y,10} \circ \text{SubBytes} \circ \text{AddRoundKey}'_{x,y,9} \circ L^{-1} \circ f^{-1})(b)$$

3. The **MB** table. This table has no AES specific operations, but allows the mixing bijections operations to be undone. Therefore the input (8 bits) is encoded with internal encodings. Then it applies two 32x32 bits mixing bijections (the latter having four 8x8 bits mixing bijections on its diagonal) and finally it applies internal encodings to the output (32 bits). There are 144 MB-tables in total.

$$MB_{x,y,r}(b) = (g \circ L_x \circ M_{x,y}^{-1} \circ f^{-1})(b)$$

4. The **XOR** table. This table contains the x-or operations needed to merge the partial result of a matrix multiplication (either from the mix columns operation or from a mixing bijection). The input (8 bits) and output (4 bits) are encoded with internal encodings. In total there are 1728 XOR-tables.

$$XOR_{x,y,r,i}(a,b) = g(f^{-1}(a) \oplus \ f^{-1}(b))$$

In these formulas $\vec{v_y}$ is the y-th column vector of the mix columns matrix, M and L^{-1} are randomly generated mixing bijections (resp. 32 and 8 bits) and f^{-1} and g represent a series of parallel randomly generated 4 bits bijections.

The flow of one byte through one of the internal rounds is shown in figure 4.

2.3.5 Implementation and performance

We have implemented this in Java, which caused an issue with using lookup tables. The Java language specification requires the size of a method to be at

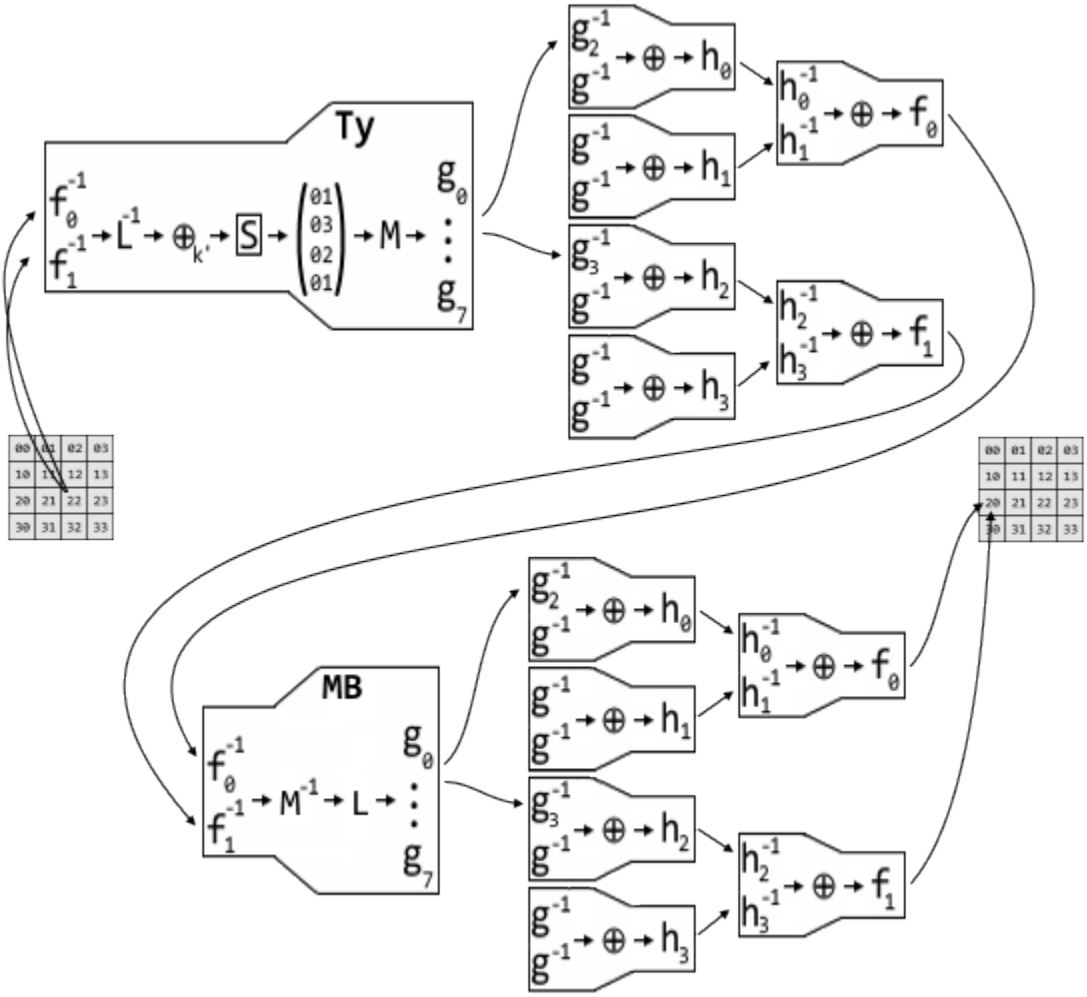

Figure 4: The flow of a byte through the tables from an inner round.

most 64 KB. Because all lookup tables are implemented as a static array and all static variables in one class count as one method, it would not compile. To solve this we created a static class for every table.

The total size of all tables is $2^8 \cdot (144 \cdot 4 + 16 \cdot 1 + 144 \cdot 4 + 1728 \cdot 1)$ bytes ≈ 741 KB. Java source code and Java byte code have a lot of overhead however. A generated file with the source code for a white-box AES is approximately 6.8 MB (41279 lines of code), compared to approximately 10 KB of source code for a straightforward implementation of AES. Compiled byte code of a white-box AES is approximately 5 MB in size, in comparison with 7 KB for a normal AES.

Generating the code of a white-box is relatively fast; it takes just under a second. Compiling this code to Java byte code however is quite slow; it takes approximately 2.5 minutes to complete on our system.

Encrypting a single block takes approximately 6 seconds. This is mainly because Java takes a long time to load all the tables in memory. Once it has the tables in memory the execution is done in a matter of milliseconds. Decrypting all the java byte code of the demo takes about 10 seconds.

Note that we ran these tests on a moderate laptop (it has a 2.50 GHz Intel i5 Core processor) and we did not do any performance optimizations. The implementation by Chow et al. [2] is about 10 times slower than a normal AES implementation. This is still a major performance hit, but it is acceptable in certain situations.

3 A 'software dongle'

As a proof of concept we created a white-box application to replace a dongle. A dongle is a piece of hardware with a cryptographic algorithm inside. Dongles are used by UL TS to license their software (by means of challenge-response calls). The advantage of using hardware over traditional software applications is that it is is infeasible to copy the dongle or to see what keys are used in the cryptographic operations (it is cheaper to just buy new software).

To protect the software against copying, a hardware fingerprint of the clients machine is sent to UL TS that will be embedded in the white-box. The application itself does not depend on the machine, only the 'software dongle'. This makes the installation convenient and adds the possibility to run the software on another computer, by requesting a new 'software dongle' as opposed to requesting an entire new software package.

This demo consists of three parts, the Generator, the 'software dongle' and the software itself. An overview of these parts is displayed in figure 5.

The *Generator* is the part that runs on a computer within UL TS and is considered to be secure. It generates the white-boxes with the keys and encodings. It also generates the challenges for the software and it encrypts the Java byte code (with $F_A \circ AES_A^{-1}$ in ECB mode). The main reason to encrypt the Java byte code is to ensure that an attacker must have access to a valid 'software dongle' in order to attack the software protection.

The *'software dongle'* contains two white-box AES implementations like we described in section 2. The external encodings are created by applying a x-or operation between the dongle id, the hardware fingerprint (only mixed in F_B) and randomly generated bytes.

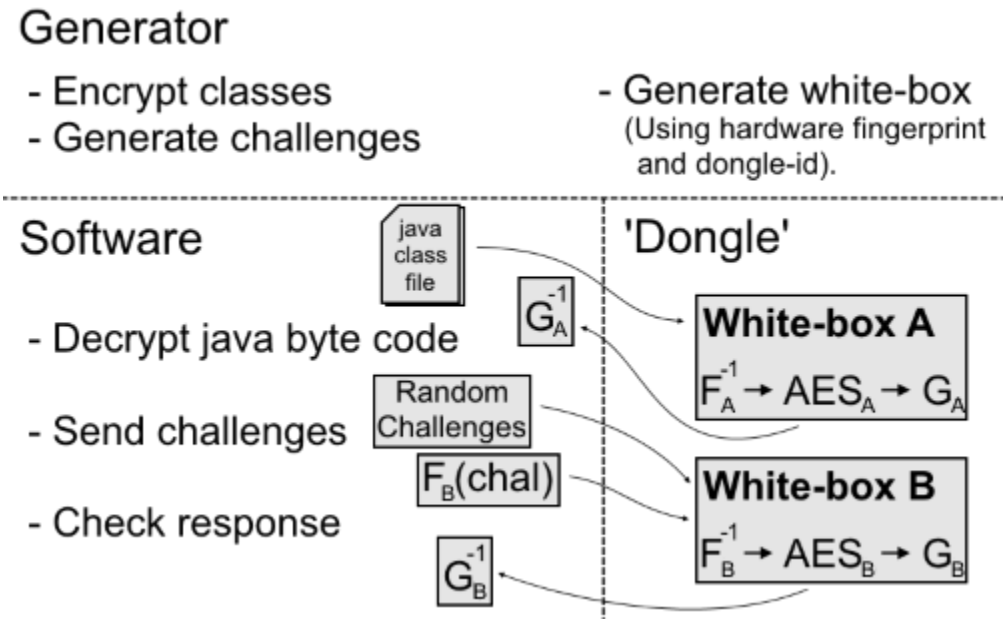

Figure 5: An overview of the demo.

The *Software* represents the actual piece of software that is bought by the customer. It contains the sophisticated logic of the software (in this case a hello world application), a class to decrypt the other Java byte code files using the dongle (White-box A) and two classes to manage random and hard-coded challenges. The software sends these hard-coded challenges to the dongle (white-box B) and verifies the response against hard-coded answers. At the same time it sends random challenges to the dongle and ignores the response. The purpose of these random challenges is to make it harder for a potential eavesdropper to find out which challenges are used. We have to use hard-coded challenges, because our software does not contain a white-box and therefore cannot compute AES_B itself.

4 Vulnerabilities

An attacker might want to install his software on multiple machines without purchasing extra licenses, or publish the software on the internet. We describe five ways in which an attacker could achieve this: social engineering, hardware spoofing, bypassing the white-box, code lifting and breaking the white-box. We attacked our application by bypassing the white-box and a lifting the code. Both were easy to do because the software was not obfuscated.

4.1 Social engineering

One example of a social engineering attack is when an attacker tells that his computer has crashed and asks for a new (free of charge) 'software dongle' for another machine. This is similar to asking for a new (hardware) dongle because the old one is lost. In this paper we will not go into details on social engineering issues, but these kind of attacks could be mitigated by blacklisting the old machine (or the old (hardware) dongle).

4.2 Hardware spoofing

An attacker could spoof his hardware so that another machine will send the same fingerprint as the original machine. While replacing or modifying the actual hardware might be difficult and expensive to do, he can use a virtual machine. Although there are ways to check if a program is running on a virtual machine, that goes beyond the scope of this demo and we have ignored these type of attacks.

4.3 Bypassing the white-box

To bypass the white-box, an attacker can catch the decrypted byte code of the classes and then modify the application to ignore the outcome of the responses, see figure 6 (left).

This attack is by far the easiest to do and very hard to prevent. It is not specific to our white-box, as an attacker can do similar attacks when a hardware dongle is used instead of a white-box implementation.

The downside of this attack is that for every software update the attacker has to extract the new decrypted Java byte code files from the updated software, modify them again and manually reinstall them at the other machines. Since it is a lot of work to do this for every update, an attacker might prefer to attack the white-box instead of the software.

4.4 Code lifting

To attack the white-box, an attacker could duplicate it, and change the encodings so that it gives the correct response for another machine. In this way the attacker can abuse the encryption functionality without knowing the key, this is known as *code lifting* (see the right side of figure 6).

To do this he needs to know the external encodings. Since some parts of the encodings are pre-applied in the generator, he cannot find these parts

anywhere in the code. However, the only thing he really needs to do is to undo the hardware fingerprint of the original machine and re-apply the hardware fingerprint of his new machine. He can find these hardware fingerprints because they are added in the software. And since we only use a x-or operation to create the external encodings, this is everything he needs. If the encodings are created such that the order matters the attacker can still do this, because he only needs to undo and redo the parts added after the hardware fingerprint which have to be in the source code as well.

If the attacker can find the hardware fingerprints, he can also find the hard-coded challenges. This means that he can easily build a new 'software dongle' that just returns the answer to these challenges and returns zeros for all other challenges (the response will be ignored). He still needs to use the first white-box to decrypt the Java byte code class files, but this white-box is machine-independent, so it can be copied without a problem. Because in practice the challenges are not updated, this is just as efficient. Note that this last attack can also be applied against the hardware dongle.

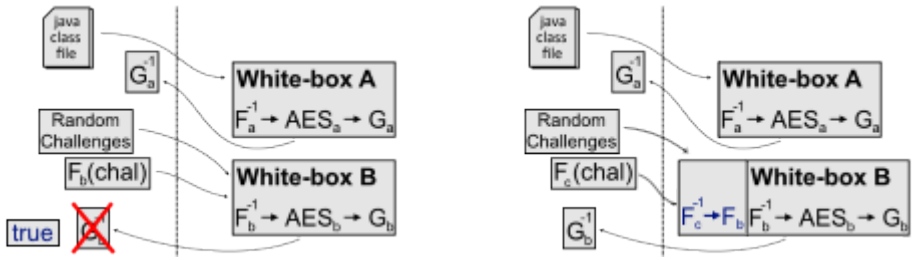

Figure 6: Attack the demo by bypassing the white-box (left) and code lifting (right).

4.5 Break the white-box

The ultimate form of attacking the white-box is to recover the key. Once an attacker has the key he can also recover the encodings and create a new 'software dongle'. In comparison with the hardware dongle this is similar to applying side channel attacks, which are either very expensive or not possible. To break just this demo it will be a lot easier to apply code lifting attacks. However, if an attacker can automate an attack against a white-box, he can potentially crack more applications than just our demo. In practice different variations are used to implement the white-box AES and a generic program to break them all will be very hard to make. Still, it is interesting to see what such an attack would look like.

4.5.1 A square like attack

The 'square like' attack was already found by Chow et al. [2]. It is inspired by an attack on Square [11], which can also attack up to 6 rounds of AES [10]. To apply it an attacker needs to remove the external encodings first. Additionally, if Mixing Bijections are not used in the creation of the white-box, then this attack can be applied at the inner rounds as well after a frequency analysis on the nibble encodings [2, 1].

If there are no input encodings to the first round, than we can apply a chosen plain-text attack, where our input will go directly to the AddRoundKey step. Because the output is encoded, we cannot see much useful about two different outputs, but we can see when two inputs have the same output.

Lets assume that we find two column values $w = (w_0, w_1, w_2, w_3)$ and $x = (x_0, x_1, x_2, x_3)$ that have a different input in all bytes, and the same output in all but one of the bytes (say, the first byte). Because the output of the last three bytes is the same, the x-or of the outputs is 0. So

$$\begin{gathered}
01 \cdot y_0 \oplus\ 02 \cdot y_1 \oplus\ 03 \cdot y_2 \oplus\ 01 \cdot y_3 = 00 \\
01 \cdot y_0 \oplus\ 01 \cdot y_1 \oplus\ 02 \cdot y_2 \oplus\ 03 \cdot y_3 = 00 \\
03 \cdot y_0 \oplus\ 01 \cdot y_1 \oplus\ 01 \cdot y_2 \oplus\ 02 \cdot y_3 = 00, \\
\text{where } y_i = S(w_i \oplus\ k_i) \oplus\ S(x_i \oplus\ k_i).
\end{gathered}$$

This is a system of linear equations, if we solve it we get:

$$\begin{aligned}
y_0 &= \mathrm{EC} \cdot y_3 \\
y_1 &= 9\mathrm{A} \cdot y_3 \\
y_2 &= \mathrm{B7} \cdot y_3
\end{aligned}$$

Now if we guess one key byte, the other three key bytes follow. As a result, we can do a brute force search on $2^{8 \cdot 4} = 2^{32}$ different keys. This of course is under the assumption that we find such w and x. According to chow et al. [2] we can find them with approximately 2^{13} one-round encryption steps.

4.5.2 The BGE attack

In 2005 Billet, Gilbert, and Ech-Chatbi published a cryptanalysis on the inner rounds of the white-box AES. The attack will recover the key and the external encodings in at most 2^{30} steps. For the details of the attack we refer to their paper [4] and Muir's tutorial [1]. The outline of the attack is:

1. Write an AES round as a system of equations. Represent one column of an AES round with the function $y_0(x_0, x_1, x_2, x_3)$.

2. Apply differential cryptanalysis; keep most variables constant and vary a few. In this case, build functions for all values of x_1: $f_b(x) = y_0(x, b, 0, 0)$, with $0 \leq b < 256$.

3. Consider all compositions: $f_b(x) \circ f_0^{-1}(x) = Q \circ \oplus_c \circ Q^{-1}$ (where Q is the output encoding of a white-boxed table and $\oplus_c$ an x-or operation with some unknown constant c).

4. Remark that these functions form a group $G = (GF(2^8), \circ)$, with some unknown base vectors (i.e. β_0 to β_7) that span up G. Consider the possibility of an isomorphism from G to $GF(2)$.

5. We can make an isomorphism that maps known base vectors (i.e. $e_0, \ldots, e_7$). So there must exist some unknown affine function that transposes between β_i and e_i.

6. Now we can create a $Q' = Q^{-1} \circ A$ (with A an unknown but affine transformation).

7. Repeat this trick for the other columns and we can replace the in- and output encodings for the next round to affine encodings (by applying Q'^{-1} to the input and next round's Q' to the output).

8. Now our system of equations has become a system of linear equations (with some constant translations), these are much easier to break.

In 2009 a generalized version of the BGE attack was published by Michiels et al. [6]. They show that this technique can be used to break any substitution linear-transformation (SLT) cipher (like AES) that uses a matrix with some specific properties (unfortunately exactly those properties that make an SLT cipher strong). The consequence of this result is that with the current techniques we cannot create a secure white-box implementation of any currently known secure ciphers. To get a secure white-box algorithm either new white-box techniques have to be found, or a new block cipher should be designed with white-box cryptography in mind.

5 White-box cryptography in practice

So far we have seen white-box cryptography being used in a DRM application and some ways to circumvent or attack this application. But in what other applications could it be used?

5.1 Security goals

Before we answer this question we look at some possible goals or properties we might want to achieve when applying white-box cryptography in practical situations:

1. *Key protection.* Key protection is the most basic property we want to have. If we can't protect the AES key we certainly won't have any of the other properties. Currently all academic white-box implementations are broken, so we don't have this property. However, we know of no proprietary white-box that is broken up to the level of key recovery and we also do not know what white-box implementations there may be in the future, so we can still look at the other properties.

2. *Non invertibility.* The property of non invertibility means that given a white-box decryption algorithm, an attacker cannot create the corresponding encryption algorithm, or the other way around. In applications such as our demo, this property is not important, while in some other applications this is essential (for example, when it is used as an asymmetric algorithm). Inverting this AES implementation is not trivial because of the encoded network of XOR tables, however, it is possible to do (it takes $40 \cdot 2^{32}$ steps). It is unknown however if white-box implementations of other cryptographic algorithms can be impossible to invert. Note that if invertibility is the only property you want, than you can create a white-box implementation of RSA. Proprietary implementations of RSA exist, though it is unknown how strong they are.

3. *Non abuse-ability.* With non abuse-ability we mean that an attacker cannot isolate the cryptographic part and use that for his own means (because why would an attacker want to recover the key if he already has the decryption functionality). In white-box cryptography we try to achieve this with the external encodings, so that an attacker needs to reverse engineer the entire application (which can be protected with obfuscation techniques) instead of just locating the white-box.

4. *Non copy-ability.* Non copy-ability means that an attacker cannot duplicate a white-box implementation. In our demo we tried to achieve this by binding it to a hardware id. However, these kind of hardware fingerprints can always be spoofed (e.g. by using a virtual machine). Because of the nature of software, this will always be an issue.

5. *Traceability.* With traceability we mean that if an attacker publishes his key (or program that uses the key) than we can trace him and know

where the leak originated. With unique keys this comes naturally (unless traitors could work together to create new keys out of their own), but even if keys are not unique, the external encodings used in the white-box can still be used to trace the origin of the leak.

5.2 A view on some applications

The classical application of white-box cryptography is *DRM*. There are many types of DRM; We have looked at licensing software, but it is also widely applied in the music and film industry, or in pay TV. In these applications it is not such a problem if an attacker can use an already compromised machine, but it is a problem is he can recover the keys and publish them to compromise other machines. Since using secure hardware elements is often unrealistic or too expensive in these situations, white-box cryptography is a good choice. Even though the white-box implementations have not yet achieved the level of security they aim to achieve, they are, when mixed with traditional obfuscation techniques, a great step forwards (Schultz [8]). We see here that traditional obfuscation and white-box cryptography go hand in hand. A white-box algorithm alone can easily be abused if the external encodings are not obfuscated throughout the rest of the program. Likewise traditional obfuscation techniques fail to hide known constants or key bytes from memory allowing them to be recovered without ever decompiling the obfuscated binary [9].

At UL TS we where curious if we can use white-box cryptography for *EMV*: secure payments. Can we stop using smart cards altogether and just use our phone? To do that we would need to use our phone to perform a cryptographic operation with a key that is hidden inside software (as opposed to a key that is hidden in a secure element). There are several risks to this, mainly because software can be duplicated. Instead of stealing the smart card, an attacker can copy the app without the owner noticing it. Likewise, the pin code is also easier to brute force, as there no longer is a fuse that blows if the pin is tried too often. To do this, an attacker only has to publish one popular app with a trojan and he can compromise millions of devices. One can of course add a requirement for a password or fingerprint recognition, but this could be attacked by intercepting system calls or registering key presses (or the touch locations on the screen if the app doesn't rely on the system keyboard). Furthermore the profit an attacker can make of breaking EMV is very high, which means that the fact that white-box cryptography is merely a great step forwards and can still be broken is a big issue. Therefore we don't recommend relying on software for EMV.

What we see happening today however is that people can use their phone

(without a secure element) to make low value payments (up to 25 euro). This is known as *HCE* (Host Card Emulation) or *cloud based payments*. The idea here is to get a set of secret tokens or keys from the internet with a limited lifetime and use these with secret keys on the phone. Because an attacker can only make low value payments, the potential profit is very small and not worth a lot of time to break it. Here white-box cryptography could be used very well to further strengthen the cryptographic operations.

6 Conclusion

We have seen a white-box implementation of AES-128 and used it to build a proof of concept in which we replaced a hardware token with a white-box. In terms of security both versions are comparable. Our white-box has the disadvantage that it is bound to a single computer and it requires a new white-box to move the software from one computer to another. On the other hand it has the advantage that it can easily be deployed on a server, for which inserting a dongle is not practical. Another serious disadvantage for our white-box is the performance. It takes about 10 seconds to decrypt the Java byte code which is way too long. Finally we have looked at other possible applications of white-box cryptography including mobile payment.

An interesting future research project would be to strengthen the cloud based payment solutions with white-box cryptography. And of course future research is needed to study the design of a new block cipher that is designed with white-box cryptography in mind, as it remains an open question whether or not theoretically secure white-box implementations will ever exist.

References

[1] Muir, J. A. (2013). *A Tutorial on White-box AES.* In Advances in Network Analysis and its Applications (pp. 209-229). Springer Berlin Heidelberg.

[2] Chow, S., Eisen, P., Johnson, H., & Van Oorschot, P. C. (2003, January). *White-box cryptography and an AES implementation.* In Selected Areas in Cryptography (pp. 250-270). Springer Berlin Heidelberg.

[3] Chow, S., Eisen, P., Johnson, H., & Van Oorschot, P. C. (2003). *A white-box DES implementation for DRM applications.* In Digital Rights Management (pp. 1-15). Springer Berlin Heidelberg.

[4] Billet, O., Gilbert, H., & Ech-Chatbi, C. (2005, January). *Cryptanalysis of a white box AES implementation.* In Selected Areas in Cryptography (pp. 227-240). Springer Berlin Heidelberg.

[5] Wyseur, B., Michiels, W., Gorissen, P., & Preneel, B. (2007, January). *Cryptanalysis of white-box DES implementations with arbitrary external encodings.* In Selected Areas in Cryptography (pp. 264-277). Springer Berlin Heidelberg.

[6] Michiels, W., Gorissen, P., & Hollmann, H. D. (2009, January). *Cryptanalysis of a generic class of white-box implementations.* In Selected Areas in Cryptography (pp. 414-428). Springer Berlin Heidelberg.

[7] Xiao, J., & Zhou, Y. (2002). *Generating large non-singular matrices over an arbitrary field with blocks of full rank.* arXiv preprint math/0207113.

[8] Schultz, R. (2012). *The many facades of drm.* MISC HS, 5, 58-64.

[9] Kerins, T., & Kursawe, K. (2006, November). *A cautionary note on weak implementations of block ciphers.* In 1st Benelux Workshop on Information and System Security (WISSec 2006) (p. 12).

[10] Daemen, J., & Rijmen, V. (1999). *AES proposal: Rijndael.*

[11] Daemen, J., Knudsen, L., & Rijmen, V. (1997, January). *The block cipher Square.* In Fast Software Encryption (pp. 149-165). Springer Berlin Heidelberg.

[12] Koeune, F., Quisquater, J. J., & Quisquater, J. J. (1999). *A timing attack against Rijndael.*

[13] Bonneau, J., & Mironov, I. (2006). *Cache-collision timing attacks against AES.* In Cryptographic Hardware and Embedded Systems-CHES 2006 (pp. 201-215). Springer Berlin Heidelberg.

[14] Mangard, S., Pramstaller, N., & Oswald, E. (2005). *Successfully attacking masked AES hardware implementations.* In Cryptographic Hardware and Embedded SystemsCHES 2005 (pp. 157-171). Springer Berlin Heidelberg.

[15] Schramm, K., Leander, G., Felke, P., & Paar, C. (2004). *A collision-attack on AES.* In Cryptographic Hardware and Embedded Systems-CHES 2004 (pp. 163-175). Springer Berlin Heidelberg.

[16] Bringer, J., Chabanne, H., & Dottax, E. (2006, January). *Perturbing and protecting a traceable block cipher.* In Communications and Multimedia Security (pp. 109-119). Springer Berlin Heidelberg.

[17] De Mulder, Y., Wyseur, B., & Preneel, B. (2010). *Cryptanalysis of a perturbated white-box AES implementation.* In Progress in Cryptology-INDOCRYPT 2010 (pp. 292-310). Springer Berlin Heidelberg.

[18] Xiao, Y., & Lai, X. (2009, December). *A secure implementation of white-box AES.* In Computer Science and its Applications, 2009. CSA'09. 2nd International Conference on (pp. 1-6). IEEE.

[19] De Mulder, Y., Roelse, P., & Preneel, B. (2013, January). *Cryptanalysis of the XiaoLai White-Box AES Implementation.* In Selected Areas in Cryptography (pp. 34-49). Springer Berlin Heidelberg.

[20] Karroumi, M. (2011). *Protecting white-box AES with dual ciphers.* In Information Security and Cryptology-ICISC 2010 (pp. 278-291). Springer Berlin Heidelberg.

[21] De Mulder, Y., Roelse, P., & Preneel, B. (2013). *Revisiting the BGE Attack on a White-Box AES Implementation.* IACR Cryptology ePrint Archive, 2013, 450.