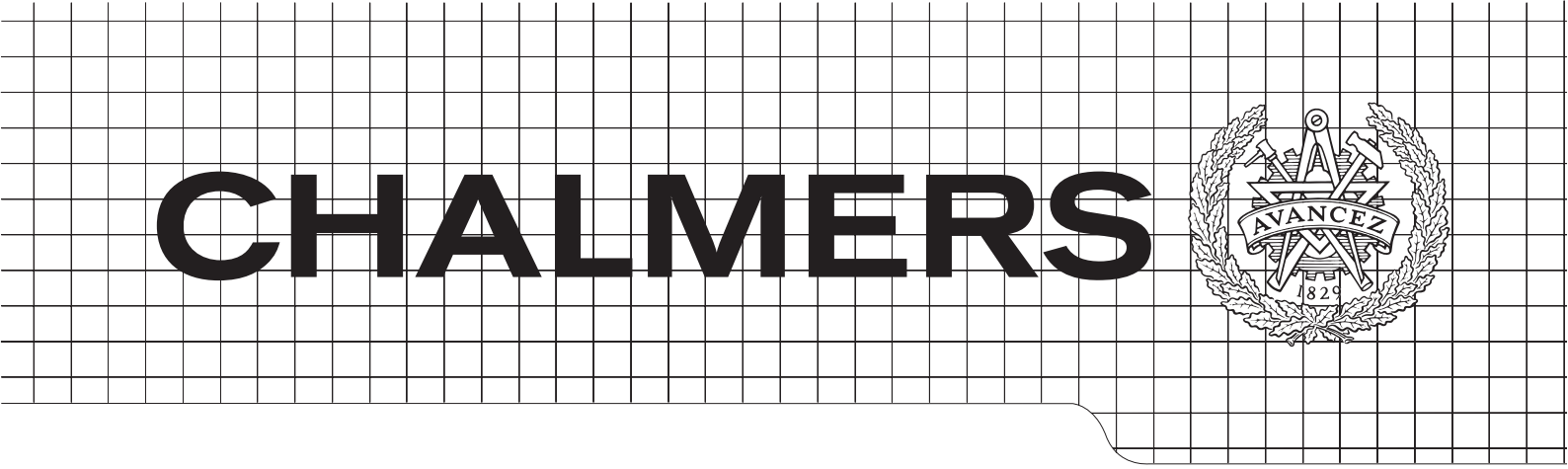


Hiding Information in Software
With Respect to a White-box Security Model

Master of Science Thesis in Computer Science

ERICA LÖFSTRÖM
ANDRÉ MALM

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
Göteborg, Sweden, June 2014

Hiding Information in Software
with Respect to a White-box Security Model

ERICA A. LÖFSTRÖM,
ANDRÉ P. MALM

Examiner: AIKATERINI MITROKOTSA

Chalmers University of Technology
University of Gothenburg
Department of Computer Science and Engineering
SE-412 96 Göteborg
Sweden
Telephone: + 46 (0)31-772 1000

Department of Computer Science and Engineering
Göteborg, Sweden June 2014

Abstract

In the digital society we rely upon our devices to both function correctly and securely. With more and more general purpose devices these properties become increasingly difficult to assure. Traditionally hardware specific devices with dedicated usage scenarios have been used to provide a safe environment for safety critical applications. With more complex devices, such as smartphones, it is however very difficult to guarantee a safe execution environment. This thesis will investigate the possibilities of hiding sensitive information in an insecure host environment. By combining several state of the art obfuscation techniques such as white-box cryptography and control flow flattening a proof of concept implementation have been created and evaluated. Although security through obscurity will offer far from perfect protection it can increase the cost of an attack. Depending on the level of security required and the types of adversaries expected it can in some scenarios offer an acceptable protection level.

Acknowledgements

We would like to thank our supervisor Aikaterini Mitrokotsa for her support and encouragement throughout the thesis. We would also like to thank John Karman, our contact person at Gemalto for his time and support. Lastly we would like to thank Petr Sturc for his time and assistance.

Erica Löfström and André Malm, Göteborg, May 20, 2015

Contents

Glossary

AES Advanced Encryption Standard (AES) is a world-wide used symmetric-key encryption algorithm, established by the U.S. National Institute of Standard and Technology (NIST) in 2001. The encryption key length can either be 128, 192 or 256 bits and the block length is 128 bits.

BGE BGE is an attack against the white-box AES implementation presented by Chow et al. The name BGE stands for the first letter of each of the authors, Billet, Gilbert and Ech-Chatbi.

Black-box Black-box describes a system model where only the produced input and output of a program is known to an observer.

DES Data Encryption Standard (DES) is a symmetric-key encryption algorithm, developed in the 1970s. DES is considered to be broken due to the short key length of 56 bits.

GF Galois Field (GF) is a mathematical term of a field containing a finite number of elements.

IDE Integrated Development Environment (IDE) is one or several programs that usually contains a text editor, a debugger, a compiler and other tools that together aims to provide an environment for programming.

JAD JAva Decompiler (JAD) is a tool written by Pavel Kouznetsov that transforms compiled java class files into java source code.

JAR JAva Archive (JAR) is a package that contains one or more java class files together with other associated data that are required in order to execute the program.

SHA Secure Hash Algorithm (SHA) is a family of cryptographic hash functions, used as a one-way function to transform some data into an unrecognizable form.

White-box White-box describes a system model where all internal states of a program is known to an observer before, after and during runtime.

XOR Exclusive Or (XOR) is a logical operation which evaluates two values and returns false if the values are equal and true if they differ.

1

Introduction

In the modern society digital devices have become a more and more integral part of every day life. One can find embedded computers in almost everything. As technology advances the demand for functionality and availability increases with it. We rely on that these devices function correctly and securely. Traditionally, purpose specific hardware has been used to run safety critical applications. While this creates a secure execution environment it also comes with disadvantages such as high hardware cost and low user flexibility. In comparison protecting the safety critical execution with software, where the end-user would already be in possess of the required hardware, for example a smartphone, would not only be both cheaper and more flexible but also more convenient. Since such hardware would need to have access to the Internet it is crucial to protect the device from adversaries.

The goal with protecting the safety critical execution is to hinder an adversary from retrieving sensitive information from the application. Both data, such as secret keys, and code, such as proprietary algorithms, may be of protection needs. How secure a system is against different attacks depends largely on the security model describing the assumptions of the system. The conventional *black-box* and *grey-box* models are no longer sufficient because of the assumption that the application is run on an uncompromised host. Therefore, an additional, *white-box* model has been introduced. A short description of the different models are given below.

In the *black-box* model the assumption is that an adversary can only access the input/output behavior of the running program. In order to be completely secure the adversary should not gain any knowledge of the functionality of the system with this information. Execution in the black-box model will henceforth be known as running in a *black-box environment*.

The *grey-box* model is similar to a black-box environment with the exception that the system is leaking some kind of input dependent information, such as the execution time, radiation or other physical characteristics. In some cases this might give enough information about the system for a successful attack to be performed.

In contrast to the previous, rather limited, models the *white-box* model implies that an adversary has complete access to the system. An adversary can read and modify any data used by the system at any time. An adversary also has the ability to analyze the program before, during and after execution. Analyzing the system without executing it is called *static analysis* while performing a run-time analysis is called *dynamic analysis*. It is important to protect the system against both types of analyses as well as making the underlying logic incomprehensible for an adversary. Execution in the white-box model will henceforth be known as running in a *white-box environment*.

1.1 Problem Description

Due to the significant advantages an adversary gains in a white-box environment the traditional protection techniques designed for a black-box environment is no longer giving a satisfactory security. An adversary can perform reverse engineering by observing and modifying the internals' of the program and therefore other techniques to secure sensitive information are required. Achieving the same security in a white-box environment as in a black-box environment by obfuscation was in 2001 proven to be impossible by Barak et al. [1] The impossibility result was based on the definition that an obfuscated program in a white-box environment behaves as a "virtual black box", i.e. an adversary is not gaining any knowledge that can lead to an exploit of the program. Even though Barak et al. [1] provided an impossibility result, obfuscation should not be discarded. By providing a more loose definition of an obfuscated program, one can achieve "good-enough" security. The main goal with obfuscation is to make the process of successfully reverse engineering the program difficult enough so that the time and resources required are not cost effective.

Collberg et al. [2] has defined a more loose definition of an obfuscated program, presented in Definition 1 below. In this paper we will use this definition when referring to obfuscation.

Definition 1. *(Obfuscating Transformation)*
Let $P \xrightarrow{\tau} P'$ be a transformation of a source program P into a target program P'.

$P \xrightarrow{\tau} P'$ is an *obfuscating transformation*, if P and P' have the same *observable behavior*. More precisely, in order for $P \xrightarrow{\tau} P'$ to be a legal obfuscating transformation the following conditions must hold:

- If P fails to terminate or terminates with an error condition, then P' may or may not terminate.

- Otherwise, P' must terminate and produce the same output as P.

□

1.2 Aim and Method

The aim of this Master Thesis is to research how and whether it is possible to obfuscate a source program P so that it is unprofitable for an adversary to extract the secret information. The program is considered to be unprofitable to reverse engineer if the cost it takes to reverse engineer is greater than the potential gain. The point is to make the reverse engineering process so tedious that the adversary give up before even finishing. In order to make the reverse engineering process unprofitable, the obfuscate program P' should withstand known automatic reverse engineering tools as well as harden the manual analysis of P'. Furthermore, each instance of P' should differ in some way so that the adversary have to spend additional resources for each instance of the program. By forcing an adversary to manually reverse engineer each instance the amount of time and resources required increases drastically compared to using an automatic tool to perform the entire reverse engineering process alone.

To gain a better understanding of the area of obfuscation and to determine which techniques to investigate further a literature review of relevant research has been conducted. While there is relatively much information available on different obfuscation techniques in an isolated perspective, we found that little research has been published of combining techniques into a single solution. First we use an obfuscation technique called *white-box cryptography* to implement a white-box Advanced Encryption Standard (AES) encryptor which aims to hide an AES key while still maintaining encryption functionality. This implementation acts as our source program P. We then investigate how well other obfuscation techniques work together with the source program P by creating an obfuscated version P'. The goal of the obfuscated program P' is to improve the security level so that it will be even more expensive for an adversary to extract the secret key in comparison to P. Both the source program P and the obfuscated program P' will be further described in Chapter 4.

Our work will be performed according to the following steps:

- Implement source program P.
- Identify the goal of an adversary.
- Define the knowledge-space and capability of an adversary.
- Attempt to reverse engineer P with the defined adversary knowledge capacity.
- Select obfuscation techniques that makes it more difficult for an adversary to reach his goals.

- Implement the obfuscated program P'.
- Evaluate the total obfuscation quality of P'.
- Attempt to reverse engineer P' with the defined adversary knowledge capacity.
- Compare the reverse engineering process of P' with the reverse engineering process of P.

1.3 Contribution

Our contribution to the area of obfuscation is to investigate if combining several obfuscation techniques hardens the reverse engineering process. Specifically, how well white-box AES obfuscation technique works together with other existing obfuscation techniques. We have done this by first implementing a white-box AES solution, referred to as P. We have then continued by developing an obfuscated version of P, referred to as P' that combines several other obfuscation techniques together with the white-box AES implementation. The result is a tailor-made solution that combines several state-of-the-art obfuscation techniques into a unique solution that hardens the defense against known white-box AES attacks. The aim is not to make the reverse engineering process impossible but instead as cost ineffective as possible, optimally to the point where an adversary would not profit from performing an attack by reverse engineering.

1.4 Ethical and Sustainability Issues

Our research aims to explore and analyze different methods for hiding information in the program code. Consequently the results of this study may be used for malicious intent, for instance obfuscating and hiding unwanted behavior in program code of viruses or malware.

Obfuscating program code almost always causes a negative impact on performance. Compared to an unobfuscated program more time and resources for executing the same job is required. This will increase the energy consumption and may for that reason have a negative impact on the environment. However the environmental impact of the increased energy consumption ought to be marginal compared to the manufacturing and distributing of hardware.

1.5 Outline

The report begins with a chapter on obfuscation techniques which contains a general introduction to different obfuscation techniques available. Furthermore each obfuscation technique is described, and a short review of the state-of-the-art research is presented. Chapter 3 contains the theory on both black-box and white-box AES. The following two

next chapters describe the implementation as well as an evaluation of that implementation. Before discussion and conclusion the result from the implementation is presented.

2

Obfuscation Techniques

Obfuscation is used to harden reverse engineering by making the program code difficult to analyze while keeping the same functionality. The objective of obfuscation is to make the logic of the program incomprehensible to both humans and automated analysis tools. There exist several different techniques, each with different level of security, some are used to hinder static analysis while others aim to prevent dynamic analysis.

2.1 Code Obfuscation

One important obfuscation technique is code obfuscation which aims to transform a program to a functionally equivalent one with the goal of making it more difficult for both humans and automatic tools to understand it. Code obfuscation can range from a primitive name scrambling transformation to a more complex transformation where the control flow of the program is obfuscated. Collberg et al. [2] have classified code obfuscation into four different types; 1) layout obfuscation, 2) control obfuscation, 3) data obfuscation and 4) preventive transformations. The following sections will go into more details about each of them.

2.1.1 Layout Obfuscation

Depending on the programming language used, different amounts of identifier information, such as variable or argument names, will remain in the final form. This information may help an adversary to gain understanding of the program as names are often descriptive. Layout obfuscation, also known as lexical transformation, is a technique that transforms, or scrambles, all identifiers to a non meaningful name. Layout obfuscation is used to make it harder for humans to understand the logic of a program. In Listing 2.1 a short method written in Java is presented. Listing 2.2 presents the same code after layout obfuscation has been applied. By just looking at the identifier names in the deobfuscated code, one can easily guess what the program is supposed to do without

actually running it. The logic of the obfuscated code on the other hand, is significantly harder to understand.

Listing 2.1: Unobfuscated code

```
private void SendNewsletter(Template template, Recipient recipient){
   if (recipient.IsSubscribed()){
      Mail mail = GenerateMail(template);
      recipient.updateSendCount();
      SendMail(mail, recipient);
   }
}
```

Listing 2.2: Obfuscated code with lexical transformation

```
private void a(b c, d e){
   if (e.a()){
      F g = h(c);
      e.a();
      i(g, e);
   }
}
```

Apart from scrambling identifier names, layout obfuscation also includes removing comments from the original code as well as removing source code formatting information in class files. Layout obfuscation is primarily applied by running an automatic tool which is specific for each program language.

2.1.2 Control Obfuscation

In order to successfully understand a program one needs to be able to tell when and in which order operations are executed. A valuable method used when analyzing a program is to construct a so called control flow graph. Control flow graphs visualize all executions paths that can be traversed in a given program, what code blocks precedes and succeeds every other code block as well as the conditions that determine which execution path to take next. The aim of control obfuscation is to increase the difficulty to trace the execution flow of a program. If the adversary has trouble understanding the correct control flow of a program, then he might also find it hard to make any meaningful manipulations to the code.

Control Flow Flattening

Control flow flattening is a technique that can be used to make control flow analysis more difficult. Control flow flattening uses a dispatcher node that decides what code blocks should be executed next. When a code block is finished executing it will direct the flow to the dispatcher node instead of passing it directly to the succeeding block. By using this technique a control flow graph of the program will show all blocks as potential predecessors or successors of each other.

Wang et al. [3] proposed a method for flattening control flow graphs using aliased pointers, though no actual algorithm was presented. In 2009 László and Kiss [4] introduced an algorithm that made use of control flow flattening, and then implemented it in C++. Listings 2.3 below shows the original source code which is about to be flattened. Listings 2.4 shows the resulting code when control-flow flattening has been applied.

Listing 2.3: Source code original

```
i = 1;
s = 0;
while (i <= 100){
  s+=i;
  i++;
}
```

Listing 2.4: Source code flattened

```
int swVar = 1;
while(swVar != 0){
  switch (swVar){
    case 1:{
      i = 1;
      s = 0;
      swVar = 2;
      break;
    }
    case 2:{
      if(i<=100)
        swVar = 3;
      else
        swVar = 0;
      break;
    }
    case 3:{
      s+=i;
      i++;
      swVar=2;
      break;
    }
  }
}
```

Cappaert et al [5] found Lászlò and Kiss implementation vulnerable since the assignment of the `swVar` variable is hard-coded. An attacker can simply perform a local scan of the code for `swVar` assignments in order to determine the successor block. They also raised a concern regarding the `if-else` statement since the code still revealed where the execution would branch.
To hinder a local analysis the `swVar` should be assigned relatively, i.e; the new value should depend on the current value. If the current value is unknown the succeeding block will also be unknown. However as the value of the `swVar` is always equal to the current case label the value is still vulnerable to a local analysis. One solution is to use a bijective, one-way function that operates on the `swVar`. While not strictly bijective, cryptographic hash functions such as Secure Hash Algorithm (SHA) can be used for this purpose as finding collisions is hard. The hash function will take the `swVar` together with the previous hash in order to produce a new hash to be used in the switch case. This will render a local attack impractical as the hash, determining the succeeding block, will be based on all preceding blocks. This will force an attacker to trace the execution

path from the very first block in order to find the correct hash. To complicate things further one can also introduce bogus blocks that either never gets executed or executes operations that doesn't affect the end result.

Opaque Predicates

Another way to harden control flow analysis is to embed opaque predicates in the code. A predicate is a branch with two different outcomes that is determined by a statement that can be either true or false. The opaque property implicate that even though the outcome is known a priori it is hard to deduce the outcome by using static program analysis. Even though opaque predicates offers good resistance against static analysis they are still vulnerable to dynamic analysis since an adversary can observe the outcome during runtime.

2.1.3 Data Obfuscation

It is not only the structure of the program that needs to be obfuscated but also the data used during computations. There exist several different kinds of data obfuscation techniques, dependent on the data being obfuscated. Collberg et al. [2] define data obfuscation as altering the storage, ordering, encoding and aggregation of the data.

Depending on the programming language used, there are some "traditional" ways to store data, such as storing the data in variables and arrays. Memory allocation for such variables and arrays usually is done in sequence and stored in the same memory area. In order to obfuscate the data non traditional storage forms would be preferable. For example, shuffling or splitting data into different memory regions and using non-standard structures are actions that can be taken in order to reduce understandability.

Another important part to obfuscate is the data access operations, for instance the index values in arrays. Shuffling or splitting data into different memory regions does not do much unless the fetching operations is obfuscated as well. Consider iterating over an array by using a for-loop as an example. When accessing the array one normally uses the iterating variable as index for the array. This can be changed so instead of using the iterating variable directly one can for instance use hashing algorithms together with algebraic operations in order to produce unpredictable indexes that are hard to statically analyze.

Apart from index values in arrays, evaluation expressions used in predicates should also be obfuscated. Traditionally, a boolean expression, which either evaluates as either true of false, is used to check a condition in a predicate. An example is the comparison of two values to see if the first value is greater than second. By splitting the variables into several separate variables, which are algebraically combined during evaluation, hardens the understanding of the expression. For example, if `i = 6` and `j = 9` then the boolean expression `i > j` will evaluate to `False`. The expression `32 * x + 7 - y / 5 * 7 -`

`z`, where `x = 5, y = 52 and z = 155`, will also evaluate to `6` and can therefore replace `i`. However, this only hardens the human understanding and an automatic tool can just evaluate the expression into `6` again. On the other hand, the hard-coded constants that are used for comparison can be hashed in order to hide their true value. This forces the adversary to perform a dynamic analysis rather than statically analyzing the expression.

Aggregating data is also a usable technique in data obfuscation. Structures can be aggregated and stored in an unconventional way. An example is storing two 32-bit values in one 64-bit variable which when examined is regarded as a 64-bit value rather than 2 smaller values. Another way to aggregate data is to reorder the whole data structure. Arrays can be split in smaller sub-arrays which forces an adversary to find all the sub-arrays in order to retrieve the data of the original array. In the same way, several arrays can be merged into one large array. Arrays can also be folded or flattened, which means increasing or decreasing the dimensions of the array respectively.

2.1.4 Preventive Transformation

The previous obfuscation techniques described foremost tries to harden the understanding of the underlying program logic for a human reader. The main aim of preventive transformation on the other hand is to harden the use of known automatic deobfuscation techniques. The goal of an automatic deobfuscation tool is to evaluate transformed code in order to attempt to restore the original program code. However, if a deobfuscation technique is known one can take specific measures in order to fool it. For instance, inserting junk bytes between instructions may counter the dissembling process. Preventive transformation can also be used as a compliment to other obfuscation techniques. Reordering a for-loop to run backwards can easily be identified by an automatic deobfuscation tool. However, by adding a bogus data dependency inside the reversed loop can hinder the tool to identify that the loop is reversed.

2.2 Code Encryption

When reverse engineering an adversary usually analyze the program both statically and dynamically. In order to protect the program from static analysis the code can be stored in an encrypted form, which means that it is unreadable unless executed. Naturally an entry point containing a decryption routine will need to be stored in an unencrypted form. The design of this decryption routine will affect the difficulty of a dynamic analysis. Also, even though the decryption routine is stored in an unencrypted form it should be obfuscated in some way to harden the reverse engineering process further. Generally, one can identify two different types of decryption techniques, bulk decryption and on-demand decryption, both presented below.

2.2.1 Bulk Decryption

With bulk decryption the entire program will simply be decrypted and loaded into memory upon start. The program will then execute the unencrypted version of the program. This method offers very little protection as an adversary could just copy the decrypted program from memory after the decryption routine is run. However, the technique still requires that the adversary needs to execute the program in order to retrieve a decrypted copy. Bulk decryption should however be used together with other obfuscation techniques for best results.

2.2.2 On-Demand Decryption

On-demand decryption works by decrypting specific code parts only when they are needed. The decrypted parts are loaded into memory, executed and then encrypted again as they have already been executed. This method increases the amount of work required to extract the decrypted code. Still, the on-demand decryption offers greater protection than bulk decryption since only executed code is decrypted. If program execution differs from time to time, the adversary needs to execute the program with all possible outcomes in order to retrieve the complete decrypted program. On-demand decryption should as well be used in complement with other obfuscation techniques for best results.

2.3 Software Diversification

Software diversification is a method of changing the program execution so that different instances of the same program operate differently, while still preserving the identical functionality. The aim is to thwart a *global* attack, i.e. prevent an attacker from using information gained from one program execution to attack all other programs in the same way. By introducing software diversification in a program each instance of the program must be attacked individually.

Software diversification is a relatively new technique mostly used to hinder software piracy [6]. Schrittwieser and Katzenbeisser [7] presents a software diversification approach in assembler code. The solution is split into so called gadgets, which contains some assembler instructions each. The gadgets are of varying sizes and different input results in different execution behavior. Dependent on the input, the program executes different paths through the program.

2.4 White-box Cryptography

In a white-box environment the regular black-box model cryptographic algorithms no longer are applicable since they were developed with secure execution environment in mind. These algorithms depend on the secrecy of the cryptographic key, which is easily

acquired by an adversary in a white-box environment by just examining the program code. Consequently, several attempts have been made to evolve regular Data Encryption Standard (DES) and Advanced Encryption Standard (AES) implementations to be resistant to attacks in a white-box environment. The execution of both DES and AES consists of several different operations performed in a specific order. The white-box solutions consist of transforming those operations into a series of lookup tables. Each operation in a white-box solution is done by fetching a value from the lookup tables.

In 2003, Chow et al. published a white-box DES implementation [8], soon followed by an AES application [9]. Later that year Jacob et al. [10] found an attack against the white-box DES implementation by injecting faults during execution. Improvements to the white-box DES implementation have been proposed by Link and Neumann [11] as well as Wyseur and Preneel [12]. In 2007 two independent cryptanalysis on the white-box DES implementations were published by Wyseur et al. [13] and Goubin et al. [14].

The white-box AES implementation was first broken in 2005 with the so called BGE attack presented by Billet et al. [15]. Bringer et al. [16] proposed a whole new white-box AES implementation two years later. In 2009 Xiao and Lai [17] proposed an improvement of the first white-box AES implementation proposed by Chow et al. [9] which was resistant to the attack proposed by Billet et al. [15]. However, both the improvement and the new implementation has been broken through cryptanalysis [18], [19].

3

Advanced Encryption Standard

The Advanced Encryption Standard (AES) is a symmetric block cipher used for encryption and decryption of data. The original design of AES is intended for execution in a black-box environment, i.e; the existence of secure endpoints are assumed. This assumption is however far from always correct and an encryption standard that can resist attacks in a white-box environment is desirable. This chapter will first describe the original AES specification and then continue with explaining the modifications required in order to obtain a white-box solution.

3.1 Black-box AES

The original AES design is specified in FIPS 197 [20] and it is based on the Rijndael block cipher. AES operates on a fixed block size of 128 bits at a time and process the blocks as a 4x4 byte matrix. The blocks are passed through several four-step rounds that mixes the key together with the plaintext in order to produce a ciphertext. Before the block processing begins the key is expanded using Rijndael's key schedule [21]. The key length used can either be 128, 192 or 256 bits. Depending on the key length the number of rounds differ. This thesis will henceforth focus on AES with a key length of 128, which operates on 10 rounds.

Each round the four steps `SubBytes`, `ShiftRows`, `MixColumns` and `AddRoundKey` are performed. Each step is described in detail below. Most of the steps are calculated in the finite field; $\mathbf{GF}(2^8)$ [21].

3.1.1 The SubBytes step

The `SubBytes` step is a non-linear byte substitution where each byte $a_{i,j}$ in the block is replaced with another byte $b_{i,j}$ according to a predefined lookup table called the S-box, see Figure 3.1. The S-box, defined in the Rijndael block cipher, is a 16x16 byte matrix that contains a permutation of all possible 256 byte values. When performing a S-box lookup the 4 leftmost bits of the byte are used as index for the row and the 4 rightmost bits are used as index for the column. Different byte values will therefore never be mapped to the same value. The S-box is constructed to avoid fixed points, $a_{i,j} \neq b_{i,j}$, as well as to avoid opposite fixed points, i.e. $a_{i,j} \oplus b_{i,j} \neq 0xFF$.

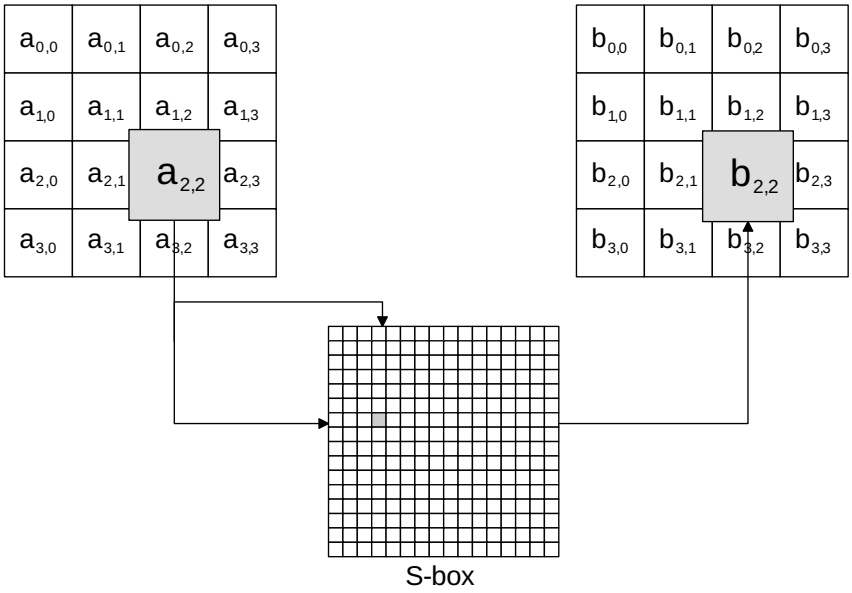


Figure 3.1: S-box lookup performed in the `SubBytes` step.

3.1.2 The ShiftRows step

The `ShiftRows` step is a transposition step where each row of the block are cyclically shifted a certain number of steps. The first row is left unchanged, the second row is shifted once, the third row is shifted twice and the fourth row is shifted three times. The complete `ShiftRows` step is illustrated in Figure 3.2 below.

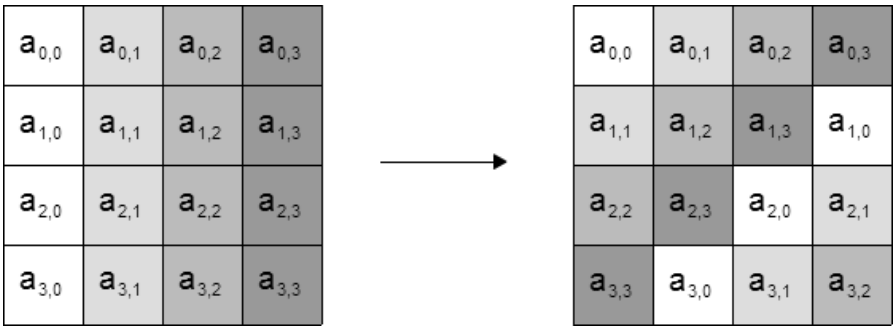


Figure 3.2: The ShiftRows step

3.1.3 The MixColumns step

The `MixColumns` step, operates on the columns of the block. Each byte of the column is combined together will all other bytes of the column and hence transformed into a new value, as shown in Figure 3.3. The transformation is achieved by multiplying the column with a pre-computed matrix, shown here.

$$\begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix}$$

The matrix multiplication are performed in $\mathbf{GF}(2^8)$ and therefore the matrix multiplication can be simplified as follows; when multiplying a value by 01 the value remains the same, and when multiplying by 02 the value is shifted one step to the left. Multiplication by 03 can be implemented as shifting the value one step to the left and then XOR-ing the result together with the original, unshifted value. Addition in $\mathbf{GF}(2^8)$ is simply implemented as XOR.

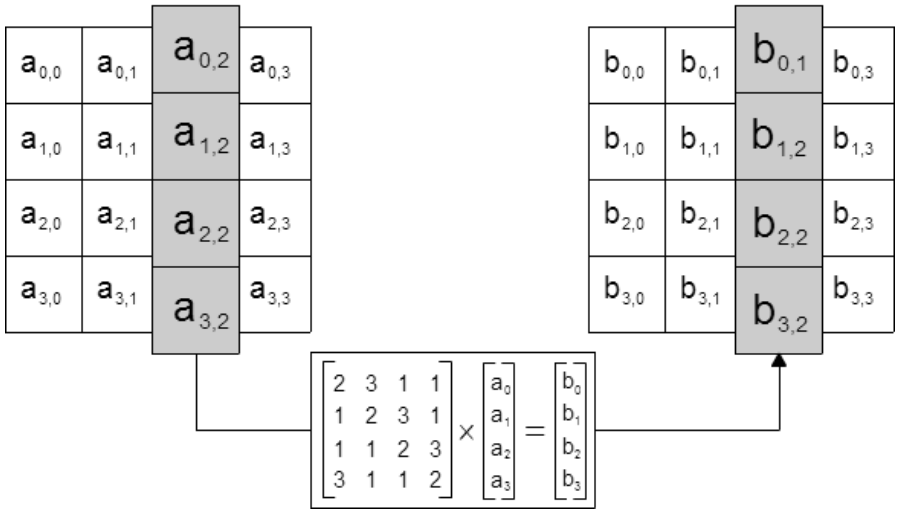


Figure 3.3: The MixColumns step

3.1.4 The AddRoundKey step

The `AddRoundKey` step combines the block and the round key. Each byte of the round key, which has the same size as the block, is added to the corresponding byte of the block using bitwise XOR, as shown in Figure 3.4.

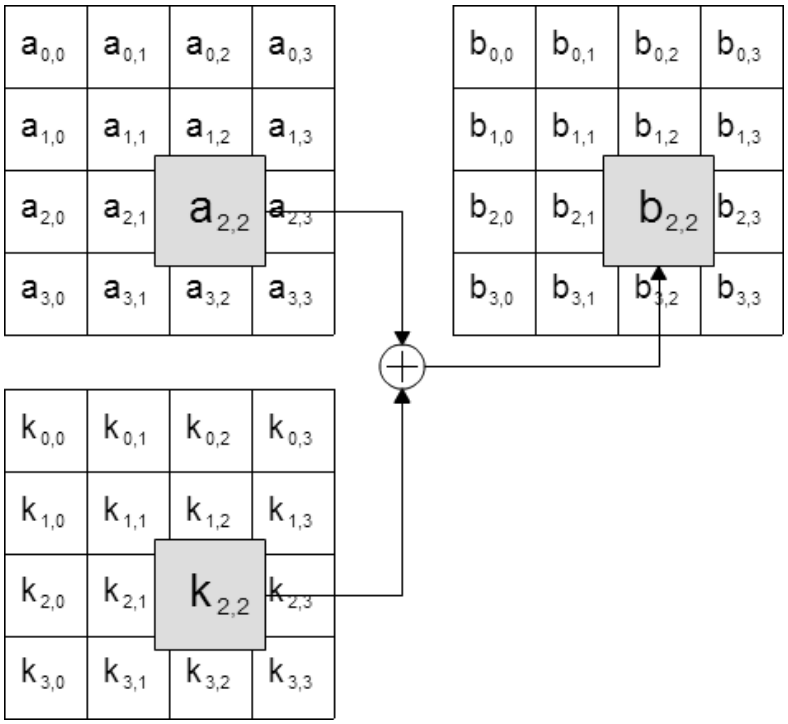


Figure 3.4: The AddRoundKey step

In each round of AES, the steps described above are performed in the order illustrated below in Figure 3.5.

```
block ← plaintext
AddRoundKey(block,k0)
for r = 1...9
    SubBytes(block)
    ShiftRows(block)
    MixColumns(block)
   AddRoundKey(block, kr)
SubBytes(block)
ShiftRows(block)
AddRoundKey(block, k10)
ciphertext ← block
```

Figure 3.5: Pseudocode of the black-box AES

3.2 White-box AES

In contrast to the original black-box AES implementation the white-box AES implementation is based on several lookup tables. The ideal way to implement a secure white-box AES, i.e; with no information leakage, would be to create a huge lookup table that maps the entire plaintext to the corresponding ciphertext with respect to some key. Unfortunately, such a lookup table would be completely impractical because of the massive amount of storage space required for it. For instance, a 128-bit cipher would require $2^{128} \cdot 128$ bits, i.e. $4.95 \cdot 10^{27}$ terabytes of storage space. Chow et al. [9] has presented a more practical solution containing a number of smaller lookup tables. The specification by Chow et. al. is designed around AES with a key length of 128 bits, so called AES-128. However the same obfuscation techniques can be applied to AES-256 as well. The basic idea of Chow's white-box AES implementation is described below.

3.2.1 Rearranging the steps

Starting from the black-box AES steps, shown in Figure 3.5, the `AddRoundKey` step is moved into the for-loop. Also, since in the same S-box is applied to each byte of the block the `SubBytes` step followed by the `ShiftRows` step gives the same result as `ShiftRows` step followed by the `SubBytes` step. The result of these changes are shown in Figure 3.6 below.

`block` $\leftarrow$ *plaintext*
for $r = 1 \ldots 9$
    `AddRoundKey(block,`k_{r-1}`)`
    `ShiftRows(block)`
    `SubBytes(block)`
    `MixColumns(block)`
`AddRoundKey(block,` k_9`)`
`ShiftRows(block)`
`SubBytes(block)`
`AddRoundKey(block,` k_{10}`)`
ciphertext $\leftarrow$ `block`

Figure 3.6: Pseudocode of AES with rearranged steps

Since the `ShiftRows` step is a linear transformation it is possible to perform the `ShiftRows` step on the round key k_{r-1} prior to the actual `AddRoundKey` step, in order to switch places on the `ShiftRows` and the `AddRoundKey` step. The preshifted round key is here denoted as $\hat{k}_{r-1}$, and the resulting steps are as shown in Figure 3.7 below.

```
block ← plaintext
for r = 1...9
    ShiftRows(block)
    AddRoundKey(block, k̂_{r-1})
    SubBytes(block)
    MixColumns(block)
ShiftRows(block)
AddRoundKey(block, k̂_9)
SubBytes(block)
AddRoundKey(block, k_10)
ciphertext ← block
```

Figure 3.7: Pseudocode of AES with rearranged steps and shifted key $\hat{k}_{r-1}$

3.2.2 T-boxes

Combining the S-box from the `SubBytes` step together with the `AddRoundKey` step creates a series of sixteen so called T-boxes, denoted ${T_{i,j}}^r$, which are defined as follows:

$$T^r_{i,j}(x) = S(x \oplus \hat{k}^{r-1}_{i,j}) \qquad \text{for } i = 0,\ldots,3, j = 0,\ldots,3, r = 1,\ldots,9$$
$$T^{10}_{i,j}(x) = S(x \oplus \hat{k}^{9}_{i,j}) \oplus k^{10}_{i,j} \qquad \text{for } i = 0,\ldots,3, j = 0,\ldots,3$$

where the $\hat{k}_{r-1}$ is the preshifted round key. The resulting steps are shown in Figure 3.8:

```
block ← plaintext
for r = 1...9
    ShiftRows(block)
    TBoxes(block)
    MixColumns(block)
ShiftRows(block)
TBoxes(block)
ciphertext ← block
```

Figure 3.8: Pseudocode of AES with T-boxes

3.2.3 Ty_i tables

In the `MixColumns` step the bytes are multiplied with the MC matrix, as described in Figure 3.9.

$$\begin{bmatrix} 02 & 03 & 01 & 01 \\ 01 & 02 & 03 & 01 \\ 01 & 01 & 02 & 03 \\ 03 & 01 & 01 & 02 \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{bmatrix} = y_0 \oplus y_1 \oplus y_2 \oplus y_3$$

$$y_0 = x_0 \begin{bmatrix} 02 \\ 01 \\ 01 \\ 03 \end{bmatrix}, y_1 = x_1 \begin{bmatrix} 03 \\ 02 \\ 01 \\ 01 \end{bmatrix}, y_2 = x_2 \begin{bmatrix} 01 \\ 03 \\ 02 \\ 01 \end{bmatrix}, y_3 = x_3 \begin{bmatrix} 01 \\ 01 \\ 03 \\ 02 \end{bmatrix}$$

Figure 3.9: MixColumns multiplication

The terms on the right of the figure (denoted y_0,y_1,y_2,y_3) are each a function of an 8 bits input. Since each y_i maps 8 bits to 32 bits, they are represented as so-called Ty_i tables which are defined as follows:

$$\begin{aligned} Ty_0(x) &= x \cdot [02\ 01\ 01\ 03]^T \\ Ty_1(x) &= x \cdot [03\ 02\ 01\ 01]^T \\ Ty_2(x) &= x \cdot [01\ 03\ 02\ 01]^T \\ Ty_3(x) &= x \cdot [01\ 01\ 03\ 02]^T \end{aligned}$$

The Ty_i tables maps 8 bits to 32 bits and therefore the results need to be XORed back together to 8 bits. The resulting steps are shown in Figure 3.10. All steps except the `ShiftRows` are now transformed into lookup tables. The `ShiftRows` steps is instead implemented by shifting the input block before the lookup table step.

```
block ← plaintext
for r = 1...9
    ShiftRows(block)
    Tboxes/TyiTables(block)
    XORTables(block)
ShiftRows(block)
TBoxes(block)
ciphertext ← block
```

Figure 3.10: Pseudocode of AES with T-boxes and Ty_i tables

Up to this point, the implementation still functions as a black-box AES. In order protect the AES implementation to withstand attacks in a white-box environment the AES implementation is further modified by adding `Input and output encodings, external encodings and mixing bijections`. These operations will be described in more detail below.

3.2.4 Mixing Bijections

Mixing bijections are invertible linear transformations, they are used in the key-dependant lookup operations in order to achieve diffusion [22]. Without them Chow et al. showed that it's possible to perform a frequency analysis in order to extract the internal encodings. The mixing bijections are selected uniformly at random by generating invertible matrices in GF(2). Moreover Chow et al. recommends that, in order to maximize the diffusion, matrices should be generated by combining several smaller matrices of full rank. A method for doing this is described in an article by Xiao and Zhou [23].

For all rounds except the first, sixteen 8x8-bit mixing bijections are used, one for each byte in the block. The inverses are used as input to the T-boxes in order to cancel out the bijection from the end of the last round.
For all rounds except the last, four 32x32 bit mixing bijections that cancel each other out are used before and after the XOR operations.

3.2.5 Encodings

In order to withstand attacks in a white-box environment all table data needs to be encoded. If the tables were not encoded it would be easy for an adversary to learn the contents of the tables, including the incorporated round keys in the T-boxes/Ty_i tables. The encodings used are both internal, input and output encodings, and external encodings. Both are described below.

Input and Output Encodings

The content of the lookup tables are protected by adding encodings to every table. The encodings adds confusion and are used in a networked fashion, meaning that the output encoding of one table is cancelled out by the input encoding of the next table. The encodings are simply bijections and is selected uniformly at random. Because of the XOR, which operates over 4-bit values, the encodings are created as 4-bit chunks and concatenated together when needed in order to form longer encodings.

External Encodings

As a last step external encodings are applied to the input and output of the entire cipher. This will modify the implementation to instead of mapping raw plaintext to raw ciphertext it will to map encoded plaintext to encoded ciphertext. The external encodings are required in order to prevent an attacker, with either knowledge of the key or decoding functionality, to to retrieve the plaintext. An adversary will not be able to retrieve the plaintext without the knowledge the encodings therefore the encodings should be stored on a secure remote location. Chow et al [9] suggests that the external encodings should be 128-bit to 128-bit mixing bijections.

3.2.6 Table-based implementation

In the white-box AES implementation by Chow et al [9], the tables are divided into 4 types; Type 1, Type 2, Type 3 and Type 4. Type 1 tables consists of the external encodings, which are 32 4-bit encodings and are generated as shown in Figure 3.11. Type 1 tables are applied before the first round and after the last round. Type 2 tables, shown in Figure 3.12, are applied in the first half of each round transformation. Figure 3.12 shows the round transformation for round 2, though round 3 to 9 are applied the same way but with different mixing bijections. In round 1 the inverse mixing bijection L is not applied, and in round 10 the mixing bijection MB is not applied. Hence, round 10 does only contain Type 2 tables. Type 3 tables are tables which cancels the mixing bijection MB as well as adding the mixing bijection L, and Type 4 tables consisting of simple XOR operation, as shown in Figure 3.12.

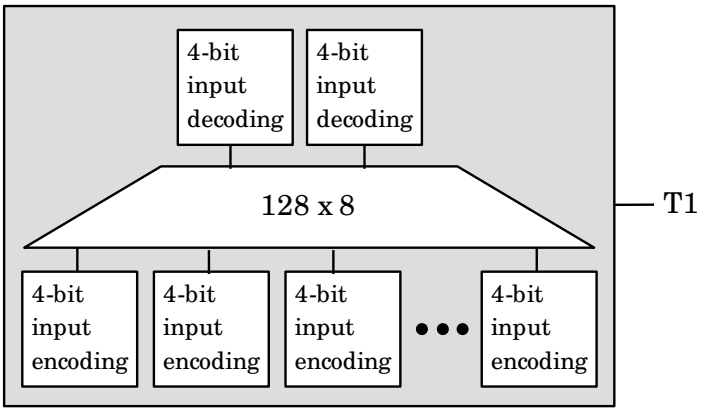


Figure 3.11: A figure of a T1 table presented by Chow et al [9]

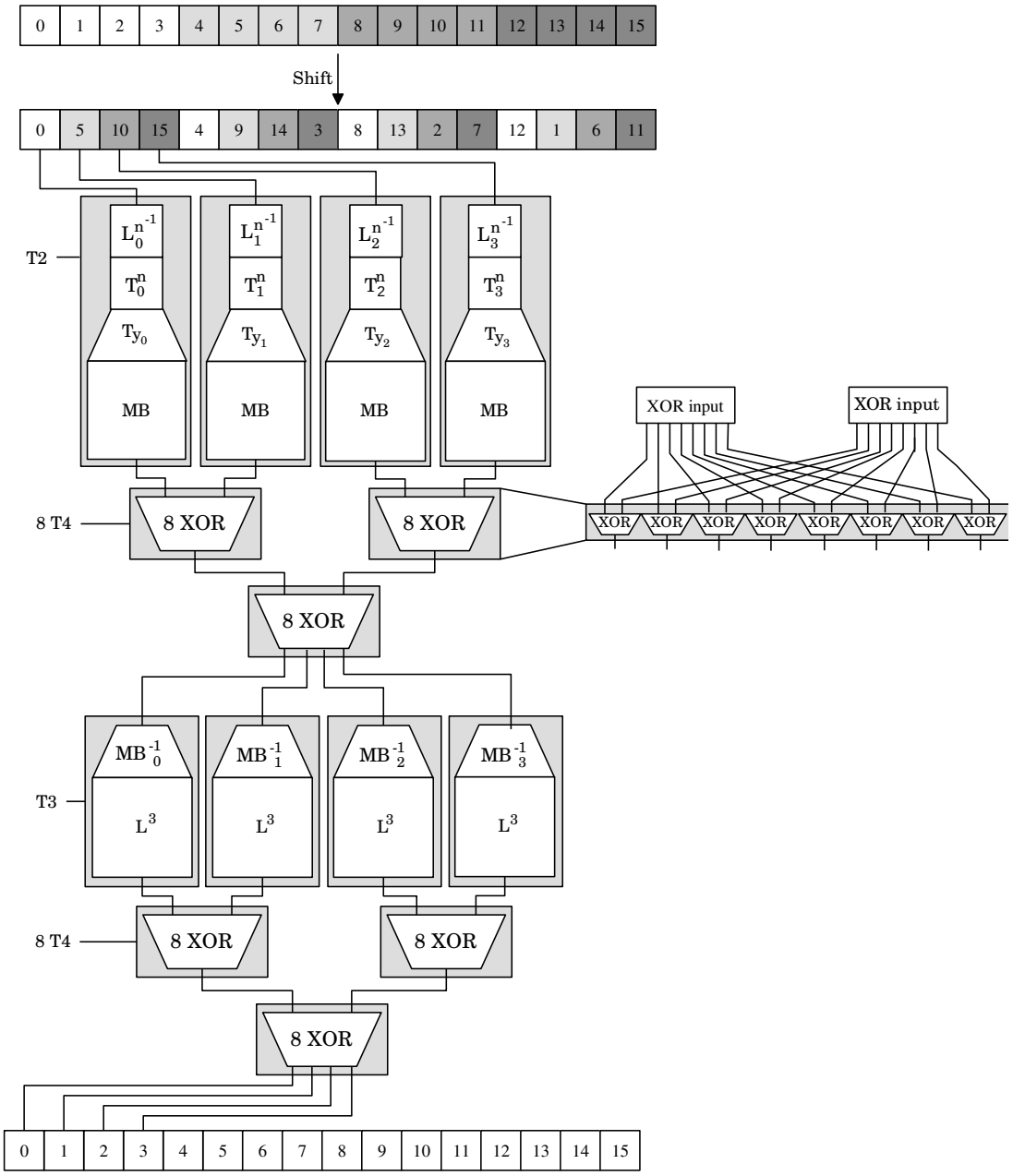


Figure 3.12: Round transformation of AES in round 2, modified figure from [24].

3.2.7 Summarization

To sum up the difference between black-box AES and white-box AES is that the steps described in Figure 3.5 are incorporated into several lookup tables. Instead of storing the key in memory the white-box solution utilizes multiple lookup tables that produces the equivalent result without storing the key in memory at all. In addition the lookup tables in the white-box solution are protected by `input and output encodings, external encodings and mixing bijections`, all which are not a part of the black-box solution. Another difference is the constraint that different keys requires different white-box AES implementations, since the T-boxes/Ty_i tables have the round keys incorporated in the tables. A black-box AES implementation does not have this constraint.

3.2.8 BGE Attack

The paper by Billet et al. [15] presents an algebraic attack against the white-box AES specification by Chow et al. [9]. The attack is named the BGE attack and denotes the initials of the authors. The attack can recover the AES key in 2^{30} computational steps. On a very high level the BGE attack works by analyzing entire AES rounds and interpreting them as a 32-bit to 32-bit transformations. By transforming the non-linear parts to affine transformations in GF(2) and using algebraic analysis the round keys can be obtained. Because of the reversibility of the AES schedule the round keys can be used to recover the AES key. By adding additional obfuscation layers to a white-box AES implementation the BGE attack cannot directly be applied without first reversing the added layers. The implementation can therefore be seen as broken if the obfuscation layers protecting the white box AES implementation are reversed. For that reason we consider the BGE attack out of scope for this report. For more information about the BGE attack please refer to the cryptanalysis by Billet et al. [15].

4

Method

As previously stated, the aim of this Master Thesis is to investigate whether it is possible to obfuscated a source program P so that it is unprofitable for an adversary to extract the secret information. Our work is performed according to the following steps.

- Implement source program P.
- Identify the goal of an adversary.
- Define the knowledge-space and capability of an adversary.
- Attempt to reverse engineer P with the defined adversary knowledge capacity.
- Select obfuscation techniques that makes it more difficult for an adversary to reach his goals.
- Implement the obfuscated program P'.
- Evaluate the total obfuscation quality of P'.
- Attempt to reverse engineer P' with the defined adversary knowledge capacity.
- Compare the reverse engineering process of P' with the reverse engineering process of P.

4.1 Source Program

Our source program P, which later on is obfuscated into P', is a white-box AES implementation. We chose AES encryption since it is widely used and AES make use of a secret which needs protection. Further, we chose to implement a white-box AES instead of a black-box AES since the obfuscated program is designed to be tested in a white-box environment. Although all specifications of white-box AES implementations

have been theoretically broken it is still a valuable method for drastically increasing the work required for extracting the cryptographic key compared to a black-box AES implementation. Practical examples of broken white-box AES implementations are difficult to find and if the implementation differs from the specification in some way the difficulty for breaking it will be even greater.

Our white-box AES implementation is based on the specification given by Chow et al. [9], as described in Section 3.2. The implementation is essentially a generator that takes a key as input and generates a set of lookup tables. The tables then can be exported and later used in order to give either encryption or decryption functionality in a stand-alone solution. A set of loops iterate over each round, column and row in order to pass each byte through the lookup tables in a specific order. The implementation is written in Java. The Java language was chosen because it is good for development. It offers good debugging functionality and can be decompiled easily. This will speed up the development process and the used obfuscation techniques are relatively easy to apply in other syntactically similar languages.

4.2 Threat Model

Before selecting which obfuscation techniques to improve P with, a threat model of an potential adversary needs to be determined. The goal of implementing a obfuscated program P' is to harden the reverse engineering effort made by an adversary to retrieve the secret key. In order to successfully evaluate how the implementation can withstand reverse engineering attacks from an adversary we must define the skill level of that adversary. Apart from determining the skill level, it is also vital to determine the adversary's goal. Yamauchi et al. [25] presents a goal-oriented approach to identify the goal and capacity of a hypothetical adversary, which consists of the following five steps:

Step 1 Define the capability of the adversary.

Step 2 Identify an adversary's goal.

Step 3 Conduct a goal-oriented analysis.

Step 4 For every terminal sub-goal, select obfuscation.

Step 5 Apply the selected obfuscations to the program.

Our implementation was made as an iterative process of development and evaluation. As a result, step 3 to 5 in the list above was performed after each development round with input from the previous evaluation round.

4.2.1 Adversary's Capability

An adversary's capability is a combination of knowledge and understanding as well as the resources the adversary has access to. Yamauchi et al. [25] categorizes the capability

of an adversary into three dimensions; knowledge, system observation and system control. The dimensions are described below.

Knowledge is the level of understanding an adversary has about the program and its behavior.

System observation is the level of observation an adversary can perform on the program. Depending on the level, the adversary might have access to a decompiled version of the program. The adversary might also have a debugger with breakpoint functionality used to gain more understanding of the underlying logic. With the debugger the adversary can observe the internal states as well as the execution trace of the program.

System control is the level of control the adversary has over the system. Depending on the level, the adversary might have access to control the input to the program by controlling the mouse and keyboard inputs. The adversary might also have the ability to change instructions and memory values in any way desired, both before, during and after execution of the program.

Both P and P' are designed to be run in a white-box environment where the adversary has the ability to observe, change and alter everything. Hence, the adversary has maximum level in both the *system observation* and *system control* dimensions. The *knowledge* dimension however, is highly diverse depending on the adversary. The level of knowledge is a combination of the understanding about white-box AES implementations in general, understanding of the different obfuscation techniques being used as well as the understanding of our implementation. The level of understanding of the programming language used is also a part of the *knowledge* dimension. The *knowledge* dimension in our implementation can be defined as:

$$K_{tot} = K_{WBAES} + K_{obf_0} * K_{obf_1} ... * K_{obf_n} + K_{imp} + K_{lang} \tag{4.1}$$

where

- K_{tot} is the total level of knowledge in the knowledge dimension
- K_{WBAES} is the understanding of white-box AES in general
- K_{obf_n} is the understanding of the obfuscation technique n
- K_{imp} is the understanding of our implementation
- K_{lang} is the understanding of the programming language being used

In P no additional obfuscation techniques are used and hence all K_{obf_n} parameters are set to zero. In P' though, the understanding of each additional obfuscation technique are multiplied since all of the techniques are combined.

4.2.2 Goal of the Adversary

The second step of the analysis is to define the goal of the adversary. The primary goal of an adversary using our implementations is to retrieve the encryption key used when encrypting or decrypting. In order to reach the primary goal the adversary might need to reach smaller subgoals, which in turn can be divided into smaller subgoals and so on. Yamauchi et al. [25] presents a goal tree, which is a graph over the relationship between the primary goal and all subgoals. In order to create a goal tree the symbols shown in Table 4.1, originally from the paper by Yamauchi et al. [25], will be used.

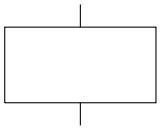
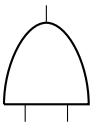
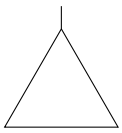
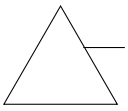
	**Root goal** The final goal of an adversary in attacking the target system.
	**Intermediate goal** A sub-goal decomposed from the parent node (root goal or an intermediate goal). An adversary needs to complete intermediate goals before achieving the root goal.
	**AND gate** A gate that indicates all lower goals must be completed to achieve the higher goal.
	**OR gate** A gate that indicates either one or more goals must be completed to achieve the higher goal.
	**Transfer in** A transfer node connected to a “transfer out” node of other goal tree (child tree).
	**Transfer out** A transfer node connected to a “transfer in” node of other goal tree (parent tree).

Table 4.1: Goal tree symbols.

4.2.3 Reverse Engineering: Source Program

By using the goal tree symbols from Table 4.1 a goal tree has been constructed for the white-box AES source program P, as shown in Figure 4.1 below. The primary goal of an adversary is to retrieve the AES key. Since our source program P is a white-box AES implementation based on the specifications by Chow et al. [9] our implementation also is vulnerable to the BGE attack presented by Billet et al. [15]. The BGE attack was proven to successfully extract the cipher key with a worst time complexity of 2^{30} operations and with negligible space requirements. In order to execute a successful BGE attack an adversary needs the input and output from one AES round. Since the BGE attack has already been demonstrated by Billet et al. [15] we consider the implementation broken if it is possible to retrieve input and output from one AES round. For a deeper insight of the BGE attack we refer the reader to the original paper by Billet et al. [15]

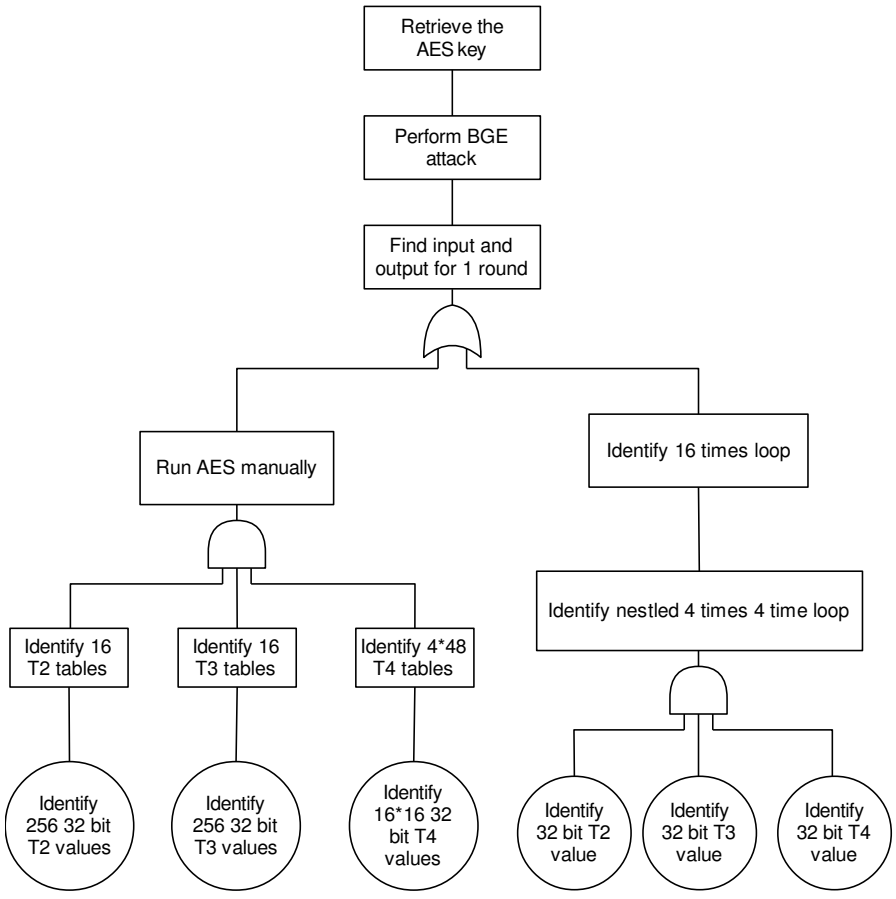


Figure 4.1: The goal tree of white-box AES without obfuscation.

The goal tree analysis in Figure 4.1 shows that the adversary can be successful in retrieving input to the BGE attack in one of two ways. In order to succeed with the

subgoals in the left branch of the tree the adversary need to analyze the code in order to retrieve all tables and later on use those tables to manually run a specially crafted white-box AES implementation. P only performs table lookups that are relevant to the given input, i.e. a subset of all possible lookups, which forces the adversary to run the application several times in order to retrieve the tables which makes this approach both tedious and time-consuming. The other approach, as can be seen in the right branch of the goal tree, is to statically analyze the decompiled code. The nestled loops are easily recognizable and with knowledge of the white-box AES specification the location of the round data handling can be identified easily. Since reaching the subgoals in the right branch of the goal tree is considered to be the easiest way to go the reverse engineering of P, focus has be on harden these subgoals.

Below in Listing 4.1 is a short code snippet of the source program given. As can be seen in the code the loops for each round, column and row are easily identified without the need of executing the program. In the most inner for loop the T2 operation is made by fetching a value from the look up table array `T2`.

Listing 4.1: Code snippet of source program.

```
public static byte[] encrypt(byte[] in) {
...
 for(int l = 0; l < len; l += 16) {
 byte[] block = new byte[l]
 ...
  for (int i = 0; i < ROUNDS; i++) {
  ...
   for (int c = 0; c < COLUMNS; c++) {
   ...
    for (int r = 0; r < ROWS; r++) {
      int idx = shift[c * 4 + r];
      result[r] = T2[i][c * 4 + r][block[idx]];
    }
    ...
   }
   ...
  }
  ...
 }
 ...
}
```

Retrieving the input and output of one AES round from P was rather easy. The only effort required was to analyze the program and identify the when the next round begun. Generating a control flow graph of the program has been proven to be a profitable approach when performing analysis of a program. Below in Figure 4.2 a generated control flow graph of P is shown. The control flow graph is generated by an Eclipse plugin called Eclipse CFG Generator [26]. Besides illustrating the program complexity

the graph can also be used as a complement to the code in order to gain a better understanding of the underlying logic.

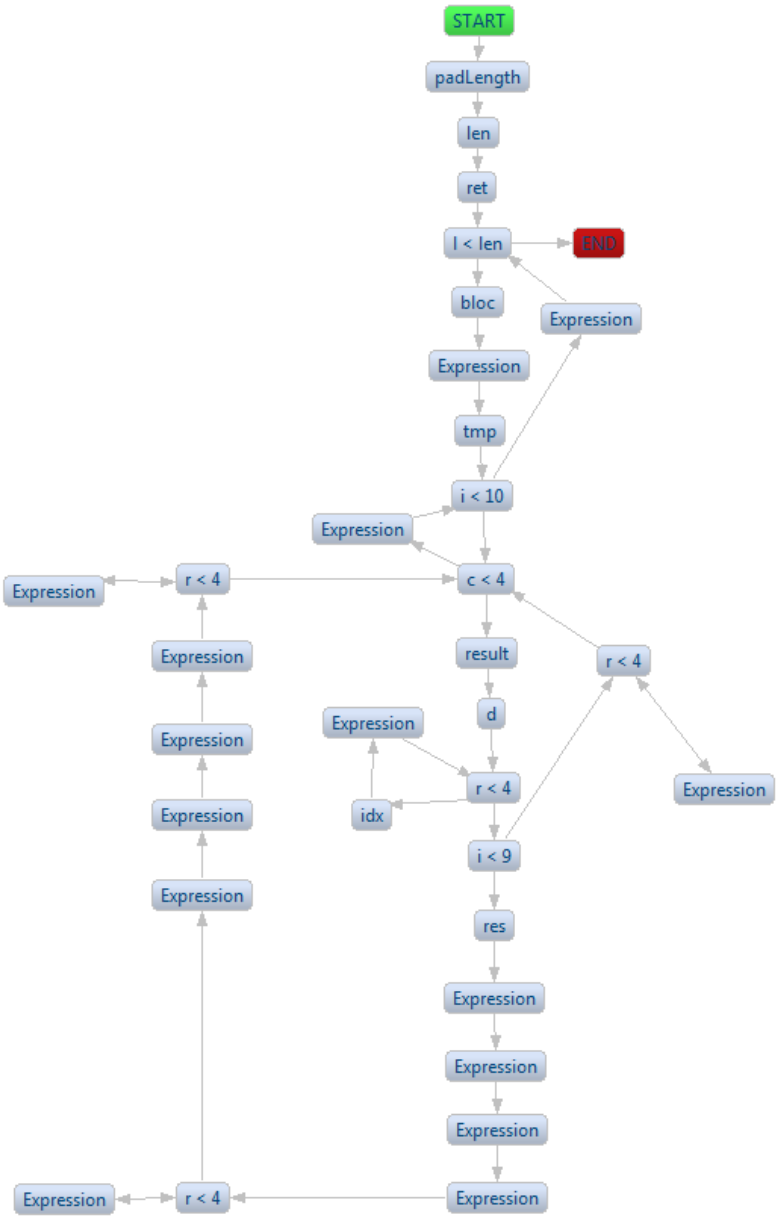


Figure 4.2: Generated control flow graph from the unobfuscated white-box AES implementation.

As can be seen in Figure 4.2, the control flow of P is rather straight forward. Even though the names of the nodes in the graph are not that descriptive it is still easy to identify some of the underlying logic, such as for-loops. With the knowledge of how white-box AES is constructed, our adversary could easily identify the points where the table lookups are performed just by looking at the control flow graph. With help from the control flow graph, it is easy to identify where a new round begins in the code, shown in Listing 4.1. Simply adding a printout to the code at the end of the for-loop of a round was the only effort required to retrieve the input and output of one round. Hence, the effort to reverse engineer the source program can be considered as low.

4.3 Implementation Process

The implementation of P' has been developed in an iterative process, with the following steps:

- Identify a 'weak point' in the program which simplifies the reverse engineer process.
- Find an obfuscation technique that hardens the reverse engineering of the found weak point.
- Add the obfuscation technique to the implementation.
- Reverse engineer the improved implementation.

The reverse engineering process has been done in parallel with the continuous implementation of P'. After each iteration of the implementation, an attempt to reverse engineer the improved implementation was made. Hence, both the implementation choices and the reverse engineering process is described below.

The obfuscated program P' is an implementation that combines white-box AES with several traditional obfuscation techniques. The traditional obfuscation techniques provide an additional layer of protection that protects the white-box AES data tables. The obfuscation techniques are not limited to any specific white-box AES implementation and should work, with little or no modification, together with any white-box AES implementation. Together the obfuscation techniques in P' aims to provide AES-128 encryption functionality without revealing the key nor giving the ability to decrypt any data. Even though the implementation is written in Java the methods used can be applied to most syntactically similar languages.

The organized way of storing and applying the lookup tables in a plain white-box AES implementation make it relatively easy for an adversary to extract data needed in order to perform an attack such as the BGE attack [15]. To hinder the extraction of such data we have created a generator that takes white-box AES table data as input and outputs an obfuscated version of the otherwise plain implementation. This section will describe how different obfuscation techniques were used in order to achieve this.

4.3.1 Reverse Engineering Process

In order to test how well the obfuscated program P' withstands adversaries, we created an automatic tool to reverse engineer the implementation, hereby called round data extractor (RDE). The implementation process of the RDE tool was done in an iterative manner, hence different versions; RDE-1, RDE-2, RDE-3 and RDE-4. All versions of the RDE tool was implemented by parsing the source code of P' in order to identify the various operations. This was done by modifying the source code to output the current state at different points of interest. The modified program was then executed and the state information was used to extract the round data. Since P is a white-box AES implementation then P' is considered broken if it is possible to retrieve the input and output on one round which is needed to launch a BGE attack [15].

P' is written in Java, thus the code is compiled and packaged into a jar file. In order to retrieve the source files the jar file needed to be decompiled. For that purpose, the Java decompiler JAD [27] was used. JAD runs through the command prompt and expects a jar file as input. It interprets the Java byte-code and outputs Java source code that are functionally equivalent to the original source. The decompiled source files was then imported into the Java IDE Eclipse. [28]. During the reverse engineering process the Eclipse debugger was used as aid to analyze the code.

Defining Our Adversary

After each iteration of the implementation an attempt to reverse engineer that implementation is being performed. In order to fully evaluate how hard the reverse engineering process is we need to define the knowledge and abilities of the adversary making these reverse engineering attempts - our adversary. As presented in Section 4.2.1 an adversary's capability is highly dependent on its knowledge level . Equation 4.1 presents all parameters that results in the total knowledge of an adversary. Since we reverse engineered our own code, all K_{obf_n} as well as the K_{imp} parameter and can be considered to be extremely high. Also, since our source program is a functioning white-box AES implementation the parameter K_{WBAES} can be considered to be high as well. The last parameter, K_{lang}, might not be as high as the other parameters but it is still considered to be quiet high. Hence, the total knowledge domain of our adversary is considered to be exceedingly high. Considering that we reverse engineered our own code, our adversary most likely has a significantly higher insight in the underlying logic compared to any other adversary. Therefore, the knowledge of our adversary is considered to be much higher than an adversary who has never seen the code before.

One other important aspect to keep in mind is that our adversary has the advantage of performing the reverse engineering in iterations. Which means that all knowledge from previous iterations can be used in the next iteration. This is also an advantage that any other adversary would not have.

4.3.2 Control Flow Flattening

The analysis of the adversary's goal, represented by a goal tree shown in Figure 4.1, demonstrates that P is vulnerable to both static and dynamic analysis. When performing analysis of a program, generating a control flow graph of the program has been proven to be a profitable approach. The reverse engineering of P showed that it was very easy to identify the for loops with the help of a control flow graph. Hence, the control flow of the program is the weakest point of P. According to our literal research we evaluate that the control flow flattening obfuscation technique is the best obfuscation technique against constructing control flow graphs, which is an important step in static analysis. Also, control flow flattening can harden the underlying logic of the program and hence indirectly harden the dynamic analysis as well. Therefore, we choose control flow flattening to be implemented in the first iteration of the obfuscated program P'.

In our implementation, the control flow flattening technique, as described in Chapter 2, is applied to the set of loops that control the lookup operations. For each loop, the lookup values are extracted and stored as separate methods. The methods are stored in a randomized order and are called in the correct order by a dispatcher node during execution. In addition to performing the lookup operations, each method will modify two global variables that together are used to calculate a random hash value. The random hash value is used by the dispatcher node in order to decide what method to execute next. This means that by just looking at one specific method there is no way to tell which method will execute next. In order to get the general structure of how the obfuscation technique altered the behavior of the program, a short example is presented in Listing 4.2 below. The pseudo code is merely an illustrative example which will be used to illustrate how the program changes over the iterations. It is not vital for the reader to understand the complete logic of the code, and hence only a brief explanation will be given.

The `swVar` variable is the dispatcher node, and the variables `h` and `s` together with the current dispatcher node is hashed together to determine which next case statement that will be executed. It is important to note that the hash is calculated of the value of the variables and not the name of them. The reason why the case value are so large is because they are in fact hash values calculated in previous case statements. In each case statement one table lookup is performed, for example T2, and stored in the `block` variable. The block variable is 16 bytes large represented by a 2x2 matrix and hence ranges from [0][0] to [3][3]. The table lookups are also matrices with various lengths. The T2 and T3 operations has three matrix levels which corresponds to round, columns and byte. To simplify the byte is in the example represented by `x`. The value of `x` is in P' represented as an integer. The T4 operation on the other hand, has four matrix levels since it is an XOR operation between two bytes. In order to simplify one of the bytes are represented as a number and the other one as `x`. The four matrix levels are; current row, which of the XOR operation in the XOR chain that should execute, and lastly the two byte that should be XOR:ed with each other.

Listing 4.2: An pseudo code example of P' after adding the control flow flattening technique.

```
switch swVar
    case -2090260694 :
        block[2][1] = T2Table[5][3][x];
        h-=17; s*=32; swVar = hash(swVar,h,s);
    case 1245918113 :
        block[3][0] = T4Table[3][13][57][x];
        h*=4; s-=1; swVar = hash(swVar,h,s);
    case 8329260 :
        block[0][2] = T3Table[2][7][x];
        h-=6; s/=17; swVar = hash(swVar,h,s);
    case -202756108 :
        block[3][2] = T4Table[1][32][9][x];
        h+=50; s*=3; swVar = hash(swVar,h,s);
    ...
    case 268264046 :
        block[3][3] = T2Table[5][15][x];
        h/=3; s-=7; swVar = hash(swVar,h,s);
    case -1819546496 :
        return block;
    case 742416915 :
        block[1][3] = T4Table[0][27][30][x];
        h-=10; s+=43; swVar = hash(swVar,h,s);
    case 561877267 :
        block[0][0] = T3Table[6][9][x];
        h/=7; s*=6; swVar = hash(swVar,h,s);
    case 1557566154 :
        block[2][2] = T4Table[2][39][110][x];
        h-=0; s+=14; swVar = hash(swVar,h,s);
    case 804012105 :
        block[0][1] = T2Table[3][15][x];
        h*=5; s-=189; swVar = hash(swVar,h,s);
```

Compared to the code snippet of P, shown in Listings 4.1 under Section 4.2.3, there is no longer any signs of loops in the implementation which was the ultimate goal with adding control flow flattening. The example above also illustrates that fact that by just looking at the `swVar` variable it is not possible to determine which case statement that will be executed next.

4.3.3 Reverse Engineering: Control Flow Flattening

The first iteration of the P' implementation included the control flow flattening obfuscation technique. The technique flattens the control flow of the program and it is no longer possible to get any useful information from a generated control flow graph. The generated control flow graph, shown in Figure 4.3, no longer reveals any underlying logic.

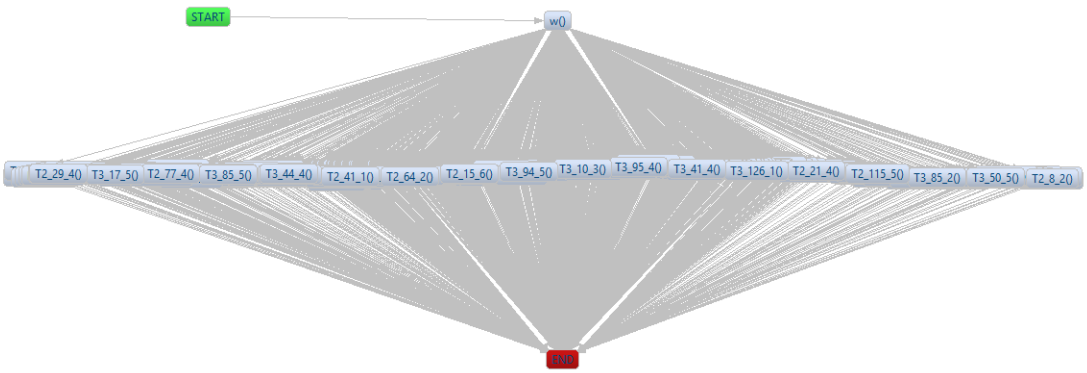


Figure 4.3: Generated control flow graph from the control flow flattened implementation. The figure is meant to illustrate that a control flow graph of a control flow flattened program is essentially useless.

Even though the control flow of the program no longer consists of for-loops it is still rather easy to identify the table lookups when analyzing the code. This because lookup tables are stored together in arrays which are easily identified. This issue is well illustrated in the example presented in Listings 4.2. As can be seen in the example, the `T2`, `T3` and `T4` tables are stored as multidimensional arrays. By dynamically analyzing the flow of the program during execution the actual lookup can be determined when the arrays containing the lookup tables are accessed. By using a debugger little effort was required to extract the input and output of one AES round. Hence it was possible for our adversary to get the correct data without implementing any tool.

Adding the control flow flattening obfuscation technique altered the subgoal of the adversary a bit. This is illustrated by redefining the goal tree, shown in Figure 4.4. Only the right branch of the goal tree has changed, and as showed the adversary no longer can identify the rounds by finding for loops. The weak point of the implementation is still the right branch of the goal tree, since the adversary can statically and dynamically analyze the code. Control flow flattening did however hinder the adversary from generating a control flow graph which is a good aid when analyzing code.

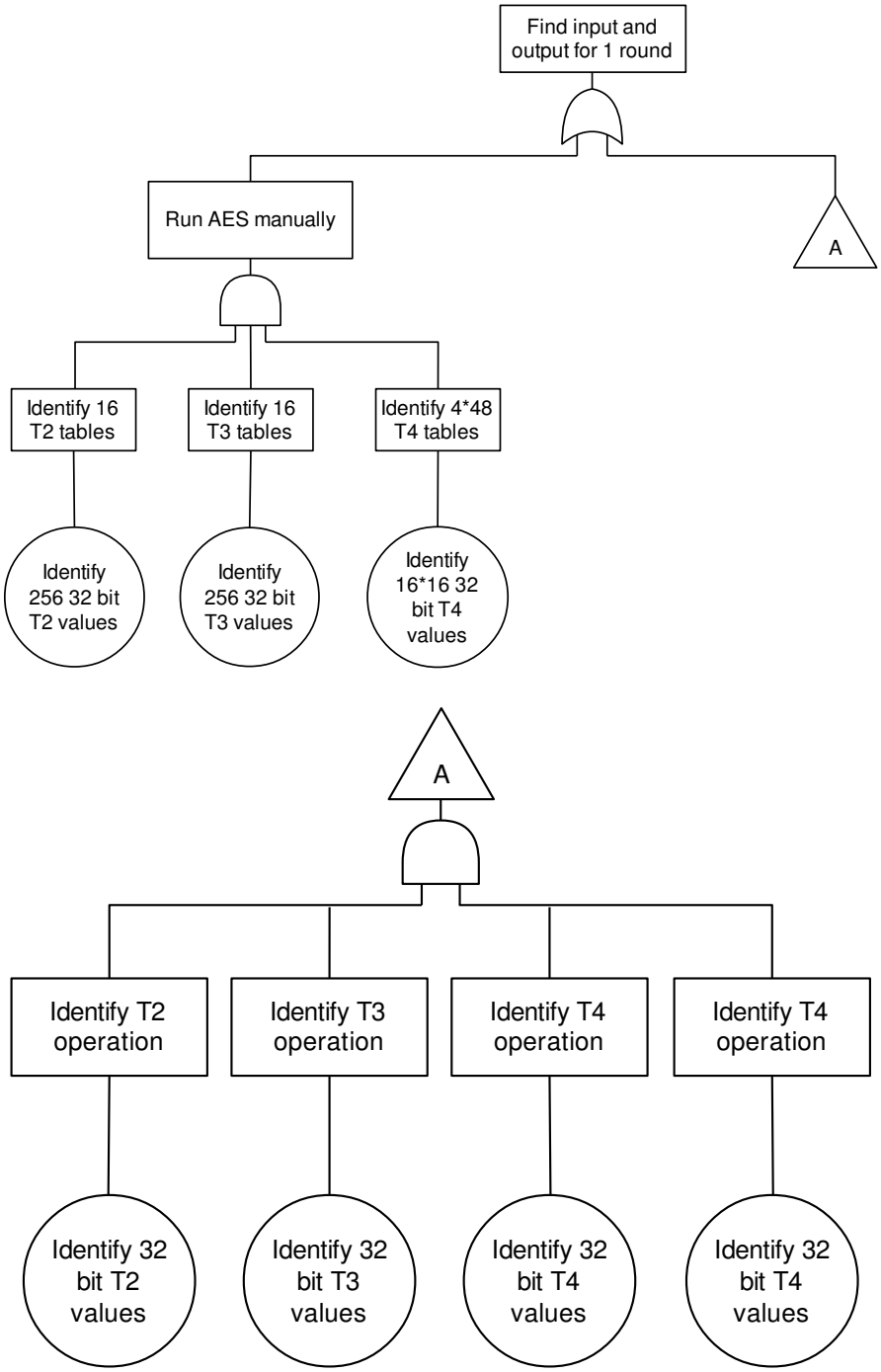


Figure 4.4: The goal tree of P' with control flow obfuscation

4.3.4 Data Obfuscation I

The reverse engineering of P' with just the control flow flattening obfuscation technique was proven not to be sufficient enough. To hinder an adversary from just reading the lookups when the lookup tables are accessed this data needs to be stored in unconventional ways. The next obfuscation technique to be implemented is data obfuscation where the ordering of data will be transformed.

Normally the lookup tables are stored as multidimensional maps, typically organized after round, column and row. Each of these tables contain a final table that maps input values to different output values. Instead of storing these tables as multidimensional maps it is possible to split them up and put them into separate methods. However, the most inner table is operating on the program input rather than the state and can therefore not be transformed into separate methods.

Listings 4.3 below shows the same example as presented in Listings 4.2. There is not much difference between the two except that instead of performing table lookups on multidimensional arrays, each look up is now performed in its own method. As previously stated the most inner table is operation on the program input rather than the state, each method will take a input value as argument. In the example below each method takes a byte `x` as input and outputs the correct operation on the given byte. This means that inside each method there is a lookup based on 128 different values. This is considered a flaw but it is not possible to work around. The focus lies instead on trying to harden the understanding so that the adversary has trouble reaching this conclusion.

Listing 4.3: An pseudo code example of P' after the first data obfuscation technique was added.

```
switch swVar
    case -2090260694 :
        block[2][1] = ApplyT2Table_5_3(x);
        h-=17; s*=32; swVar = hash(swVar,h,s);
    case 1245918113 :
        block[3][0] = ApplyT4Table_3_13_57(x);
        h*=4;  s-=1; swVar = hash(swVar,h,s);
    case 8329260 :
        block[0][2] = ApplyT3Table_2_7(x);
        h-=6;  s/=17;  swVar = hash(swVar,h,s);
    case -202756108 :
        block[3][2] = ApplyT4Table_1_32_9(x);
        h+=50; s*=3; swVar = hash(swVar,h,s);
    ...
    case 268264046 :
        block[3][3] = ApplyT2Table_5_15(x);
        h/=3;  s-=7; swVar = hash(swVar,h,s);
    case -1819546496 :
        return block;
    case 742416915 :
        block[1][3] = ApplyT4Table_0_27_30(x);
        h-=10; s+=43; swVar = hash(swVar,h,s);
    case 561877267 :
        block[0][0] = ApplyT3Table_6_9(x);
        h/=7;  s*=6; swVar = hash(swVar,h,s);
    case 1557566154 :
        block[2][2] = ApplyT4Table_2_39_110(x);
        h-=0;  s+=14; swVar = hash(swVar,h,s);
    case 804012105 :
        block[0][1] = ApplyT2Table_3_15(x);
        h*=5;  s-=189; swVar = hash(swVar,h,s);
```

4.3.5 Reverse Engineering: Data Obfuscation I

Changing so that the lookup tables no longer are stored in arrays is proven to be a good method to confuse the adversary. Still, one major problem with the implementation is that the names of the variables and methods are still revealing much information. Usually for program variables a descriptive name is used to help the programmer to be able to identify the underlying logic without execution. However, the descriptive names also helps the adversary in understanding the code. So even though the control flow is altered and the data is stored separately, the descriptive names helps to keep track of what is being executed when using a debugger. For example, in Listings 4.3 above, the method name `ApplyT2Table_5_3` suggests that the operation is of type `T2` in round 5, column 3. By debugging the relevant methods it was possible to get the input and output of one AES round, thus the effort to reverse engineer P' at this stage was by our

adversary considered to be rather low. Again, since it was easy to reverse engineer with just a debugger no tool was developed.

Altering the storage of the lookup tables is not affecting the adversaries goals, hence the goal tree is unchanged. This because the data obfuscation technique only alters the understanding of the program, rather than actually altering the behavior of the program. The adversary still needs to identify the various operations, though adding data obfuscation has made that a bit harder than in previous iteration.

4.3.6 Layout Obfuscation

As previously stated, an adversary can make use of descriptive identifier names so naturally the next obfuscation technique to implement is the layout obfuscation technique where all identifiers are scrambled. Layout obfuscation also refers to removing comments. However, removing comments is not needed because the Java compiler already removes them during compilation.

All identifiers need to be named so they don't leak any information on what their purpose is. In our implementation the variables are simply given non-descriptive names and for the methods random strings are generated. The real method names are then mapped to the generated strings and replaced in the implementation. The randomly generated strings are different for each instance and each instance will also have different ordering of all method calls and declarations.

Below in Listings 4.4 the pseudocode example is presented with all named scrambled. As can be seen it is impossible to guess which method that performs which operation, compared to the descriptive names shown in Listings 4.3.

Listing 4.4: An pseudo code example of P' after the layout obfuscation technique was added.

```
switch Zcx
    case -2090260694 :
        jsZ[2][1] = UWD5y5ue(x);
        s-=17; h*=32;  Zcx = hash(Zcx,s,h);
    case 1245918113 :
        jsZ[3][0] = y9FKSQKg(x);
        s*=4;  h-=1; Zcx = hash(Zcx,s,h);
    case 8329260 :
        jsZ[0][2] = MB9c84SX(x);
        s-=6;  h/=17; Zcx = hash(Zcx,s,h);
    case -202756108 :
        jsZ[3][2] = rFuZkgnU(x);
        s+=50; h*=3; Zcx = hash(Zcx,s,h);
    ...
    case 268264046 :
        jsZ[3][3] = UpRE9c6k(x);
        s/=3; h-=7; Zcx = hash(Zcx,s,h);
    case -1819546496 :
        return jsZ;
    case 742416915 :
        jsZ[1][3] = NssnGrGp(x);
        s-=10; h+=43;  Zcx = hash(Zcx,s,h);
    case 561877267 :
        jsZ[0][0] = n6aCnTBB(x);
        s/=7;  h*=6; Zcx = hash(Zcx,s,h);
    case 1557566154 :
        jsZ[2][2] = cSQS3b6j(x);
        s-=0;  h+=14; Zcx = hash(Zcx,s,h);
    case 804012105 :
        jsZ[0][1] = wrpJT3Ph(x);
        s*=5; h-=189; Zcx = hash(Zcx,s,h);
```

4.3.7 Reverse Engineering: Layout Obfuscation

The level of effort required in the reverse engineering process really increased during this iteration. Any previous analysis of the code did not give much knowledge on how to reverse engineer the implementation this time and it was no longer possible to reverse engineer just by using a debugger. As can be seen in the examples shown in Listings 4.4 and Listings 4.3 the method name `ApplyT2Table_5_3` was scrambled to `UWD5y5ue`. Our adversary did not have the luxury of having to identical implementations where one was scrambled and one was not as in the examples. Hence, another way than just analyzing the code was needed to reverse engineer P' at this iteration. After thorough debugging of the program flow, a pattern was identified. The found pattern made out the order of which each operation was executed as well as where the result was stored. Once these key parts were identified it was rather easy to create the first version of

the RDE, RDE-1, which kept track on which operations had been executed. Once all operations from one round had been executed the value currently in memory was the output of said round. Since previous reverse engineering processes has been done manually by using a debugger the effort no longer is considered to be low but rather medium.

Even though this iteration raised the bar for the adversary the goal tree is still unchanged. Once again, the obfuscation technique being added only altered the understanding of the program.

4.3.8 Data Obfuscation II

In the previous iteration, the easiest way to reverse engineer the implementation was to keep track of which order the operations were executed. Although the control flow has been altered and the data storage been randomized the first method that was executed correspond to the first lookup operation. To harden the identification of which operation that is executed all operations that does not depend on each other will be run in a randomized order. This forces the adversary to keep track of more data simultaneously as well as forcing the adversary to identifying which operation that is executed each time.

Another problem arisen from the previous reverse engineering iteration is the storage of intermediate results. Intermediate results from each operation have to be stored somewhere. Normally when a byte is being processed the result will overwrite the old value and therefore corresponding sub-results will be stored in a fixed memory space. An attacker could monitor this memory area and extract data of interest. To hinder this all sub-results will be stored in an unstructured fashion. The current executing method determines where the data should be fetched from and stored to in the succeeding method.

In the example below, shown in Listings 4.5, the second data obfuscation technique was added to P' and has altered the program further. Each method is extended to also return the values of `i` and `j` which determines where the next intermediate result will be stored. Apart from this change, the variables `s` and `h` has changed values to illustrate the random execution order of operations, compare to the previous example in Listings 4.4.

Listing 4.5: An pseudo code example of P' after the second data obfuscation technique was added.

```
switch Zcx
    case -2090260694 :
        jsZ[i][j], i, j = UWD5y5ue(x);
        s+=42;  h+=1; Zcx = hash(Zcx,s,h);
    case 1245918113 :
        jsZ[j][i], i, j = y9FKSQKg(x);
        s*=18;  h/=84; Zcx = hash(Zcx,s,h);
    case 8329260 :
        jsZ[j][i], i, j = MB9c84SX(x);
        s-=109; h/=57; Zcx = hash(Zcx,s,h);
    case -202756108 :
        jsZ[i][j], i, j = rFuZkgnU(x);
        s/=22;  h*=4;  Zcx = hash(Zcx,s,h);
    ...
    case 268264046 :
        jsZ[i][j], i, j = UpRE9c6k(x);
        s+=19;  h-=5;  Zcx = hash(Zcx,s,h);
    case -1819546496 :
        return jsZ;
    case 742416915 :
        jsZ[j][i], i, j = NssnGrGp(x);
        s/=76;  h+=13; Zcx = hash(Zcx,s,h);
    case 561877267 :
        jsZ[i][j], i, j = n6aCnTBB(x);
        s-=90;  h*=5;  Zcx = hash(Zcx,s,h);
    case 1557566154 :
        jsZ[i][j], i, j = cSQS3b6j(x);
        s*=10;  h+=3;  Zcx = hash(Zcx,s,h);
    case 804012105 :
        jsZ[j][i], i, j = wrpJT3Ph(x);
        s*=52;  h-=6;  Zcx = hash(Zcx,s,h);
```

4.3.9 Reverse Engineering: Data Obfuscation II

Once again the effort required to reverse engineer the implementation has risen. The development of RDE-1 was based on the assumption that all operations executed in a specific order. Also, some assumption was made regarding where in the program the results, both intermediate and final, was stored. Since these assumption no longer were valid, RDE-1 no longer functioned as expected. Also, as shown in Listings 4.5 the lookup methods returned values which determined where to store the next calculated result. Since this was not the case when developing RDE-1 the tool did not work at all. So even though an automatic tool already had been implemented adding data obfuscation techniques hardened the reverse engineering process quite a lot. A new analysis of the program behavior had to be done to identify new patterns. The RDE was developed into RDE-2 which helped a lot to keep track on the unconventional storage

of the results. Also, RDE-2 could keep track on which operations that succeeded each other and therefore bypass the randomized execution order of operations. The required effort to retrieve the input and output of one AES round was by our adversary considered to be some where between medium and hard, but the possibility to retrieve the data needed for a BGE attack [15] is still considered to be relatively easy. Once again the goals of the adversary has not been changed, even though it is now somewhat more hard to achieve. However, if the adversary did not have any prior knowledge of the program then it might have been harder to reverse engineer.

4.3.10 Bogus Operations

With knowledge of the exact operations that are performed in an white-box AES implementation an attacker could compare the the executed operations to the operations that should be executed according to the specification. By doing this an attacker could identify when operations of interest occur and extract data from them. To counter this, bogus operations is inserted in the implementation. Bogus operations look and behave exactly like real operations, the result from the bogus operations will be stored in the same way as real operations and may also later be used as input for other bogus operations. The bogus operations will however not have an affect on the computation of the final result. The bogus operations are chosen and inserted randomly for each instance. This means that no two instances will perform the same operations. In our implementation the majority of operations are bogus operations to harden the identification of real operations. Also, the bogus operations are run in such a way that removing them from the program would break the flow of the program and the program would stop functioning. As illustrated in the example below, shown in Listings 4.6, there is no apparent difference between real and bogus operations. In the example additional comments has been added to show the reader which operations that in fact is bogus. There are no such comments in the original code. Another aspect is also that P' has a majority of bogus operations but in the example below only a few was added. To be a complete representation of the code, the example should have at least a dozen more bogus operations. These are however emitted to shorten the example since the point was only to show that it is not possible to distinguish between real and bogus operations without executing the code.

Listing 4.6: An pseudo code example of P' after the bogus operations was added.

```
switch Zcx
    case -2090260694 :
        jsZ[i][j], i, j = UWD5y5ue(x);
        s+=42;  h+=1;  Zcx = hash(Zcx,s,h);
    case 1267505648 :
        jsZ[i][j], i, j = jbK3M4RZ(x); // Bogus
        s+=62;  h-=10; Zcx = hash(Zcx,s,h);
    case 1245918113 :
        jsZ[j][i], i, j = y9FKSQKg(x);
        s*=18;  h/=84; Zcx = hash(Zcx,s,h);
    case 8329260 :
        jsZ[j][i], i, j = MB9c84SX(x);
        s-=109; h/=57; Zcx = hash(Zcx,s,h);
    case -50017354 :
        jsZ[j][i], i, j = UupCHU9K(x); // Bogus
        s*=14;  h-=2;  Zcx = hash(Zcx,s,h);
    case -202756108 :
        jsZ[i][j], i, j = rFuZkgnU(x);
        s/=22;  h*=4;  Zcx = hash(Zcx,s,h);
    ...
    case 268264046 :
        jsZ[i][j], i, j = UpRE9c6k(x);
        s+=19;  h-=5;  Zcx = hash(Zcx,s,h);
    case -1819546496 :
        return jsZ;
    case 742416915 :
        jsZ[j][i], i, j = NssnGrGp(x);
        s/=76;  h+=13; Zcx = hash(Zcx,s,h);
    case 561877267 :
        jsZ[i][j], i, j = n6aCnTBB(x);
        s-=90;  h*=5;  Zcx = hash(Zcx,s,h);
    case 856817361 :
        jsZ[j][i], i, j = jbK3M4RZ(x); // Bogus
        s+=1;   h/=79; Zcx = hash(Zcx,s,h);
    case 1893017956 :
        jsZ[j][i], i, j = F8vzzbxJ(x); // Bogus
        s-=24;  h+=31; Zcx = hash(Zcx,s,h);
    case 1557566154 :
        jsZ[i][j], i, j = cSQS3b6j(x);
        s*=10;  h+=3;  Zcx = hash(Zcx,s,h);
    case 804012105 :
        jsZ[j][i], i, j = wrpJT3Ph(x);
        s*=52;  h-=6;  Zcx = hash(Zcx,s,h);
```

4.3.11 Reverse Engineering: Bogus Operations

During this reverse engineering process it was vital to keep track of which operations that was real and which one that was not. The definition of a real operation is that a

real operation affects the outcome of the final result in round 10, i.e. the value had an impact at the whole encryption.

RDE-2 was developed to during run-time keep track on where the execution were in the program flow, i.e. which round was executed. The first attempt for RDE-3 was to just develop RDE-2 further to keep track only of the real operations. However, since the bogus operations behave exactly like real operations, with the only difference that they do not affect the end result, it was really hard to distinguish between real and bogus operations. Hence, the first attempt of RDE-3 identified several so called false positives, i.e. identified bogus operations as real. It was hard to implement RDE-3 to discard bogus operations since they were hard to identify due to their real behavior. So when it was hard for our adversary to distinguish between real and bogus operations, as illustrated in the example in Listings 4.6 above, it was nearly impossible to try to 'learn' the RDE to see the difference. Some obvious bogus operations could however sometimes be identified. For example, a `T2` result used as input to another `T2` operation. This is an obvious bogus operation since a `T2` operation is followed by an XOR, or `T4`, operation, as is illustrated in 3.12. Removing such obvious bogus operations from the code also proved to be a bad approach as the control flow is dependent on that all methods, including bogus ones, are executed in the right order for the program to function correctly. After several attempts to fix the first version of RDE-3 the whole idea of keeping track of the operations as they run was discarded. Instead, a new idea was used when developing RDE-3. In the worst-case scenario the bogus operations would perform the exactly same operations as the real through out all rounds. In that case, the only way to identify the real operations is to observe which results that are being used when returning the output. The operation that was made right before saving the end result is by definition a real operation, and so is the operation that happened before that one. Hence, new attempt for RDE-3 was to keep track of the operations in the reverse way. Therefore by beginning to track the operations backwards, from the values returned as output to the values used as input while still keeping track on the operations executed the final version of RDE-3 could successfully identify the input and output of all rounds. Even though it was possible to successfully reverse engineer this iteration the effort for reverse engineering was by our advanced adversary considered to be exceedingly high.

Since this iteration altered the logic of the program the adversary's goal shifted and a new goal tree was drawn, shown in Figure 4.5. The only addition to the previous goal tree is that the branch of identifying bogus operations was added.

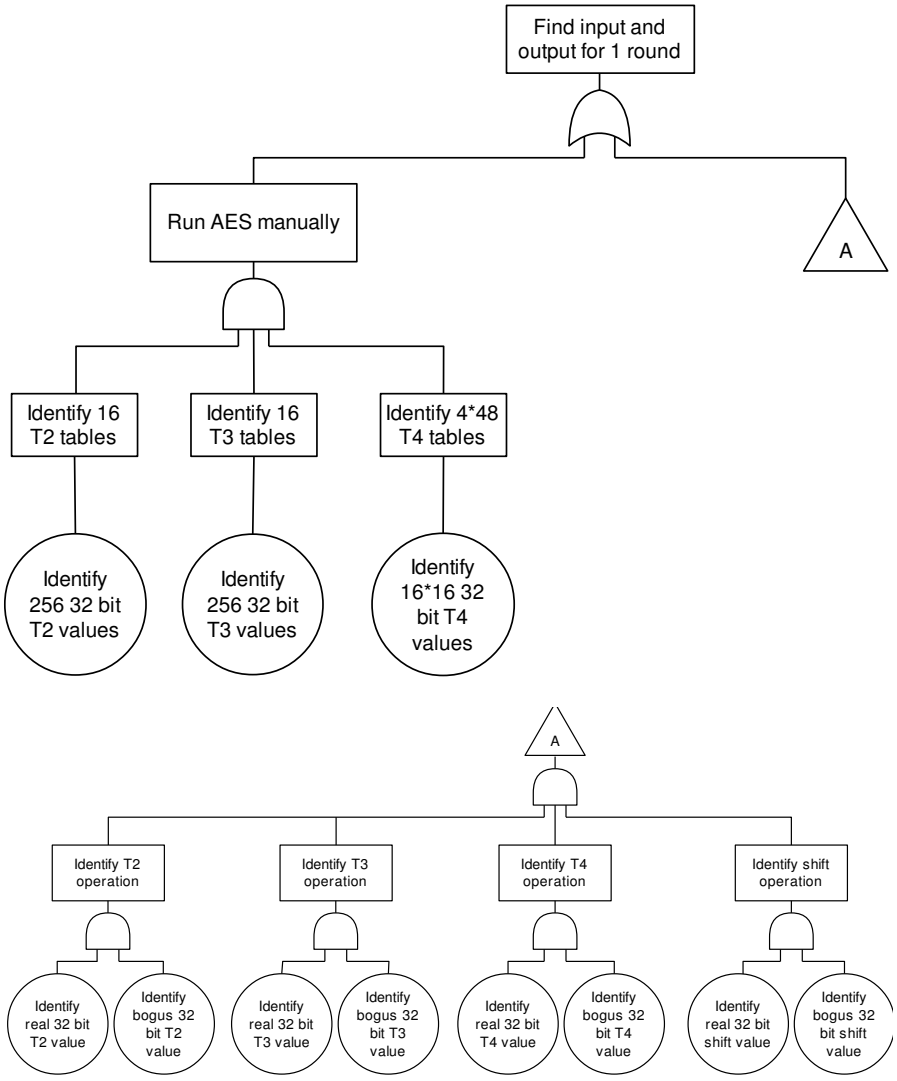


Figure 4.5: The goal tree of P' after adding bogus operations

4.3.12 Memory Shuffling

The last functionality added was memory shuffler. In arbitrary parts of the program operations were added that would swap two random elements in the intermediate result memory structure. This means that values that may be required for the next operation would now have been swapped with another one. This should cause further confusion as an adversary would have to keep track of the memory swap operations and immediately take it into consideration as the next operation may be dependent on it.

There is not much difference between the example shown below, Listings 4.7 and the

previous example in Listings 4.6. Below there are a few more case statements where the internal structure of the intermediate results stored in the array `jsZ`. Since the array is shuffled based on values in the array the adversary must execute the program to know which values that are shuffled.

Listing 4.7: An pseudo code example of P' after memory shuffling was added.

```
switch Zcx
    case -2090260694 :
        jsZ[i][j], i, j = UWD5y5ue(x);
        s+=42;  h+=1;  Zcx = hash(Zcx,s,h);
    case 1540322536  :
        jsZ[i][j] = jsZ[jsZ[i-82]][jsZ[j+5]];
    case 1267505648 :
        jsZ[i][j], i, j = jbK3M4RZ(x); // Bogus
        s+=62;  h-=10; Zcx = hash(Zcx,s,h);
    case 1245918113 :
        jsZ[j][i], i, j = y9FKSQKg(x);
        s*=18;  h/=84; Zcx = hash(Zcx,s,h);
    case 8329260 :
        jsZ[j][i], i, j = MB9c84SX(x);
        s-=109; h/=57; Zcx = hash(Zcx,s,h);
    case -50017354 :
        jsZ[j][i], i, j = UupCHU9K(x); // Bogus
        s*=14;  h-=2;  Zcx = hash(Zcx,s,h);
    case -202756108 :
        jsZ[i][j], i, j = rFuZkgnU(x);
        s/=22;  h*=4;  Zcx = hash(Zcx,s,h);
    ...
    case 268264046 :
        jsZ[i][j], i, j = UpRE9c6k(x);
        s+=19;  h-=5;  Zcx = hash(Zcx,s,h);
    case -1819546496 :
        return jsZ;
    case 742416915 :
        jsZ[j][i], i, j = NssnGrGp(x);
        s/=76;  h+=13; Zcx = hash(Zcx,s,h);
    case -1270516304 :
        jsZ[i][j] = jsZ[jsZ[j+6]][jsZ[i+51]];
    case 561877267 :
        jsZ[i][j], i, j = n6aCnTBB(x);
        s-=90;  h*=5;  Zcx = hash(Zcx,s,h);
    case 856817361 :
        jsZ[j][i], i, j = jbK3M4RZ(x); // Bogus
        s+=1;   h/=79; Zcx = hash(Zcx,s,h);
    case 1893017956 :
        jsZ[j][i], i, j = F8vzzbxJ(x); // Bogus
        s-=24;  h+=31; Zcx = hash(Zcx,s,h);
    case 1557566154 :
        jsZ[i][j], i, j = cSQS3b6j(x);
        s*=10;  h+=3;  Zcx = hash(Zcx,s,h);
    case -2035379708 :
        jsZ[i][j] = jsZ[jsZ[j-33]][jsZ[i+9]];
    case 804012105 :
        jsZ[j][i], i, j = wrpJT3Ph(x);
        s*=52;  h-=6;  Zcx = hash(Zcx,s,h);
```

4.3.13 Reverse Engineering: Memory Shuffling

The last iteration of the implementation added memory shuffling to the program. RDE-3 did not work as intended and a minor upgrade was needed. RDE-3 needed to keep track of the memory locations of all intermediate data and once an memory shuffling operation was executed RDE-3 had to keep track on that swap. RDE-4 was developed from RDE-3 with the fix to keep track of memory shuffling.

Even though this iteration only had a small impact on the RDE, adding memory shuffling was a useful obfuscation technique. If an adversary did not have an automatic tool from previous reverse engineering iterations, then the memory shuffling would have hardened the reverse engineering process quiet a bit. Hence, if an adversary only had gotten the final version of P' and no prior reverse engineering iterations, the effort for retrieving the key would be considered to be extremely high.

The goal tree of the final version of P' is shown in Figure 4.6 below. Compared to the previous goal tree, the memory shuffling obfuscation technique has been added as a subgoal.

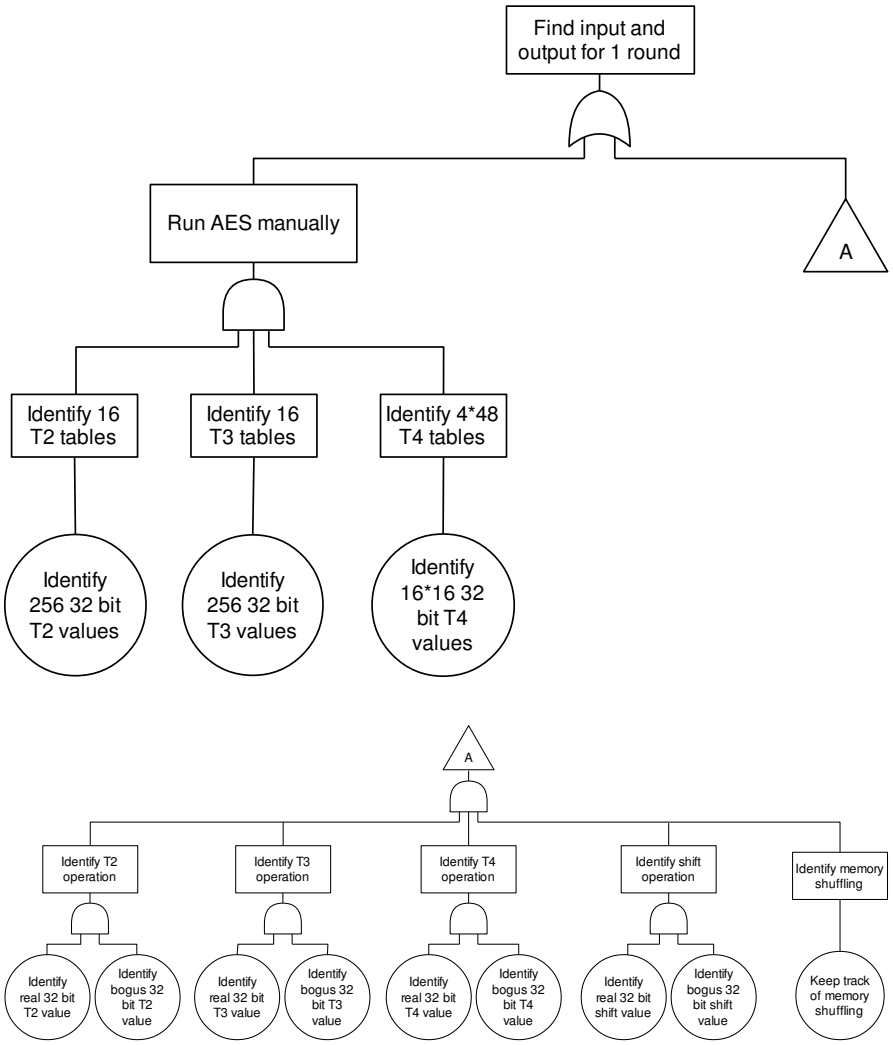


Figure 4.6: The goal tree of white-box AES with obfuscation

The left side of the tree looks exactly the same as the goal tree for the source program, shown in Figure 4.1. However, by splitting the arrays into separate methods these goals are no longer as easily reached as in the goal tree of the source program. On the other hand, the right side of the goal tree has been totally altered. Instead of identifying nestled for-loops as in the source programs goal tree, the adversary now needs to identify each operation separately.

4.4 Reverse Engineering Tool Summary

As presented in Section 4.3.1, the automatic tool used in the reverse engineering process was named Round Data Extractor or RDE. For each iteration the RDE was developed further and to separate them apart we named them by numbers; RDE-1, RDE-2, RDE-3 and RDE-4.

The first two versions of the tool, RDE-1 and RDE-2, was developed to help keep track on which operation in which round that was currently executed. Our adversary's knowledge helped to develop the RDE since the adversary was well aware of in which order the different operations was executed. Also, the tools was based on several assumptions that was true for the given iteration. After each iteration, some of the assumptions was no longer true and the previous version of the tool did not longer work.

After the bogus operations was added to the implementation the previous version of the tool started to behave unexpectedly. The tool identified bogus operations as real operations because bogus operations behave, with the exception that they do not affect the final output, exactly like real operations so it was impossible to identify bogus operations until the result was either discarded or saved. By the end of the execution it was possible to identify which of the last operations that was real and then work you way backwards. Therefore, RDE-3 was developed so that during run-time a multi-rooted tree structure was built with the 16 bytes input as roots and each operation represented by nodes which had one or more parents. After the whole program finished executing the tree was complete. RDE-3 then traversed through the tree from the bottom to the top, choosing the parents of each real operation. By doing this the real operations could be identified because the bogus operations would never be traversed. In the example in Figure 4.7 below the green path is the path that RDE-3 traverses in the tree. As can be seen, no bogus operations are traversed. Note that the program traverses the tree from the bottom up.

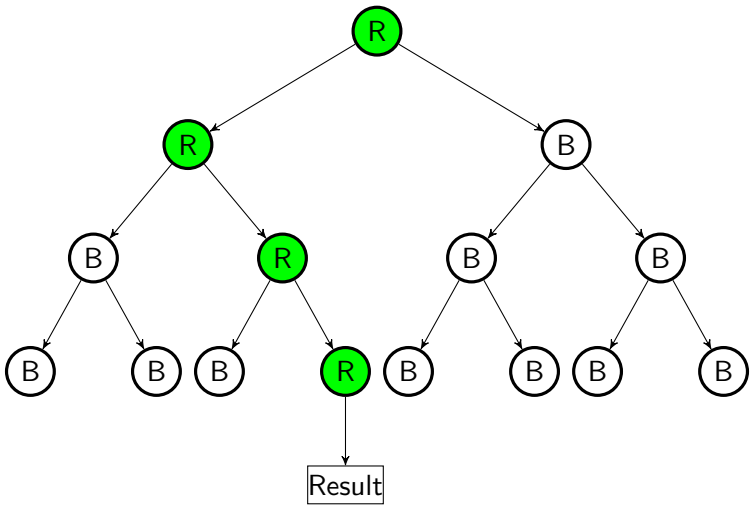


Figure 4.7: An example of the tree structure in RDE-3. R nodes represent real operations and B nodes represent bogus operations.

In order to know which round the operation belonged to, our adversary once again used the knowledge of which operations that are executed during one round. Because of how the tree was traversed, all bogus operations were filtered out. RDE-4 was a development of RDE-3 that also kept track of memory shuffling.

Due to how the tool works the running time of all versions of the RDE tool is basically the same as running the obfuscated program P' as normal, i.e, depending on the input length; from a few hundred milliseconds to a few seconds. The extra execution time introduced by parsing the source code and the added operations from the tool is negligible.

RDE-1 and RDE-2 only helps keeping track on which round P' is executing by inspecting the operation. After a finite set of operations has been executed, a new round begins. So, when P' is done executing, the tool is done executing. The complexity for both RDE-1 and RDE-2 is $\mathcal{O}(n)$.

RDE-3 and RDE-4 needs to execute whole P' to construct the tree structure. Then the tool traverses the tree and in worst case it traverses the whole tree. If each round has m number of operations and there are p rounds then the time complexity for both RDE-3 and RDE-4 is $\mathcal{O}(n + mp)$. Traversing the tree introduces some extra execution time, but it is still negligable. This because the RDE only traverses real operations in the tree, rather than bogus ones, and the program contains a majority of bogus operations.

4.5 Java Limitations

Java comes with a number of limitations that have to be taken into consideration. Other languages may give better resistance against a reverse engineering attack, the techniques used are however not language dependant and can be used in any language that supports the required functionality. One major drawback with the Java language is that it's very easy to restore high-level source code from the compiled byte-code. Several tools that offer automatic decompilation, such as JAD [27], exists and require no special knowledge to use. Compared to other compiled languages, such as C / C++, that produces machine code Java is easy to analyze and understand. However an adversary with good knowledge in assembly can reverse engineer machine code using the same approach as with Java. Another limitation of Java is that the size of methods are limited to 2^{16} bytes. This can be an issue with control flow flattening as the entire program flow will be transformed into a huge switch case. This is however easily circumvented by splitting the switch case into several smaller ones and having the default case referring the program flow to the next switch case.

4.6 Final implementation

The final implementation is a combination of all of the techniques described above. The generator will depend on many randomly generated numbers and by feeding the generator with different seeds no two instances will be the same. Identifiers will have different names and order, lookup tables and intermediate results will be stored differently and bogus operations will differ in placement and quantity between all instances. Below a small example snippet of the resulting code can be seen.

Listing 4.8: Code snippet of obfuscated implementation.

```
switch (hash(h, s)) {
  case 2083867827:
    A();
    break;
  case 122799172:
    B();
    break;
  ...
  case 530909204:
    C();
    break;
}
...
void A() {
  m[i[0]] = D(m[i[1]], m[i[2]]);
  s += -4722;
  h = 2083867827;
  i = new int[] { 74, 69, 5 };
}
...
long D(long l0, long l1) {
  switch(l0) {
    case 121:
      return 3654854678L;
    case 34:
      return 503624924L;
    ...
    case 47:
      return 835892428L;
  }
}
```

As can be seen in Listing 4.8, the dispatcher node is represented as a switch case. The switch case operates on a has value of two parameters `h` and `s`. It is this hash that determines which method should be executed next. In the example above `A`, `B` and `C` are such methods representing one operation each. In the middle of the Listing, method `A` is shown as an example on how the methods look. `A` changes both `s` and `h` so that the right method is executed after `A`. `A` calls on a method `D` which takes as input two values from the memory array `m` and performs an operation. As can be seen `D` performs an operation on only one of the input. However, `D` takes 2 values as input anyway in order to make the methods hard to distinguish from each other. The result returned from `D` is stored in a new position in `m`. The current method `A` alters the variable `i` which determines the next method's memory allocation. Hence, it is extremely hard to manually analyze this implementation.

5

Evaluation Results

Measuring the efficiency of an obfuscation technique has proven to be somewhat difficult. Most obfuscation techniques focuses on increasing the complexity of a program, i.e. making it harder to understand the underlying function of the program. The word complexity in this context refers to how complicated a program is and should not be confused with time complexity. The level of complexity of a program is rather hard to measure since it is highly individual and there are no specifications to follow. Therefore another measure for evaluating how well an obfuscation technique works is needed. Collberg et al. [2] have defined three different types of measurements that can be used for evaluate the quality of an obfuscation technique, namely; potency, resilience and cost. In order to test the obfuscation techniques used in our implementation these evaluation methods are used.

As defined in Definition 1 a source program P is transformed into a target program P', where P' is the obfuscated version of P. In our case the source program P is a plain white-box AES implementation with no additional obfuscation techniques added, according to the specification given by Chow et al [9]. Our target program P' is a obfuscated version of the source program P, as described in Chapter 4.3. Note that hereafter P' refers the implementation with all obfuscation techniques added, i.e. the final implementation.

5.1 Reverse Engineering

Since P is a white-box AES implementation it is vulnerable to white-box AES attacks such as the BGE attack [15]. Our goal was therefore to obfuscated P so that the obfuscated program P' would harden the defense against white-box AES attacks. In order to successfully implement the BGE attack the input and output of at least one round of the white-box AES is required. Due to the obfuscations of P' it is not possible to perform

a BGE attack on P' without reverse engineer the program first, in order to retrieve the input and output of one round. Hence, we consider the program broken if it is possible to retrieve those values since it is then possible to perform the BGE attack.

In all iterations of the implementation the input and output of all rounds was found, with more or less effort. As an aid in the reverse engineering process an automatic tool was developed. As we had access to the correct round data it was possible to compare the output generated by the automatic tool and draw conclusions whether the changes made were correct or not. Any other adversary would not have the ability, making the development process of such a tool significantly more difficult. In order for an adversary to know if the output of the automatic tool is correct or not the adversary needs to perform the actual BGE attack on the output from the automatic tool. The attack then might return a valid encryption key which the adversary then must use to encrypt some input. The adversary also must encrypt the same input with P' and then compare the encrypted cipher texts with each other. Only then, the adversary would know if the automatic tool had retrieved the correct information. In other words, constructing an automatic tool would be a rather tedious work since the only way to validate the output of the tool would be running the BGE attack.

5.2 Obfuscation evaluation

Since our P' program contains several different obfuscation techniques in unison, the program as a whole will be evaluated. The total quality of an obfuscation is a combination of the potency, resilience and cost, defined in Definition 5. Hence P' is evaluated in terms of those three measurements, which are presented below.

5.2.1 Source Program

In order to successfully evaluate the quality of the final implementation P', a comparison to the source program P is made. However, firstly the unobfuscated white-box AES implementation P is evaluated. P is compared to an implementation of a regular black-box AES, described in Section 3.1, used in white-box environment.

Even though the white-box AES has been broken in theory, it provides much more obfuscation than a regular black-box AES implementation. The total understanding of a white-box AES implementation depends on the understanding of white-box AES in general as well as how the white-box AES is implemented. This might seem rather straightforward but since there are several white-box AES specifications published, see section 2.4, an adversary has to understand the white-box AES specifications used. Even though the basics are pretty much the same some key elements differ in the various specifications.

5.2.2 Potency Definition

Potency measures the complexity of the program. The complexity of a program is decided by several things such as the number of predicates, nesting levels and interblock dependencies. Basically potency can be viewed as a measure of how difficult it is for a human to analyze and understand the code. Since the complexity of a program is highly dependent on the observer it is a rather vague measurement. Collberg et al [2] attempts to concretize the term by modifying complexity formulas used in Software Engineering as shown in Table 5.1. The higher potency an obfuscation has the harder reverse engineering becomes. In order to increase the potency the measures in Table 5.1 should be maximized. The scale for the measurements are *low*, *medium* and *high*.

Table 5.1: Overview of some popular software complexity measures. $E(X)$ is the complexity of a software component X. F is a function or method, C a class and P a program.
Table originally from [2].

Metric	Metric name	Citation
μ_1	Program Length	
	$E(P)$ increases with the number of operators and operands in P	
μ_2	Cyclomatic Complexity	
	$E(F)$ increases with the number of predicates in F	
μ_3	Nesting Complexity	
	$E(F)$ increases with the nesting level of conditionals in F	
μ_4	Data Flow Complexity	
	$E(F)$ increases with the number of inter-basic block variable references in F	
μ_5	Fan-in/out Complexity	
	$E(F)$ increases with the number of formal parameters to F, and with the number of global data structures read or updated by F	
μ_6	Data Structure Complexity	
	$E(P)$ increases with the complexity of the static data structures declared in P. The complexity of a scalar variable is constant. The complexity of an array increases with the number of dimensions and with the complexity of the element type. The complexity of a record increases with the number and complexity of its field.	
μ_7	OO Metric	
	$E(C)$ increases with the number of methods in C, the depth (distance from the root) of C in the inheritance tree, the number of direct subclasses of C, the number of other classes to which C is couples[1], the number of methods that can be executed in response to a message sent to an object of C, the degree to which C's methods do not reference the same set of instance variables.	

[1] Two classes are coupled if one uses the methods or instance variables of the other

The formal definition of potency, given by Collberg [2], and is shown below.

Definition 2. *(Transformation Potency)*
Let τ be a behavior-conserving transformation, such that $P \xrightarrow{\tau} P'$ transforms a source program P into a target program P'. Let $E(P)$ be the complexity of P, as defined by one of the metrics in Table 5.1.
$\tau_{pot}(P)$, the *potency* of τ with respect to a program P, is a measure of the extent to which τ changes the complexity of P. It is defined as

$$\tau_{pot}(P) \stackrel{\text{def}}{=} E(P')/E(P) - 1$$

τ is a *potent obfuscation transformation* if $\tau_{pot}(P) > 0$.

□

5.2.3 Potency Evaluation

The control flow of the source program P is rather straight forward. Figure 4.2 shows a generated control flow graph of our implementation of P, consisting of several for-loops containing table lookups which are easily identified. However in the obfuscated program P' the control flow is drastically altered. As seen in the Figure 5.1 the control flow flattening changes the flow graph so that it consists of a enormous switch case. Since unraveling this flattened version requires dynamic analysis P' is now protected against static analysis.

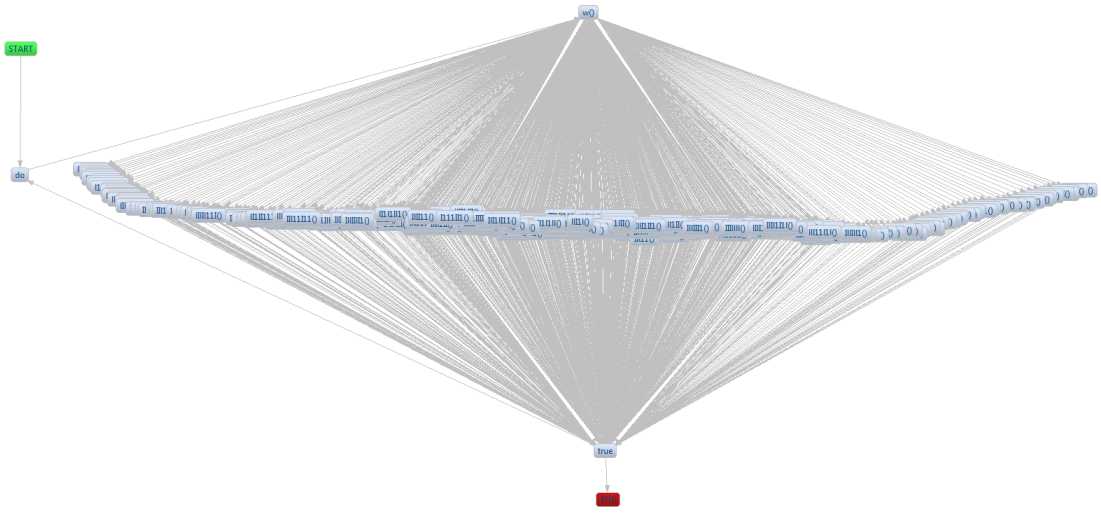


Figure 5.1: Generated control flow graph from the control flow flattened implementation

According to Definition 2 the metrics in Table 5.1 help determine the total potency. Because of how the obfuscation is made some of the metrics are roughly the same in both P and P'. The metrics that differ are presented in Table 5.2 below as a comparison between the two implementations. The metrics of the obfuscated white-box AES will vary slightly between different generated instances and the values presented are an example from one instance. The measurements was made by using a small script that counted the number of occurrences of certain identifiers. As can be seen in Table 5.2, P' has significantly higher values on all measurements, i.e; the potency has increased compared to P. By comparing the control flow graphs, Figure 4.2 and 5.1, it is not surprising that the program length (μ_1) has increased, in fact the measured program length of P' is over 4000 times more than P. The second metric (μ_2) determines the cyclomatic complexity,

i.e. the number of predicates. In this case the cyclomatic complexity is mainly a result of the switch cases that handles over half a million different building blocks. The last metric (μ_4) is the measurement of the data flow complexity. This is a measurement on the number of inter-basic block variable references in methods. Because all methods in the main switch case of P' operate on global variables this metric is also considerably higher.

Measurement	Potency P	Potency P'
μ_1	425	1 766 070
μ_2	2	527 799
μ_3	6	2
μ_4	8	62 883
μ_5	34	64 143
μ_6	11	1

Table 5.2: Comparison of potency measurements from Table 5.1.

The measurements μ_2, μ_4 and μ_5 are measured as a sum, meaning that for all methods the values are added together to a total that represents the entire program. On the other hand, the μ_3 measurement represents the maximum nesting level found in the program and is therefore represented as the highest nesting level rather than of the sum of nesting levels. The μ_1 value is measured as a total, representing the total program length. The measurement μ_7 is not taken into consideration as it measures different class properties such as inheritance which is non-existent in both P and P', and hence is not present in Table 5.1.

Splitting the lookup tables in P' and storing each value in its own method increases the potency a great deal, to a *high* potency level. Considering that the differences between the lookup tables are very small it is difficult to reconstruct the lookup tables from the methods. In P though, the lookup tables are stored as multidimensional maps which are rather easy to identify. The fact that they are stored together makes it easy for an adversary to extract the tables and run its own copy of the white-box AES. The adversary will then have a much greater chance at retrieving the input and the output from one round.

Scrambling the name of all identifiers through layout obfuscation is considered to add *medium* potency and the fact that the names differ between generated instances makes it even more difficult to try to reverse the process. Any possible comments are removed while compiling which is considered to have a *high* potency level.

In order to confuse a human analyzer more, bogus operations have been added to obfuscate P' even further. With a majority of bogus operations, the process of identifying the real operations can be lengthy. Bogus operations also make the program longer which implies that the adversary needs to process more information in order to gain understanding of the code, which results in a *high* potency level.

5.2.4 Resilience Definition

While potency is a measure of complexity in term of human understanding, resilience is a measure to determine how well an obfuscation technique can withstand an automatic deobfuscator. The level of resilience can be seen as both the effort required creating an automatic deobfuscator as well as the time and space complexity required for such deobfuscator to effectively reduce the potency of the program. It is vital to differentiate between potency and resilience since potency aims to confuse a human reader, while resilience aims to confuse an automatic deobfuscator. For instance, introducing more conditional statements will increase potency but it will not necessarily increase the resilience. The resilience depends on how difficult it is to determine the outcome of a given condition. A condition that simply compares two constant values would offer very low resilience because an automatic deobfuscator could easily evaluate it and remove the falsified path.

Resilience, as defined by Collberg et al [2], is measured on a scale from *trivial* to *one-way*, shown in Figure 5.2a. An obfuscation technique which is classified as one-way cannot be undone. For instance if some information is removed that is vital for human understand but not for the programs execution, such as removing formatting, scramble variable names etc. Even though these transformations also increase the potency, they increase the programmers effort to create an automatic deobfuscator. Adding useless, or bogus, information also increases the potency and thus indirectly the resilience. Adding bogus information does not change the execution behavior but can be reverted with various level of difficulty dependent on what and how the bogus information has been added.

Collberg et al [2] classifies the deobfuscator effort into *polynomial time* and *exponential time*, as shown in Figure 5.2b. The programmers effort, shown on the left axis, is measured as the scope of the obfuscation. It is easier to create a deobfuscation tool when the obfuscation only has been applied to a smaller part of the program, rather than the whole program. As shown in Figure 5.2b the scope is divided into *Local*, *Global*, *Interprocedural* and *Interprocess*. The obfuscation technique is *local* if it only affects a single basic block in a control flow graph, and it is *global* if the obfuscation affects the whole control flow graph. The obfuscation is *interprocedural* if it affects the information flow between procedures and it is *interprocess* if the obfuscation affects the interaction between processes executing independently. Figure 5.2b

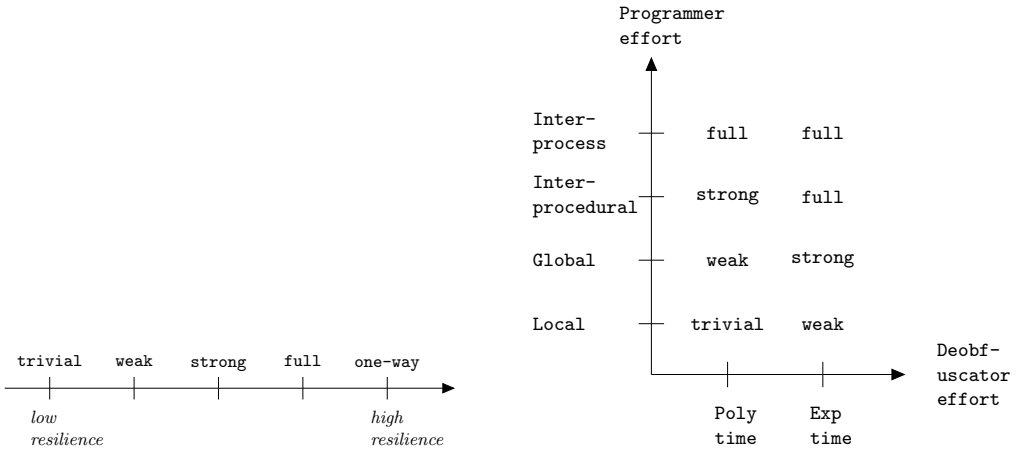


Figure 5.2: The resilience of an obfuscation technique
Original Figure in [2]

The formal definition of resilience defined by Collberg et al [2] is given below.

Definition 3. *(Transformation Resilience)*
Let τ be a behavior-conserving transformation, such that $P \xrightarrow{\tau} P'$ transforms a source program P into a target program P'. $\tau_{res}(P)$ is the *resilience* of τ with respect to a program P.
$\tau_{res}(P) =$*one-way* if information is removed from P such that P cannot be reconstructed from P'. Otherwise

$$\tau_{res}(P) \stackrel{\text{def}}{=} \texttt{Resilience}(\tau_{\texttt{Deobfuscator effort}}, \tau_{\texttt{Programmer effort}}),$$

where `Resilience` is the function defined in the matrix in Figure 5.2b.

□

5.2.5 Resilience Evaluation

As shown in Figure 5.2a the level of resilience goes from *trivial* to *one-way*. These levels are then mapped to the programmer effort and deobfuscator effort in Figure 5.2b. Since the RDEs is executed in polynomial time, the left side of Figure 5.2b is the correct scale.

Changing the control flow of P into the flow shown in Figure 5.1 not only increases the potency but also the resilience. The resilience level of the control flow flattening obfuscation technique is considered to be *strong*, as shown in Figure 5.2a, since it is infeasible to reconstruct the flow of program P, given the program P'. The resilience level of the control flow flattening obfuscation technique is highly dependent on how the implementation is made. Listing 2.1 and Listing 2.2 shows an example where the resilience level is weak since an deobfuscation tool only has to evaluate the predicates in

order to restore the old control flow.

The resilience is not only determined by the control flow flattening obfuscation technique. Splitting the lookup tables to no longer be stored in multidimensional arrays, as described in Section 4.3.4, increased the resilience level as well. The resilience level of this obfuscation technique is evaluated to *strong*, since it would be hard for an automatic tool to reconstruct the splitted values into an array again. Adding bogus operations also is considered having a *strong* resilience level and since the majority of operations are bogus operation which behaves in the same way as real operations they are hard to distinguish form real once. Altering the name of the identifiers is the only obfuscation technique that is considered to have a *one-way* resilience level. It is impossible to retrieve the old identifier names once they have been altered.

Even though most of the obfuscation techniques increases the resilience there are some that does not notably affect the total resilience level. Adding the second data obfuscation technique, making the operations execute in random order, does not affect the execution of an automatic tool at all and its resilience is hence evaluated as *trivial*. The technique is however vital since it increases the potency - i.e. the human understanding. Another technique that has low resilience as well is adding the memory swap. Even though this obfuscation technique adds a little resilience it is not notably and hence considered to have *weak* resilience.

The resilience is usually considered at each obfuscation technique alone but since our final implementation is considered as a whole the total resilience must be evaluated accordingly. Some techniques have *weak* or *trivial* resilience but since the majority of the techniques is considered to have *strong* or higher resilience the total resilience level is very strong.

5.2.6 Cost Definition

The third and last measurement is the execution cost, i.e; the time and space penalty. The cost measurement is divided into four classes; *free*, *cheap*, *costly* and *dear*. While some trivial operations such as lexical transformations are classified as free, other techniques such as control flow obfuscation are likely to cause an increase of execution cost. Since resources are bounded the cost of obfuscation will also have to be taken into consideration.

The formal definition of Cost, defined in [2], is given below:

Definition 4. *(Transformation Cost)*
Let τ be a behavior-conserving transformation, such that $P \xrightarrow{\tau} P'$ transforms a source program P into a target program P'. $\tau_{cost}(P)$, is the extra execution time/space of P' compared to P.

$$\tau_{cost}(P) \stackrel{\text{def}}{=} \begin{cases} \textit{dear} & \text{if executing } P' \text{ requires } \textit{exponentially} \text{ more resources than } P. \\ \textit{costly} & \text{if executing } P' \text{ requires } \mathcal{O}(n^p), p > 1, \text{ more resources than } P. \\ \textit{cheap} & \text{if executing } P' \text{ requires } \mathcal{O}(n) \text{ more resources than } P. \\ \textit{free} & \text{if executing } P' \text{ requires } \mathcal{O}(1) \text{ more resources than } P. \end{cases}$$

□

5.2.7 Cost Evaluation

The cost of the implemented obfuscation techniques control flow flattening, data obfuscation, bogus operations and memory shuffling are all considered *cheap* since none of these techniques increases the time complexity exponentially. The cost of the layout obfuscation on the other hand is considered to be *free* since it is not affecting the execution time in any way.

Our obfuscated implementation P' increased both in memory consumption and program size compared to the unobfuscated version P. As seen in Table 5.3 the total program size of P' is over two times larger than of P while the memory consumption remained roughly the same. The memory consumption was measured by using the built-in Java methods RUNTIME.GETRUNTIME().FREEMEMORY() and RUNTIME.GETRUNTIME().TOTALMEMORY() while the program size was checked using file properties in the operating system.

	P	P'	Difference
Total Program Size	1459 kB	3345 kB	229 %
Memory Consumption	4078 kB	4577 kB	112 %

Table 5.3: Implementation size and memory consumption of P and P'

The execution time have also increased as a result of the obfuscation. The execution time is only measured over the time it takes to encrypt data. Booting the program as well as generating lookup-tables for the control flow flattened implementation has not been taken into consideration. The result of the execution times is shown in Table 5.4 below.

Input Length	Execution Time P	Execution Time P'
2^4	0.4 ms	317 ms
2^8	1 ms	572 ms
2^{12}	15 ms	5312 ms
2^{16}	236 ms	58616 ms

Table 5.4: Execution time for the implementation with and without control flow flattening obfuscation technique added.

5.2.8 Quality Definition

The total *quality* is a combination of the *potency, resilience* and *cost*. The formal definition, given by Collberg et al [2], is given below:

Definition 5. *(Transformation Quality)*
$\tau_{qual}(P)$, the quality of a transformation τ, is defined as the combination of the potency, resilience and cost of τ:

$$\tau_{qual}(P) = (\tau_{pot}(P), \tau_{res}(P), \tau_{cost}(P))$$

□

5.2.9 Quality Evaluation

The total obfuscation quality of P' is a combination of the potency, resilience and cost. As previously stated, the potency and resilience level of P' is evaluated to *high* and *strong* respectively. The total cost on the other hand is evaluated to *cheap*. The total quality of P' is therefore:

$$\tau_{qual}(P) = (high, strong, cheap)$$

This implies that even if P' has high potency and resilience the execution time is not that great.

5.3 Correctness

The correctness of our implementation have been tested by generating six different versions of our implementation. Each version uses a unique AES key and random seed. The test is performed by randomly generating 600 different inputs with lengths ranging between 0 and 255 bytes. The input/output pair is then compared against a black-box reference implementation. The result of the extensive testing was that all 600 test was successfully encrypted the same way as the black-box reference implementation.

5.4 Existing Obfuscation Tools

A comparison with existing Java obfuscators have also been made. An important thing to consider is that our implementation P' is specially made for obfuscating a white-box AES implementation while existing tools are general obfuscators, meaning that they obfuscate any Java code. This will naturally effect the outcome of the measurements below. The tools tested are Allatori Java Obfuscator [29] and Zelix KlassMaster [30]. Both are available free as trial versions that are limited in some way but the limitations have been taken into consideration and should not effect the comparison in any significant way. The evaluation of obfuscation of P' was made with respect to potency, resilience and cost. Therefore, a short comparison on each of those values is presented below.

The potency measurements, presented in Table 5.1, is used to evaluate the total potency of both Allatori and Zelix. The values of Allatori and Zelix are compared to the potency values of P' in Table 5.5 below. The values of P' are the same as previously presented in Table 5.2.

Measurement	P'	Allatori	Zelix
μ_1	1766070	561	648
μ_2	527799	12	26
μ_3	2	1	1
μ_4	62883	18	16
μ_5	64143	-	-
μ_6	1	1	1

Table 5.5: Potency comparison with existing Java obfuscators.

A major difference between our implementation P' and the tested obfuscators is that both Allatori and Zelix utilize a technique called Java byte-code inlining. For instance they use byte-code to operate directly on the stack for method parameters and return values. For this reason the measurement μ_5 has not been measured as it is difficult to determine whether a referenced data structure is global or not. Attempts to measure time and space complexity of both Allatori and Zelix was also made but was unsuccessful because neither of the tools output program was runnable. Finally, a tool called ProGuard [31] was also tested but the only apparent transformation was name mangling of variables. As this does not affect potency, resilience nor cost it's not included in the comparison.

6

Discussion and Conclusion

The main aim of this master thesis is to investigate whether it is possible to obfuscate a program so that it becomes unprofitable for an adversary to try to reverse engineer it. As stated in the introduction, the obfuscated program should withstand known automatic reverse engineering tools as well as hardening the static and dynamic analysis of the program. Also, by making each instance differ in the implementation the adversary may be forced to manually perform the reverse engineering process which increases the amount of time and resources the adversary has to spend to successfully reverse engineer the program.

Our implementation P' is created in iterations with development alternating with reverse engineering. More details of the implementation can be found at Section 4.3. Based on the findings from each reverse engineering attempt the next obfuscation technique was chosen. The first iteration of P' consists of the control flow flattening obfuscation technique. Choosing control flow flattening as the primary obfuscation technique is decided based on the fact that it is the obfuscation technique that alters the program the most of all the techniques examined during the literature review, see Section 2. Also, control flow flattening drastically hardens the ability to statically analyze the program. Hence, the decision is to implement control flow flattening as the first obfuscation technique. As shown by the reverse engineering of the first iteration, described in Section 4.3.3, the control flow flattening obfuscation technique is not enough by its own to provide the desired obfuscation level. Two problems still exist; the descriptive identifier and the lookup tables being stored together in an array. Both of the problems are crucial to correct so iteration 2 and 3 that handle those problems, are equally important. After the third iteration the reverse engineering process is starting to become problematic. The previous reverse engineering processes have been performed by manually debugging the program during runtime. Now however, the program is too large and complex to follow manually with a debugger. During this iteration an automatic tool is developed

to help the reverse engineering process. The automatic tool is handwritten solely for this purpose and it is vital to realize that our adversary, as described in Section 4.3.1, is evaluated to have exceedingly high knowledge about the system and our implementation. However, the major challenge during the reverse engineering process is still to understand the underlying logic of the program. An adversary with less insight in our implementation and the obfuscation techniques being used would most likely have a greater challenge to understand the program than our highly advanced adversary. The two last iterations - adding bogus operations and memory swap - increases the level of reverse engineering effort as well. However, once again our adversary has an extreme advantage since he already has an automatic tool developed. It would have been interesting to see whether our adversary would have been as successful in reverse engineering the obfuscated program if he only was given the final implementation. During the development of the automatic reverse engineering tool some assumptions about how the real operations behaved were made, based on the reverse engineering analysis in iteration 3. Without these assumptions you can assume that an adversary would have a more difficult time to reverse engineer the final implementation.

Even though the reverse engineering process of our obfuscated implementation was successful one could still imagine usage scenarios where this, or a similar solution, would be usable. Depending on the adversary the implementation may offer different levels of protection. Even though our assumption throughout this thesis is that the execution environment is a white-box environment with an adversary present, the adversary needs to gain access in the first place. The environment may be "open" for attacks but the adversary still needs to gain access before any attack can be performed. By defining three different types of potential adversaries you can estimate how well the obfuscated implementation would withstand an attack. The first type of adversary is very powerful and may have an entire team of skilled hackers working together. Such an adversary may have found an exploit which enables remote code execution and have in that way gained white-box access to the system. In this scenario it is likely to assume that the adversary can also reverse engineer any obfuscated software on that host given a relatively short time period. The second type of adversary may not be able to produce its own exploits but can utilize already published ones. For such an adversary our obfuscated implementation may offer enough resistance to either stop or slow down the attack long enough for the target system to get patched. The third type of adversary is relatively unskilled but have gained access to the system by for instance hardware theft. Against this type of adversary the level of protection is assumed to be good enough to fully stop an attack. In all cases our obfuscated implementation results in an increased amount of work required for a successful attack. This may in turn encourage adversaries to consider other targets.

Apart from reverse engineering our implementation the obfuscation techniques was evaluated in terms of Potency, Resilience, Cost and Quality. Figure 4.2 and Figure 5.1, which shows the generated control flow graphs of P and P' respectively, are a visual representation of the change in potency. In the control graph of the source program,

Figure 4.2, the underlying logic of the program is easily identified. The control graph of P' however, shown in Figure 5.1, is nowhere close to be helpful when analyzing the underlying logic of the program, even though the Figure shows an simplified version. Even though the control flow graph of the obfuscated implementation is generated after the final iteration the graph would look pretty much the same if it was generated in one of the previous iterations. The layout obfuscation changes the identifier names and thus the node names of the flow graph, and adding bogus operations extends the flow graph with more nodes. Other than that the flow graph would look the same. This because the control flow flattening obfuscation technique changes the flow of the program so drastically. Control flow flattening increases the number of predicates in the implementation. As can be shown in Table 5.2 the μ_2 value, which represents the number of predicates, is increased around 264 000 times. The value is so high because each operation contains at least one predicate. As previously stated, the majority of the operations in P' are bogus operations and therefore the value is larger than if only real operations were present. The control flow flattening obfuscation technique also flattens the structure of the implementation, which is shown by the μ_3 value representing the nesting level. The nesting level is decreased and thus affects the potency negatively, but the increase of all the other values are far greater which make the decreased value negligible. Another value that is decreased is the μ_6 value, which shows the data structure complexity. The reason for this is that when the lookup tables are transformed from multidimensional arrays into separate methods they are no longer counted as data structures. However, the decrease of μ_6 is also negligible due to its small impact on the total potency. Also, one important fact that the potency measurements is missing is that flattening an array can actually increase the potency. A programmer that uses multidimensional arrays usually make the design choice in order to make the code more understandable. Flattening the array, i.e. removing the data structure complexity, probably makes the array less understandable even though the potency measure suggests the opposite. So even though the mu_3 and μ_6 measurements is evaluated as a decrease in potency, the understanding of the program is not affected much. Splitting the lookup tables into methods affects both the μ_2 value and the mu_1 value with such extent that it is canceling out the decrease of the μ_6 values. Adding bogus operations increases the total program length as well. The μ_4 and μ_5 values are both increased greatly as many of the data structures, such as the intermediate results and variables for calculating hash values, are stored globally. As every reference to a global or non inter-basic block variable is counted towards these measurements they are of nearly the same. The difference is that μ_5 also counts method parameters and is therefore slightly higher.

The increase and decrease of the values in Table 5.2 are as expected. However, some of the values are greater than expected. The length of the program for example, it is an increase by approximately 415 600 %. The number of predicates also is a huge increase by approximately 26 390 000%. When seeing the control flow graph of P' in Figure 5.1 these values are not surprising but still far greater than anticipated.

Table 5.5 shows a comparison of P' with some existing obfuscation tools available. The table presents the measurements of potency, presented in Table 5.1. As the values show, P' has greater values on all of the measurements compared to the two tools Allatori and Zelix. This is not surprising since P' is constructed to increase the total quality of the obfuscation, while the other tools are used for obfuscating general program code. It is relatively difficult to write an obfuscator which can operate on all kinds of programs, while writing a specific purpose program is easier. This explains the huge difference in potency.

The resilience level of P' is evaluated to very *strong* since several of the obfuscation techniques each are evaluated to *strong*, and P' is a combination of those techniques. As previously stated, our adversary might have had problems with implementing an automatic tool to reverse engineer the program if he only was given the final implementation of P'. Because even if both the control flow flattening technique added in the first iteration and adding the bogus operation which was implemented in the fifth iteration is separate evaluated to have *strong* resilience level, the combination of those two techniques are even stronger. Although the scale of resilience, as shown in Figure 5.2a, only has the values *strong* and *one-way*, we believe that the level of strong can be increased by the combination techniques with strong resilience level. Hence, the total resilience level of the implementation is very strong. The techniques evaluated to have *trivial* and *weak* resilience levels are not affecting the total resilience level negatively but rather not affecting the resilience at all.

Increasing the potency and resilience of P' has it drawbacks though. The increase of cost with respect to the program size is more than doubled in P' compared to P. The memory consumption used during execution also increased slightly. As can be seen in Table 5.4 the execution time has a correlation to the input length, which is not that surprising. Also, the execution time is far greater in P' than in P which also is not very surprising. However, the difference in execution time is decreasing as the input length increases. We suspect that this have to do with the fact that Java's just in time compiler is lazy, i.e; it only compiles code once it reaches it. As the input length increases each method will be executed more times. As the methods are already compiled in all but the first time this will decrease the average execution time. The increased cost may seem like a huge disadvantage if using the program to secure execution on hardware in a white-box environment, for example on a smart phone. However, we do not believe that the running time is the limiting factor when it comes to securing sensitive data in software. Waiting a few more seconds for a more secure authentication procedure is probably acceptable.

Due to its byte-code format and extensive debugging utilities Java is relatively easy to reverse engineer compared to many other languages. Since all the obfuscation techniques discussed in this thesis can be applied to most programming languages a better level of protection would probably have been achieved with another language. A programming language that produces assembly code, such as C / C++, would be significantly more dif-

ficult to reverse engineer and we believe that we would not have been able to successfully reverse engineer our own implementation if it was written in such a language.

6.1 Future Work

Promising areas for future work includes investigating Java byte-code inlining. Instead of distributing the program as byte-code compiled automatically by a Java compiler one can inline Java byte-code in the source code before compiling it. If correctly done automatic decompilers, such as JAD [27], will not be able to translate the inlined byte-code back to Java source code. This results in more complicated code for an adversary and will likely make the to reverse engineering process significantly more difficult. Alternatively other languages that compile into machine code, such as C / C++, could be investigated. Most of the presented obfuscation techniques should work well with other languages that are syntactically similar to Java. As decompiling and analyzing machine code, compared to Java byte-code, is significantly more difficult it would increase the work required for an adversary. Other types of languages, such as functional languages, may also be interesting to investigate as they could possibly offer other obfuscation techniques. Programs made in less known programming languages will also likely cause an increased cost for an adversary as finding people with sufficient knowledge would be harder.

In order to make the result more realistic it would be a good approach to give the complete solution P' to an outside adversary that does not have the same level of knowledge regarding the underlying logic of the program. This was not done in this Master Thesis due to time constraints, but it is an area for future work.

Another interesting area to investigate is to store parts of the program on a secure location. For the program to operate correctly it would need to request information from a remote computer. This information should differ each time so that data from previous runs have no value to future ones. While this requires a working connection of some sort it would open up for possibilities to analyze requests as they arrive. An attackers request pattern would likely differ from the one of a regular user and by running a risk analysis preventive actions could be taken.

Bibliography

[1] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan, K. Yang, On the (Im)possibility of Obfuscating Programs, in: Advances in Cryptology—CRYPTO 2001, Springer, 2001, pp. 1–18.

[2] C. Collberg, C. Thomborson, D. Low, A Taxonomy of Obfuscating Transformations, Tech. rep., Department of Computer Science, The University of Auckland, New Zealand (1997).

[3] C. Wang, J. Davidson, J. Hill, J. Knight, Protection of Software-based Survivability Mechanisms, in: International Conference on Dependable Systems and Networks, 2001. DSN 2001., IEEE, 2001, pp. 193–202.

[4] T. László, Á. Kiss, Obfuscating C++ Programs via Control Flow Flattening, Annales Universitatis Scientarum Budapestinensis de Rolando Eötvös Nominatae, Sectio Computatorica 30 (2009) 3–19.

[5] J. Cappaert, B. Preneel, A General Model for Hiding Control Flow, in: Proceedings of the tenth annual ACM workshop on Digital rights management, ACM, 2010, pp. 35–42.

[6] B. Anckaert, B. De Sutter, K. De Bosschere, Software Piracy Prevention Through Diversity, in: Proceedings of the 4th ACM workshop on Digital rights management, ACM, 2004, pp. 63–71.

[7] S. Schrittwieser, S. Katzenbeisser, Code Obfuscation against Static and Dynamic Reverse Engineering, in: Information Hiding, Springer, 2011, pp. 270–284.

[8] S. Chow, P. Eisen, H. Johnson, P. C. Van Oorschot, A White-Box DES Implementation for DRM Applications, in: Digital Rights Management, Springer, 2003, pp. 1–15.

[9] S. Chow, P. Eisen, H. Johnson, P. C. Van Oorschot, White-Box Cryptography and an AES Implementation, in: Selected Areas in Cryptography, Springer, 2003, pp. 250–270.

[10] M. Jacob, D. Boneh, E. Felten, Attacking an obfuscated cipher by injecting faults, in: Digital Rights Management, Springer, 2003, pp. 16–31.

[11] H. E. Link, W. D. Neumann, Clarifying Obfuscation: Improving the Security of White-Box DES, in: International Conference on Information Technology: Coding and Computing, 2005. ITCC 2005., Vol. 1, IEEE, 2005, pp. 679–684.

[12] B. Wyseur, B. Preneel, Condensed White-Box Implementations, in: Proceedings of the 26th Symposium on Information Theory in the Benelux, 2005, pp. 296–301.

[13] B. Wyseur, W. Michiels, P. Gorissen, B. Preneel, Cryptanalysis of White-Box DES Implementations with Arbitrary External Encodings, in: Selected Areas in Cryptography, Springer, 2007, pp. 264–277.

[14] L. Goubin, J.-M. Masereel, M. Quisquater, Cryptanalysis of White Box DES Implementations, in: Selected Areas in Cryptography, Springer, 2007, pp. 278–295.

[15] O. Billet, H. Gilbert, C. Ech-Chatbi, Cryptanalysis of a White Box AES Implementation, in: Selected Areas in Cryptography, Springer, 2005, pp. 227–240.

[16] J. Bringer, H. Chabanne, E. Dottax, White Box Cryptography: Another Attempt, IACR Cryptology ePrint Archive 2006 (2006) 468.

[17] Y. Xiao, X. Lai, A Secure Implementation of White-Box AES, in: Computer Science and its Applications, 2009. CSA'09. 2nd International Conference on, IEEE, 2009, pp. 1–6.

[18] Y. De Mulder, B. Wyseur, B. Preneel, Cryptanalysis of a Perturbated White-Box AES Implementation, in: Progress in Cryptology-INDOCRYPT 2010, Springer, 2010, pp. 292–310.

[19] Y. De Mulder, P. Roelse, B. Preneel, Cryptanalysis of the Xiao–Lai White-Box AES Implementation, in: Selected Areas in Cryptography, Springer, 2013, pp. 34–49.

[20] FIPS, 197: Advanced encryption standard (AES), National Institute of Standards and Technology.

[21] J. Daemen, V. Rijmen, AES proposal: Rijndael.

[22] C. E. Shannon, Communication Theory of Secrecy Systems*, Bell system technical journal 28 (4) (1949) 656–715.

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

[24] J. A. Muir, A Tutorial on White-box AES, in: Advances in Network Analysis and its Applications, Springer, 2013, pp. 209–229.

[25] H. Yamauchi, A. Monden, M. Nakamura, H. Tamada, Y. Kanzaki, K.-i. Matsumoto, A Goal-Oriented Approach to Software Obfuscation, IJCSNS 8 (9) (2008) 59.

[26] Eclipse CFG Generator, `http://eclipsefcg.sourceforge.net/`, accessed 2014-12-08.

[27] P. Kouznetsov, JAD, JAva Decompiler, `http://varaneckas.com/jad/`, accessed 2014-08-10.

[28] Eclipse, `https://www.eclipse.org/`, accessed 2014-08-10.

[29] Smardec, Allatori Java Obfuscator, `http://www.allatori.com/`, accessed 2014-08-10.

[30] Zelix Pty Ltd, Zelix KlassMaster, `http://www.zelix.com/`, accessed 2014-08-10.

[31] E. Lafortune, ProGuard, `http://proguard.sourceforge.net/`, accessed 2014-08-10.