White-box cryptography for digital content protection

Plasmans, Marjanne

Award date:
2005

TECHNISCHE UNIVERSITEIT EINDHOVEN

Department of Mathematics and Computer Science

MASTER’S THESIS

White-Box Cryptography for Digital Content Protection

by

Marjanne Plasmans

Supervisors: dr. ir. Bart van Rijnsoever, dr. ir. Peter Roelse,

prof. dr. ir. Henk van Tilborg, dr. Benne de Weger

Eindhoven, May 2005

Voor papa

Preface

This document is my Master's thesis and is the result of the final project to obtain the degree Master of Science in Applied Mathematics at the Eindhoven University of Technology (TU/e). This project has been done at Philips Software Eindhoven in the Digital Rights Management (DRM) group.

I am grateful to Philips Software for giving me the opportunity to perform my project in an inspiring work environment. I would like to thank my supervisors Bart van Rijnsoever, Peter Roelse, and Benne de Weger for their guidance, support, and understanding throughout this project.

Finally, I would like to thank my friends and my family for their love and support. Most of all I would like to thank my father, for everything.

Marjanne Plasmans
May 2005

Abstract

The aim of this Master's Thesis is to analyze how digital content for mobile phones can be protected in an effective way in the context of OMA-DRM. The Open Mobile Alliance (OMA) provides specifications for content distribution for mobile phones [17].

We will analyze the problem in the white-box attack context where the attacker has total visibility into software implementation and execution. First, we look at different techniques which are available for software protection, and we analyze how they can be applied for OMA-DRM. From these techniques we choose to focus on a relatively new technique: white-box cryptography. We show how this technique can be applied to AES keys, by hiding the AES key in lookup tables. This prevents an attacker from finding secret keys in the implementation. The result is a functionally equivalent program in which the key is no longer visible. However, white-box cryptography increases the amount of storage space for the white-box tables, and it causes a performance slowdown.

We will discuss an application of white-box cryptography in which we split the set of white-box tables into a dynamic part and a static part. Each client has a unique set of static tables which can only be used in combination with a unique set of dynamic tables which are transmitted to him. The result is that whenever a key needs to be updated, no longer the whole set of tables needs to be updated.

We will also analyze a new attack on a white-box AES implementation and we will look for possibilities to circumvent the attack. The attack can be carried out if an attacker has access to the complete set of white-box tables. In the application of white-box cryptography that we discuss, the set of tables is not transmitted totally. Therefore, an attacker cannot perform the attack by just eavesdropping on the communication line between the server and the client.

Finally, we will look for possibilities to apply white-box cryptography in the OMA-DRM context. Because of the total size of the tables and the slowdown we recommend using white-box cryptography only for keys which are fixed over a longer period of time. For example, white-box cryptography can be used to update the private key. White-box cryptography can also be used to store keys on the client's device.

Contents

List of Abbreviations

- **AES:** Advanced Encryption Standard
- **CEK:** Content Encryption Key
- **D:** Domain
- **DRM:** Digital Rights Management
- **KDF:** Key Derivation Function
- **KEK:** Key Encryption Key
- **MAC:** Message Authentication Code
- **OMA:** Open Mobile Alliance
- **REK:** Rights object Encryption Key
- **RO:** Rights Object
- **ROAP:** Rights Object Acquisition Protocol

Chapter 1

Introduction

Almost everybody has a mobile phone nowadays. Mobile phones keep getting smaller with improved functionality. For years it has been possible to buy the latest ringtones for our mobile phones. In the same way we can buy mp3 files and screensavers. The techniques keep getting better and better, and it may be perfectly common to watch a soccer match on our mobile phone in the future. As a result, it becomes more and more important to distribute digital content in a controlled manner. This is made possible by Digital Rights Management (DRM) systems.

DRM allows an owner of electronic content to control the content and restrict the usage of the content in various ways. This content can be games, photos, music, videos, ringtones etc.. The provider of an mp3-file could for example allow an end user to listen to an mp3-file three times before he decides to buy it. We focus on content for mobile phones. The Open Mobile Alliance (OMA) provides specifications for content distribution for mobile phones (see [17]).

We consider three threat models:

- *Black-Box Model:* In the traditional black-box model, the attacker is restricted to observe input and output of the algorithm, without any side-channels of information. A secret key of a cryptographic algorithm is hidden in the black-box and is never exposed. The security depends on the strength of the cryptographic algorithm.
- *Grey-Box Model:* Another model is the grey-box model where an attacker is also able to monitor side effects of the program execution. For example, an attacker can monitor the execution time, power consumption, and electromagnetic radiation.
- *White-Box Model:* In the white-box model, the attacker also has total visibility into software implementation and execution. To prevent an attacker from finding the key, the key needs to be hidden in the implementation.

When someone downloads digital content on a mobile phone, there are two main threats assuming that the server can be trusted (Figure 1.1): an attacker could eavesdrop messages which are transmitted between the server and the end user, or the attacker could be the end user who tries to derive information which is stored on his device. The attacker could for example be interested in distributing content or secret keys to other people, or he could try to change the licenses. As a result, it is important to send and store content securely. Because some people always try to get content for free, we assume that the end users cannot be trusted. Therefore, we will use a white-box attack model.

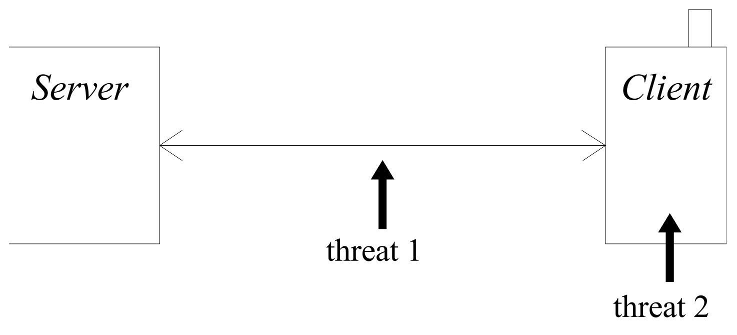


Figure 1.1: Threats

The question is how digital content for mobile phones can be protected effectively against threats. We want to protect content decryption by protecting the software and the decryption keys which are used. Many different techniques are available for software protection:

1. *Software Obfuscation Through Code Transformations:* For protection against reverse engineering. This prevents disclosure of sensitive information.

2. *Software Tamper Resistance:* For protection against program integrity threats. This can be used for preventing attackers for modifying licences which give access to content.

3. *Software Diversity:* For protection against automated attack scripts and widespread malicious software (e.g. viruses).

4. *White-box Cryptography:* For protecting secret keys in untrusted host environments. This prevents disclosure of secret keys. Actually white-box cryptography is a special class of software obfuscation.

5. *Software Watermarking/Fingerprinting:* For detecting and tracing violators who redistribute software illegally.

6. *Node Locking:* For locking software on hardware. The software can only be used on this particular hardware. This can be used for copy protection.

We will explore the first four techniques, and from these techniques we choose to focus on a relatively new technique: white-box cryptography.

In chapter 2 the key management of OMA-DRM is described. In chapter 3 software obfuscation, software tamper resistance, and software diversity techniques are outlined in general. White-box cryptography is described more extensively in chapter 4, followed in chapter 5 by a description of an attack on a white-box AES implementation. In chapter 6 the practical use of white-box cryptography is discussed. Finally, the conclusion is given in chapter 7.

Chapter 2

OMA-DRM

2.1 Introduction

OMA-DRM delivers specifications for developing applications and services that are deployed over wireless communication networks. It enables the distribution and consumption of digital content in a controlled manner. OMA-DRM addresses the various technical aspect of this system by providing appropriate specifications for the content formats, protocols, and a rights expression language. Presently, there are two versions of the OMA-DRM standard. Version two is a newer, cryptographically stronger version. In this section the key management of the second version of OMA-DRM is described as in [17].

2.2 Rights Objects

A content owner can provide appropriate permissions to a client for his content. These permissions are sent by the Rights Issuer to the client and are stored on the cient's device in a Rights Object which is managed by the DRM Agent (Figure 2.1). The DRM Agent is an entity residing on the client's device. The Rights Object is a file which describes permissions and constraints granted to the DRM Agent when accessing content. For example, a Rights Object can give permission to use specific content five times. The protocols between a Rights Issuer and a DRM Agent in a device are defined in the Rights Object Acquisition Protocol (ROAP).

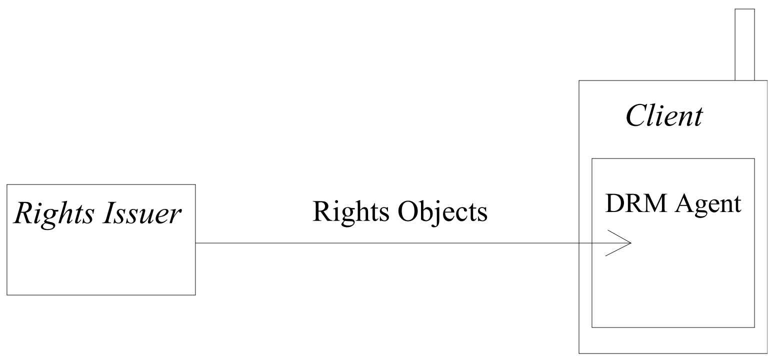


Figure 2.1: Transmitting Rights Objects

2.3 Content Encryption

DRM content is encrypted with a Content Encryption Key, K_{CEK}. K_{CEK} is a randomly generated 128-bit symmetric key. A piece of DRM content is encrypted once by the Rights Issuer, which makes the encryption the same for each client. Therefore, each client who buys content needs the same key for decryption. If someone buys content, the content and K_{CEK} need to be transmitted to the client. It is important that K_{CEK} is transmitted and stored securely, because if other people know the key, they will be able to decrypt content without paying for it. Therefore, K_{CEK} is sent in encrypted form to the client. This is done by using the encryption algorithm AES_WRAP. For details on AES_WRAP see [15]. The AES_WRAP scheme uses AES for encryption. AES is a block cipher which takes as input to the encryption and decryption algorithms a single 128-bit block. See [11] for more details on AES. AES_WRAP takes as input a key-encryption key KEK and key material K. K serves as plaintext and is encrypted with KEK:

$$\text{AES\_WRAP}(KEK, K)$$

K_{CEK} is encrypted with the Rights Object Encryption Key (K_{REK}) and it is stored in the Rights Object:

$$C = \text{AES\_WRAP}(K_{REK}, K_{CEK})$$

After receiving C, the DRM Agent decrypts C using K_{REK}:

$$K_{CEK} = \text{AES\_UNWRAP}(K_{REK}, C)$$

Additional software checks the Rights Object to see if it is valid and not expired. If this is the case, the client is allowed to access the content by decrypting it with K_{CEK}.

2.4 Protecting a Rights Object for a Device

The Rights Issuer randomly generates two keys of 128 bits: K_{REK} (Rights Objects Encryption Key) and K_{MAC} (Message Authentication Code key). K_{REK} and K_{MAC} are different for each client. K_{REK} is the wrapping key for the content encryption key K_{CEK} in Rights Objects. K_{MAC} is used for authentication of the message (Rights Object) carrying K_{REK}.

The authentication of the message goes as follows: The Rights Issuer generates a Rights Object. This Rights Object is hashed with K_{MAC} with the MAC-algorithm. The output of the MAC-algorithm is the MAC-value. The Rights Object, the MAC-value, and K_{MAC} are sent the the client. The client computes his own MAC-value over the Rights Object and compares this one with the received one.

For example, the Rights Object can allow content access for five times. If an attacker changes this value to fifty times, the calculated MAC-value over the Rights Object will no longer correspond to the sent MAC-value. As a result, no access to the content is granted.

Is it possible to restore old Rights Objects? If a client for example receives a Rights Object which allows him to access content 10 times, he could copy the Rights Object to another device, and restore this Rights Object when has used the 10 times. The MAC value is still correct. To prevent this from happening a MAC-value is calculated over all Rights Objects together. If a Rights Object would be replaced with another one, the MAC-values are no longer the same.

K_{REK} and K_{MAC} need to be transmitted securely to a recipient device. This is done by using AES_WRAP and RSA.

For each encryption operation, an independent random value Z is chosen. For the AES_WRAP scheme, K_{REK} and K_{MAC} are concatenated to form the key material K:

$$\begin{gathered} C_1 = \text{RSA.ENCRYPT}(PubKey_{Device}, Z) \\ KEK = \text{KDF}(Z, Null, kekLen) \\ C_2 = \text{AES\_WRAP}(KEK, K_{MAC}|K_{REK}) \\ C' = C_1|C_2 \end{gathered}$$

Where KDF is a key derivation function (see [20]). It is a simple key derivation function based on a hash function. $KekLen$ is set to the desired length of KEK, i.e. 16 bytes, and $Null$ is the empty string.

The Rights Issuer sends C' to the recipient device. After receiving C', the device splits it into C_1 and C_2 and decrypts C_1 using its private key, yielding Z:

$$Z = \text{RSA.DECRYPT}(PrivKey_{Device}, C_1)$$

Because Z is known, the device is able to derive KEK, and from KEK unwrap C_2 to yield K_{MAC} and K_{REK}:

$$K_{MAC}|K_{REK} = \text{AES\_UNWRAP}(KEK, C_2)$$

Now that K_{REK} is known, C can be unwrapped which yields K_{CEK}, en with K_{CEK} the content can be decrypted.

In Figure 2.2 the content decryption is summarized. C_1, C_2, C, and the encrypted content are sent to the client. With its private key the client is able to decrypt C_1 to obtain Z. Z serves as input to KDF which yields as output KEK. With KEK the client can decrypt C_2 to obtain K_{REK} and K_{MAC}. K_{REK} is the decryption key for C which is used for obtaining K_{CEK}. With K_{CEK} the content can be decrypted.

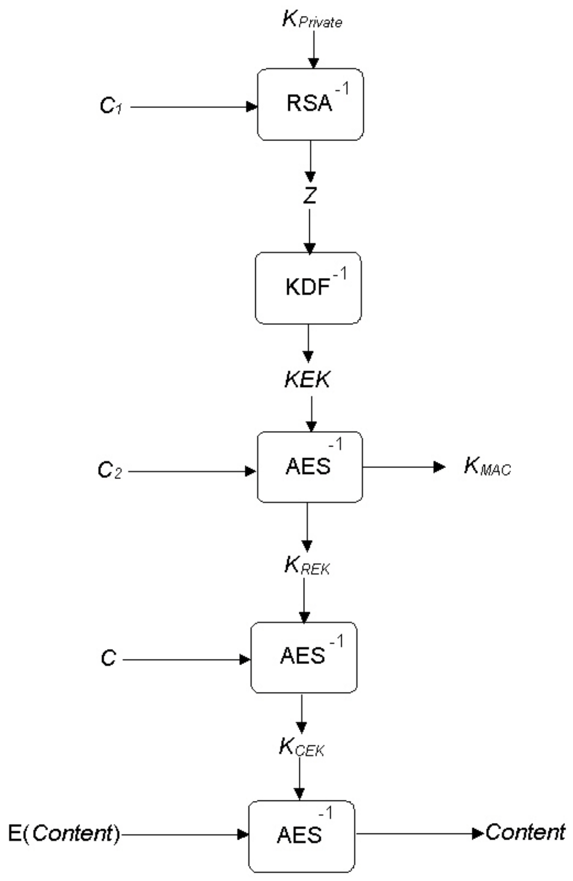


Figure 2.2: Content Decryption

2.5 Protecting a Rights Object for a Domain

If someone downloads an mp3-file, it would be nice to listen to it on different devices, for example on an mp3-player and on a computer. To make this possible Domains are introduced. A Domain is a set of Devices, which are able to share Rights Objects (the devices must be DRM compliant). Devices in a Domain share a Domain key. This section describes how to provide a device with a Domain key, K_D.

K_D is a 128-bit randomly generated AES key and is unique for each Domain D. K_D is the key-wrapping key used for protecting K_{REK} and K_{MAC} in a Rights Object issued to a Domain D. K_{MAC} is used for authentication of the message carrying K_D. K_D and K_{MAC} need to be distributed securely. The same procedure as in the previous section is used, the only difference being the replacement of K_{REK} with K_D:

$$\begin{aligned} C_1 &= \text{RSA.ENCRYPT}(PubKey_{Device}, Z) \\ KEK &= \text{KDF}(Z, NULL, kekLen) \\ C_2' &= \text{AES\_WRAP}(KEK, K_{MAC}|K_D) \\ C'' &= C_1|C_2' \end{aligned}$$

The Rights Issuer sends C'' to the recipient device. After receiving C'', the device splits it into C_1 and C_2' and decrypts C_1 using its private key, yielding Z:

$$Z = \text{RSA.DECRYPT}(PrivKey_{Device}, C_1')$$

Because Z is known now, the device is able to derive KEK, and from KEK unwrap C_2' to yield K_{MAC} and K_D:

$$K_{MAC}|K_D = \text{AES\_UNWRAP}(KEK, C_2')$$

Now we will describe how the content is protected with a Domain key. K_{REK} and K'_{MAC} are distributed under a Domain key K_D by using AES_WRAP. KEK in AES_WRAP is set to K_D and K is set to the concatenation of K'_{MAC} and K_{REK}:

$$C* = \text{AES\_WRAP}(K_D, K'_{MAC}|K_{REK})$$

After receiving $C*$, the device decrypts $C*$ using K_D:

$$K'_{MAC}|K_{REK} = \text{AES\_UNWRAP}(K_D, C*)$$

Now that K_{REK} is known, C can be unwrapped which yields K_{CEK}, en with K_{CEK} the content can be decrypted.

In Figure 2.3 the content decryption with a Domain key is summarized. $C_1, C'_2, C*, C$, and the encrypted content are sent to the client. With its private key the client is able to decrypt C_1 to obtain Z. Z serves as input to KDF which yields as output KEK. With KEK the client can decrypt C'_2 to obtain K_D and K_{MAC}. K_D is used for decrypting $C*$ which results in K_{REK} and K'_{MAC}. K_{REK} is the decryption key for C which is used for obtaining K_{CEK}. With K_{CEK} the content can be decrypted.

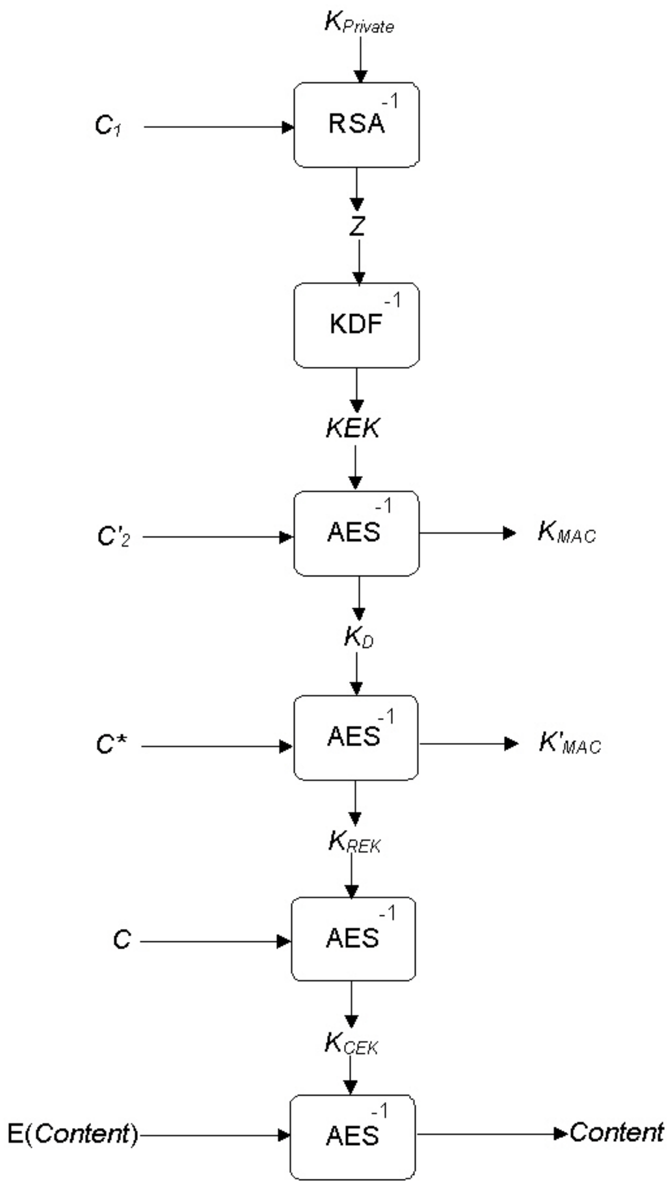


Figure 2.3: Content Decryption with a Domain key

2.6 Overview of the Keys

- K_{CEK} (Content Encryption Key): K_{CEK} is an AES-key which is generated by the Rights Issuer for content encryption. This key is unique for a particular piece of content, and it is generated once for each piece a content. This implies that for each piece of content, a different K_{CEK} needs to be stored.
- K_{REK} (Rights object Encryption Key): K_{REK} is an AES-key generated by the Rights Issuer to protect K_{CEK} in a Rights Object for a particular client. This key is chosen uniquely for each Rights Object, thus every time a client buys content, a new K_{REK} is generated. This key does not need to be stored.
- K_D (Domain key): K_D is an AES-key generated by the Rights Issuer to protect K_{REK} and K'_{MAC} for a particular Domain. This key is fixed over a long time, but the Rights Issuer can decide to update this key. The key is unique for a particular Domain.
- K_{MAC} (Message Authentication Code key): K_{MAC} is an AES-key generated by the Rights Issuer to authenticate the message carrying K_{REK} and K_D for a particular client. This key is changed for each Rights Object.
- KEK (Key Encryption Key): KEK can be calculated with the Key Derivation Function. It is an AES key for decrypting the message carrying K_D and K_{MAC}. This key is different for each Rights Object.
- $PrivKey_{Device}/PubKey_{Device}$: $PrivKey_{Device}$ and $PubKey_{Device}$ are RSA keys which are generated by a Certification Authority. $PrivKey_{Device}$ will be embedded on a device while manufacturing the device. This key is generated once, and cannot be changed. Each client had a unique $PrivKey_{Device}/PubKey_{Device}$ pair.

2.7 Threats

Some people always try to get content for free, and therefore we assume the end user cannot be trusted. As already mentioned in the introduction two main threats exist: an attacker could eavesdrop messages which are transmitted between the server and the client, or the attacker could be the end user who tries to derive information which is stored on the device. By eavesdropping an attacker cannot do anything (except a brute force attack), because all the messages are sent in encrypted form to the client (assuming that these messages are encrypted properly). All the information an attacker needs is available on the device, therefore the information on the device needs to be stored securely. For OMA-DRM we are interested in two classes of threats:

1. *Searching Keys:*
   An attacker can search for keys on the device and distribute them to other people who can use the keys to decrypt content. Therefore, it is important that the keys are protected well. The security of the system relies completely on the secrecy of the private key. Once someone is

able to extract the private key, he is able to derive K_D, K_{REK}, K_{CEK}, and to decrypt the content. Therefore, it is important that the private key is not accessible to anyone. If the client distributes his private key, everybody could decrypt each message that is sent to that person.

A technique for protecting keys is white-box cryptography. White-box cryptography can be used for protecting secret keys in untrusted host environments. In §6.6 will be discussed to which keys white-box cryptography can be applied effectively. Unfortunately these techniques cannot be used for protection of the private key, because the techniques do not apply to RSA.

2. *Changing Software:*
   Another threat is that an attacker could change the software that checks the Rights Objects. This software checks the rights in a Rights Object and decides whether or not content can be accessed and for how many times. If an attacker changes this software, he could grant himself unlimited access.

   In chapter 3, we outline the following software protection techniques: software obfuscation, software tamper resistance, and software diversity techniques.

Chapter 3

Software Protection Techniques

3.1 Introduction

There are several techniques to protect software (see [16]). In this chapter software tamper resistance (§3.2), software obfuscation (§3.3), and software diversity (§3.4) are outlined. In §3.5 a theoretic result on software obfuscation is given, and in §3.6 we describe where the different techniques can be used for protection in the OMA-DRM context.

3.2 Software Tamper Resistance

The main goal of making software tamper resistant is to protect software against program integrity threats. First, the contributions of Aucsmith to this area are described, followed by the self-checking technique of Horne et al.. After that, we describe the contributions of Chang and Atallah, and finally the contributions by Chen et al..

3.2.1 Aucsmith

Aucsmith et al. proposed a self-checking technique in which small, embedded segments of code verify the integrity of a program as the program is running. These embedded code segments (Integrity Verification Kernels, IVK's) check whether a running program has been altered. Aucsmith et al. proposed a set of four design principles for their self-checking technology in [1]:

- *Dispersion of secrets in space and time:* Secret components are evenly distributed throughout the workspace. This prevents the perpetrator from ”being lucky” and discovering the entire secret components in a single attempt. Another transformation would be temporal changes i.e. certain secrets are observed at certain times.

- *Obfuscation and interleaving:* Converting a structured program to a less readable state. The general complexity of the program is increased by interleaving tasks and rewriting commonly occurring code in more uncommon format.

- *Installation unique code:* Each code must have a unique component. This can protect against automated attacks.
- *Interlocking trust:* Each segment does not only depend on itself but also on other segments to effectively perform its tasks. Thus each segment of code would be made responsible for maintaining and verifying the integrity of other segments.

All code segments are encrypted except the one that is being executed while the program is running. After execution, the executed code segment becomes encrypted again while the next code segment decrypts. Encryption and decryption require a lot of time. Therefore, this method should only be applied to small portions of code segments. A drawback of this method, mentioned by Wang in [18], is that the encryption and decryption processes of IVK are carried out on a host that is not trusted. Although the key for encryption and decryption is only exposed shortly, it is possible that an adversary could obtain the key and subsequently compromise the entire execution. Furthermore, building the IVK requires considerable user intervention in the identification and isolation of the critical code segments that need to be specially armored.

3.2.2 Horne

Horne et al. presented in [14] a self-checking technique, in which a program while running in a potentially hostile environment, repeatedly checks itself to verify that it has not been modified. This prevents both misuse and repetitive experiments for reverse engineering or other malicious attacks. The self-checking technique consist of two kinds of components:

- *Testers:*
  Each tester computes a hash of a section of the code and compares the computed hash value to the correct value. An incorrect value triggers the response mechanism.

  The entire executable code section will be covered with overlapping intervals, each of which is tested by a single tester. The testers are randomly assigned to the intervals. The high overlap plus the random assignment provide a high degree of security for the testing mechanism: changing even a single bit requires disabling a large fraction of the testers to avoid detection.

- *Correctors:*
  Each interval has its own corrector, and is tested by exactly one tester. A corrector can be set to an arbitrary value, and is set in a way that the interval hashes to zero for the particular hash function used by the tester testing the interval.

  If copy-specific watermarks are used, an attacker might be able to locate the tester mechanism by obtaining two different marked copies of the code and comparing them. The differences do not only reveal the watermarks but also any changes needed in the self-checking mechanism to compensate for different watermarks. To prevent this from happening correctors are used. These correctors are separated from the testers and the response mechanism. If an attacker knows the values of the correctors, the attacker will not be able to disable the rest of the mechanism.

Horne at el. only present their techniques, but more experimental and theoretical research is needed on the coverage and robustness of the self-checking mechanism.

3.2.3 Chang and Atallah

Chang and Atallah proposed in [5] a method in which software is protected by a set of guards embedded within a program, each of which can do any computation . In addition to guards that compute checksums of code segments (like Horne et al. proposed), they proposed the use of guards that actually repair attacked code:

- *Checksum code:* Checksum a piece of program code at runtime and verify if it has been tampered with. If the guarded code is found altered, the guard will trigger a response mechanism.
- *Repair Code:* Restore a piece of damaged code to its original form before it is executed or used (as data). This can be done by overwriting tampered code with a clean stored copy. This repairing action effectively eliminates the changes done to the code by an attacker, and allows the program to run as if unmodified.

Guards can protect against diversity attacks, because guards can be grouped together in many different ways. As a result, a software developer can have different copies of its software applications protected differently . Successful attacks against one of the copies would not work for the others. Chang and Atallah claim that if configured properly, guards only cause slight impacts on the performance of guarded programs. In an example in [5] they showed that the impact on a program of around 300 KB is around 10 KB.

3.2.4 Chen

Most methods used in Software Tamper Resistance verify the shape of the code, and sometimes critical data, before or during the runtime. This is done by computing a cryptographic checksum on one or more segments of the code that is being protected. However, it is quite easy to detect the verification routine, because of the the atypical nature of the operation, since most applications do not read their own code segments. Spreading many smaller checks over time and space, repeating atypical operations all over the code are solutions for this problem (see [6]).

Chen et al. proposed a technique called Oblivious Hashing (OH). The idea is to hash the execution trace of a piece of code. The main goal is to blend the hashing code seamlessly with the host code (software), making them locally indistinguishable. This technique can protect against automatic attacks. Oblivious hashes introduce a number of unique issues:

- Pre-computation of correct hashes: An oblivious hash is ’active’ in the sense that the code to be protected must run (or be simulated) in order for the hash to be produced.
- Security coverage over code paths: An oblivious hash depends on the exact path through a program, as determined by input data. If execution does not reach some part of a program during hash computation, that part is not hashed and thus unprotected.

- Unhashable data: Data that are too variable or not predictable (e.g. time of the day, user identity, etc.) cannot be hashed obliviously, because they can cause oblivious hashes to vary arbitrarily. As a result, only a portion of the host code executes deterministically with respect to the input parameters. Therefore, 'unhashable' expressions need to be determined first, so they can be excluded from the hash computation. The authors experimented with some real-world programs and showed that around 80% of the code can be obliviously hashed for a majority of functions.

3.3 Software Obfuscation Through Code Transformations

The main goal of software obfuscating is to protect software against reverse engineering. By reverse engineering software, attackers can extract algorithms from an application and incorporate them into their own programs. This is difficult to detect and pursue legally. For example in the OMA-DRM context, secret key derivation functions are used which need to be protected against reverse engineering.

The idea behind software obfuscation is to transform a program P into another program P' which is functionally equivalent to the original program P, but more difficult to understand. However, given enough time, a good programmer will always be able to reverse engineer any application. But when this time is practically prohibitive, this may be acceptable [10].

Cohen discussed several obfuscation techniques in [9]. More recently Collberg et al. gave an overview of different obfuscation techniques and proposed the following classification of code transformations [10]:

- **Layout Obfuscation**: operates on the information in a program which is unnecessary for the execution of the program.
- **Data Obfuscation**: operates on the data structures used in a program.
- **Control Flow Obfuscation**: operates on the flow of execution in a program.
- **Preventive Transformation**: prevents decompilers and deobfuscators from operating correctly.

In this chapter a global overview is given of the techniques discussed by Cohen and Collberg et al.. A lot of the techniques are similar. The techniques are placed in one of the above four classes.

3.3.1 Evaluation

It is important to say something about the quality of the different obfuscation techniques. Collberg et al. classified the different techniques according to:

- **Resilience:** The resilience of a transformation measures how well the transformation protects against an automated attack (Figure 3.1). The resilience of a transformation $\mathcal{T}$ can be seen as a combination of two measures:

  - *Deobfuscator effort:* The execution time and space required for such an automated deobfuscator to be able to understand the obfuscated code.
  - *Programmer effort:* The amount of time required to construct an automated deobfuscator which is able to understand the obfuscated code.

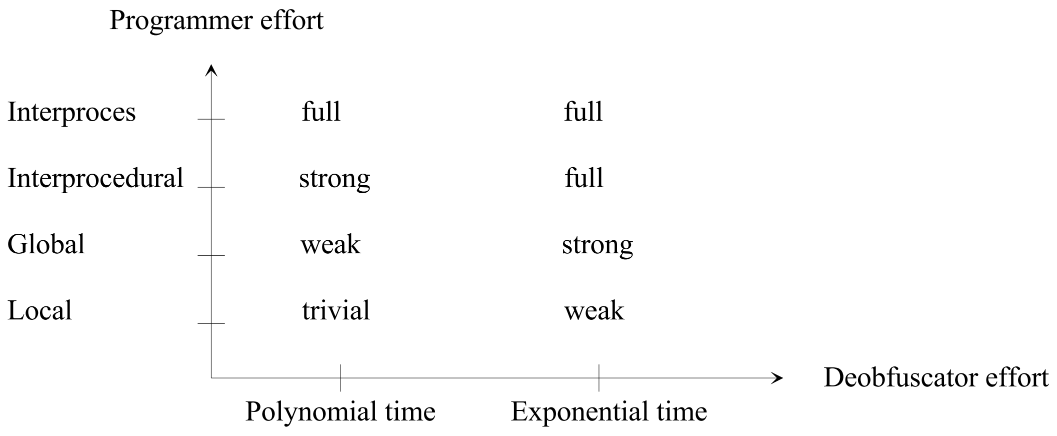


Figure 3.1: Obfuscation Resilience

Collberg et al. measure resilience on a scale from trivial to one-way (trivial, weak, strong, full, one-way). One-way transformations can never be undone. The effort of a deobfuscator can be classified as either polynomial time or exponential time. Programmer effort, the effort required to automate the deobfuscation of a transformation, is measured as a function of the *scope* of $\mathcal{T}$. This is based on the intuition that it is easier to construct counter-measures against an obfuscating transformation that only affects a small part of a procedure than against one that affects an entire program [10].

The scope of a transformation $\mathcal{T}$ is defined as:

- *Local:* if it affects a single basic block of a control flow graph.
- *Global:* if it affects an entire control flow graph.
- *Inter-procedural:* if it affects the flow of information between procedures.
- *Inter-process:* if it affects the interaction between independently executing threads of control.

- **Cost:** The ”cost” of a transformation is the execution time/space penalty which a transformation incurs on an obfuscated application. The cost is:

  - *Free:* if executing P' requires $\mathcal{O}(1)$ more resources than P.
  - *Cheap:* if executing P' requires $\mathcal{O}(n)$ more resources than P.
  - *Costly:* if executing P' requires $\mathcal{O}(n^p), p > 1$, more resources than P.
  - *Dear:* if executing P' requires exponentially more resources than P.

3.3.2 Layout Obfuscation

Layout transformations operate on information in a program which is unnecessary for the execution of the program. The layout can be obfuscated in the following ways:

- *Remove Comments (Collberg):* When available comments can be removed. This is a one-way transformation, because once the information is deleted it cannot be recovered. This technique is free of cost.

- *Scramble Identifiers Names (Collberg):* Scrambling identifier names is also a one-way transformation. Identifiers contain pragmatic information, and without it a program is more difficult to understand. This technique is also free of cost.

3.3.3 Data Obfuscation

In this section transformations will be described which obfuscate data structures. These transformations can be classified into three groups: *Storage and Encoding*, *Aggregation*, and *Ordering* of the data.

- **Storage and Encoding:** Storage transformations attempt to choose unnatural storage classes for data. For example converting a local variable into a global one. Similarly, data encoding transformations attempt to choose unnatural encodings for common data types.

  - *Split Variables (Collberg):* Boolean variables and other variables of restricted range can be split into two or more variables. The level of security depends on the number of variables in which the original variable is split.

- *Promote Scalars to Objects (Collberg):* Variables from a specialized storage class can be promoted to a more general class. For example, in Java, an integer variable can be promoted to an integer object. This technique has a strong resilience and it is free of cost.
- *Convert Static Data to Procedure (Collberg):* Useful pragmatic information can be extracted from static data. This can be made more difficult by converting a static string into a program that produces the string. This program could possibly produce other strings as well. The security depends on the complexity of the program.
- *Change Encoding (Collberg):* An example of an encoding transformation is replacing an integer variable i by $i' = c_1 \cdot i + c_2$. Here c_1 and c_2 are constants. This sort of encoding will add little execution time and the security depends on the complexity of the encoding function.
- *Change Variable Lifetimes (Collberg):* For example, a local variable can be changed into a global variable, which can then be shared between independent procedure invocations, which are not active at the same time. This technique has a strong resilience and it is free of cost.
- *Equivalent Instruction Sequences (Cohen):* Instruction sequences can be replaced with equivalent sequences. For example replacing $x + 17$ by $x + 20 - 3$.
- *Program Encodings (Cohen):* Any sequence of symbols in a program can be replaced by any other sequence of symbols, provided a method exists for undoing that replacement for the purpose of interpretation. For example encryption-decryption, compression-decompresssion.
- *Build and Execute (Cohen):* Instructions can be built prior to execution and then are executed. This is a so called self-modifying code.

• **Aggregation:** These obfuscations alter how data is grouped together.

- *Merge Scalar Variables (Collberg):* Two or more scalar variables can be merged together into one new variable. For example two 16-bit variables can be merged into one 32-bit variable. This technique has a weak resilience, but it is free of cost. An attacker only has to look at the set of arithmetic operations being applied to a particular variable in order to guess that it actually consists of two merged variables.
- *Modify Inheritance Relations (Collberg):* In Java, classes are essentially abstract data types that encapsulate data (instance variables) and control (methods). We write a class as $C = (V, M)$, where V is the set of C's instance variables and M its methods. Two classes C_1 and C_2 can be composed by aggregation (C_2 has an instance variable of type C_1) as well as by inheritance (C_2 extends C_1 by adding new methods and instance variables). The techniques are free of cost.

   - *Restructure Arrays (Collberg):* The following transformations can be used for obscuring operations performed on arrays. These techniques have a weak resilience.
     * Split an array into sub-arrays (free)
     * Merge two or more arrays into one array (free)
     * Fold an array (increasing the number of dimensions)(cheap)
     * Flatten an array (decreasing the number of dimensions)(free)

- **Ordering:**
   - *Randomize the order of declarations in the source application (Collberg):*
     * Reorder instance variables. This technique has a one-way resilience, and is free of cost.
     * Reorder methods. This technique has a one-way resilience, and is free of cost.
     * Reorder the elements in an array. This technique has a weak resilience and is free of cost.
   - *Variable Substitutions (Cohen):* Variables can be substituted to alter program appearance. If the variables are placed pseudo-randomly throughout the program, this can cause a great deal of diffusion.

3.3.4 Control Obfuscation

In this section transformations that attempt to obscure the control flow will be discussed. These transformations can be classified as affecting the *Aggregation*, *Ordering*, or *Computations* of the flow of control. We want the transformations to be not only as cheap as possible, but also resistant to attacks from deobfuscators. To achieve this, many transformations rely on the existence of so called opaque variables and opaque predicates.

A variable V is *opaque* at a point p in a program, if V has a property q at p which is known at obfuscation time, but difficult to deduce for a deobfuscator. A predicate P is *opaque* at point p in a program, if its outcome is known at obfuscation time.

Ideally, the goal is to construct opaque predicates that require worst case exponential time to break but only polynomial time to construct.

- **Aggregation:** Control aggregation obfuscations change the way in which program statements are grouped together. Control aggregation transformations break up computations that logically belong together or merge computations that do not.
   - *Inline Method (Collberg):* Replacing a procedure call with the statements from the called procedure itself. The resilience is one-way and it is free of cost.
   - *Outline Statements (Collberg):* Turning a sequence of statements into a subroutine. The resilience is strong and it is free of cost.

  - *Clone Methods (Collberg):* To understand the behavior of a subroutine, the body needs to be examined. However, it is also important to look at the different environments in which the routine is being called. To make this more difficult, different routines can be called, while in fact this is not the case. The security depends on the quality of the opaque predicate.
  - *Loop Transformations (Collberg):*
    - Loop blocking can be used to improve the cache behavior of a loop by breaking up the iteration space so that the inner loop fits in the cache. The resilience is weak and it is free of cost.
    - Loop unrolling replicates the body of a loop one or more times. If the loop bounds are known at compile time the loop can be unrolled in its entirety. The resilience is weak and it is cheap.
    - Loop fission turns a loop with a compound body into several loops with the same iteration space. The resilience is weak and it is free of cost.

    Applied in isolation, it is not really secure. With static analysis it is easy to reroll an unrolled loop. However, when the transformations are combined, it is much more difficult for a deobfuscator to restore the original form.
  - *Interleave Methods (Collberg):* The idea of interleaving different methods declared in the same class is to merge the bodies and parameter lists of the methods and add an extra parameter (or global variable) to discriminate between calls to the individual methods. Ideally the methods should be similar in nature to allow merging of common code and parameters. The security depends on the quality of the opaque predicate.
  - *Intermixing Programs (Cohen):* Instructions of two independent operations can be intermixed.

- **Ordering:** Control ordering transformations randomize the order in which computations are carried out.

  - *Control ordering transformations (Collberg):* The idea is to randomize the placement of any item in the source application when possible. The resilience is one-way and it is free of cost.
  - *Instruction Reordering (Cohen):* Many instruction sequences can be reordered without altering program execution.
  - *Adding and Removing Calls (Cohen):* Programs that use subroutine calls and other similar processes can be modified to replace the call and return sequences with in-line code or altered forms of call and return sequences.
  - *Adding and Removing Jumps (Cohen:)* Many program sequences can be modified by placing series of jump instructions where previous instruction sequences were located, relocating the previous instructions, and jumping back after sequences are completed.

- **Computation Transformations:** Computation obfuscations affect the control flow in a program. Computation transformations insert new code, or make algorithmic changes to the source application.
  - *Insert Dead or Irrelevant Code (Collberg):* Hide the real control flow behind irrelevant statements that do not contribute to the actual computations. For example code that will never be executed can be inserted (dead code). The more predicates a piece of code contains, the higher the perceived complexity.
  - *Reducible to Non-Reducible Flow Graphs (Collberg):* These obfuscations introduce code sequences at the object code level for which exist no corresponding high-level language constructs. For example, the Java bytecode has a `goto` statement, which means that the Java bytecode can express arbitrary control flow, whereas the Java language can only express structured control flow.
  - *Remove Library Calls and Programming Idioms (Collberg):* In programs written in Java library calls are common. The library calls have well known semantics and fixed names, which attackers can use to obtain information of a program. This problem can be solved by using own versions of the standard libraries. The cost is not in execution time, but in the size of the program.

    A similar problem occurs with cliches (or patterns), common programming idioms that occur frequently in a program. An attacker will look for these patterns to get some information about the program. This problem can be prevented by using techniques which identify common parts and replace them with less obvious ones. The resilience is strong.
  - *Extend Loop Condition (Collberg):* Here we want to obfuscate a loop by making the termination condition more complex. The basic idea is to extend the loop condition with an opaque predicate which will not affect the number of times the loop will execute. The security depends on the quality of the opaque predicates.
  - *Table Interpretation (Collberg):* The idea of table interpretation is to convert a section of code into a code for another virtual machine. This new code is then executed by a virtual machine interpreter included with the obfuscated application. A particular application can contain several interpreters, each accepting a different language and executing a different section of the obfuscated application. Table interpretation is costly because it makes a program very slow. Table interpretation can therefore best be used for small parts of code which need a high level of protection. However, the resilience is strong.
  - *Add Redundant Operands (Collberg):* Once we have constructed some opaque variables we can use algebraic laws to add redundant operands to arithmetic expressions. For example we can change $Y = X + 1$ into $Y = X + A/B$, where A/B is always 1. Now A and B can take on different values during the execution of the program as long as their quotient is 1.

- *Parallelize Code (Collberg):* The idea of parallelizing a program is to obscure the actual flow of control. This can be done by either introducing dummy processes which perform no useful tasks or by splitting sequential sections of the application code into multiple sections executing in parallel. This is costly because it causes a slowdown of the program. However, the resilience is high.

- *Garbage Insertion (Cohen):* Any sequence of instructions that are independent of the in-line sequence can be inserted into the sequence without altering the effective program execution.

- *Instruction Equivalence (Cohen):* Applicable operation codes can be replaced with equivalent operation codes.

- *Simulation (Cohen):* Any sequence of instructions can be replaced by an equivalent sequence for a different processor, and that processor can be simulated by an interpretation mechanism.

3.3.5 Preventive Transformations

The goal of preventive transformation is to make known automatic deobfuscation attacks more difficult, or to explore known problems in current deobfuscators or decompilers.

- **Inherent:**
  Explore inherent problems with known deobfuscation techniques.

- **Targeted:**
  Explore weaknesses in current decompilers and deobfuscators.

-*Anti-Debugger Mutations (Cohen):* They make a program resistant to a debugger.

3.3.6 Overlap

The techniques described by Cohen and Collberg et al. have different names, but show a lot of similarities. The following table shows the similarities between the techniques.

		Collberg	*Cohen*
Layout		Remove Comments	
		Scramble Identifiers Names	
Data	Storage & Encoding	Split Variables	
		Promote Scalars to Objects	
		Convert Static Data to Procedure	Build and Execute
		Change Encoding	Program Encoding
		Change Variable Lifetimes	
			Equivalent Instruction Sequences
	Aggregation	Merge Scalar Variables	
		Modify Inheritance Relations	
		Restructure Arrays	
	Ordering	Reorder instance variables	Variable Substitutions
		Reorder methods	
		Reorder arrays	
Control	Aggregation	Inline Method	
		Outline Statements	
		Clone Methods	
		Loop Transformations	
		Interleave Methods	Intermixing Programs
	Ordering	Control Ordering Transformations	Instruction Reordering
			Adding and Removing Calls
			Adding and Removing Jumps
	Computations	Insert Dead or Irrelevant Code	Garbage Insertion
		Reducible to Non-Reducible Flow Graphs	
		Remove Library Calls and Programming Idioms	Instruction Equivalence
		Extend Loop Condition	
		Table Interpretation	Simulation
		Add Redundant Operands	
		Parallelize Code	
Preventive	Inherent		
	Targeted		Anti-Debugger Mutations

3.4 Software Diversity

Genetic diversity is very important in biological systems. If there were no genetic diversity, then the outbreak of a virus could infect a lot of people at the same time, and the virus could be spread very fast. Computers, on the other side, are very homogeneous. This is advantageous in the sense that it is cheap to produce massive clones of one design, and each copy of a program runs identically on different machines. This makes maintenance tasks and distribution much easier. Once a computer can be infected with a virus, all computers with a similar configuration can be infected too. However, if each configuration issued to a user were different from all others, but functionally equivalent, then a virus which infects one configuration, would fail to infect another. A disadvantage of software diversity is that it makes the testing of software also much harder.

Forest et al. proposed the following guidelines for software diversity in [12]:

- Preserve high-level functionality. The input/output behavior of programs should be identical on different computers.
- Introduce diversity in places that will be most disruptive to known or anticipated intrusion methods.
- Minimize cost, both run-time performance cost and the cost of introducing and maintaining diversity.
- Introduce diversity through randomization.

This can be done by methods ranging from those that produce variability in the physical location of executed instructions, the order in which instructions are executed, and the location of instructions in memory run-time. See §3.3 for different techniques.

3.5 The (Im)possibility of Obfuscation?

3.5.1 Introduction

Is it possible to perfectly obfuscate a program which nobody can crack? Is it possible to claim in general whether or not an obfuscator is good? Barak et al. dealt with these questions in [3]. The main results will be described in this section. We want to know what these results imply for practical use.

3.5.2 Obfuscation

Informally, an obfuscator $\mathcal{O}$ is an (efficient, probabilistic) ”compiler” that takes as input a program (or circuit) P and produces a new program $\mathcal{O}(P)$ satisfying the following two conditions [2]:

- Functionality: An obfuscated program $\mathcal{O}(P)$ should compute the same function as the original program P.

- ”Virtual black box” property: Anything that can be efficiently computed from $\mathcal{O}(P)$ can be efficiently computed given oracle access to P.

  This means that nothing can be computed from the obfuscated program, that cannot be computed by only observing the input-output behavior of the program. For example let x be a plaintext which is encrypted with an encryption function E with key K: $E_K(x)$. If we look at the implementation of this program, we will see the key K. However, if we obfuscate this program we will get: $\mathcal{O}(E_K(x))$. The key K cannot be found in $\mathcal{O}(E_K(x))$.

3.5.3 Result

The main result of [3] is the following: There exists a family $\mathcal{F}$ of functions which is *inherently unobfuscatable* in the sense that there is some property $\pi : \mathcal{F} \to \{0, 1\}$ such that:

- Given *any* program (circuit) that computes a function $f \in \mathcal{F}$, the value $\pi(f)$ can be efficiently computed;

- Yet, given oracle access to a (randomly selected) function $f \in \mathcal{F}$, no efficient algorithm can compute $\pi(f)$ much better than random guessing.

This result is *strong* because it proves the existence of a family of programs which cannot be obfuscated. These programs are strongly non-learnable which means that it is impossible to recover its original source code by just executing it, but when looking at the implementation of the program the original source code can be recovered. However, the result is *weak* in the sense that it only shows that every obfuscator completely fails on some programs. It is not proven that every obfuscator will fail on any given program. Some classes of programs may exist which are probably obfuscatable. Customers are only interested in obfuscating their programs and it is not clear that this will not be secure (see [2]).

The result raises some interesting questions, which are not answered yet (see [16]). For example to what sort of programs does the result apply? Do obfuscators exist which are capable of obfuscating programs people are interested in? Can a more practical model be defined, allowing some level of non-critical information to leak from the execution of a program, provided it is not useful information to the attacker?

3.5.4 Problem

The security properties of obfuscation are not well-defined, and therefore obfuscation cannot have a proven security. This means that there is no mathematical proof which shows that if someone can break that implementation, then there is an efficient algorithm for a well known hard computational problem. In the case of obfuscation someone can come up with an obfuscation algorithm and claim that it is secure. People start using the algorithms because they think it is secure. After some time a hacker breaks the algorithm, and then the creators of the algorithm change the algorithm and hope it is secure. Because there is no definition of security, there is not a clearly stated conjecture of the security properties of this algorithm.

3.5.5 Conclusion

The authors of the article hoped to find a formal definition for the security of software obfuscators, and a construction that could be proven to meet this definition. Unfortunately, throughout their research, whenever they came up with a definition, they eventually found a counterexample showing that this definition could not be met.

Although we cannot prove whether or not an obfuscator is good, this does not mean that we should not use obfuscation techniques. In practice we want to achieve a level of security such that the cost to protect the application are less than the cost of the potential damage. It is also important to consider the probability of manifestation of an attack. Moreover, the cost of setting up an attack should be more than the benefit which can be gained from the attack. However, we always have to remember that obfuscation is just a tool which can be used to increase security, but it is not perfect. It should never be used alone, but always in combination with other techniques.

3.6 Using Software Protection Techniques for OMA-DRM

In this section we will look where we can apply the three described techniques in the OMA-DRM context.

- *Software Tamper Resistance:*
  Software tamper resistance techniques can be used to protect software which checks the Rights Objects. When a client has extracted K_{CEK}, additional software checks the Rights Object to see whether the client is allowed to use the key to decrypt the content. The software checks if the MAC value calculated over the Rights Object matches the transmitted MAC value. This software could be changed in such a way that it no longer checks the MAC-values for example. Software tamper resistance techniques can be used to prevent malignant people from doing this.

Software tamper resistance techniques are already used at the start up of a mobile phone. When a mobile phone is started up, a digital signature is made over the whole software. If the software is changed when the telephone was turned off, the system will notice that immediately.

- *Software Obfuscation:*
  In the OMA-DRM context, secret key derivation functions are used which need to be protected against reverse engineering. Another example is that OMA-DRM uses system times. In order to prevent people from setting back the time, the algorithm which checks the system time can be protected by obfuscation techniques.
- *Diversity:*
  Diversity techniques can be used when applying obfuscation techniques for software protection. If different obfuscation techniques are used for each program, we have software diversity.

3.7 Conclusion

Tamper resistance techniques are techniques for providing authenticity or integrity, but not for confidentiality. Obfuscation techniques can be used for confidentiality. Obfuscation techniques make programs more difficult to understand. However, it makes programs slower or larger than the original program. Because no statements about the security of obfuscation can be made, it is recommended to use obfuscation techniques in combination with other software protection techniques. If obfuscation techniques are applied in a different way for each client we will also get software diversity which prevents against automatic attacks.

In this chapter several obfuscating techniques have been discussed separately. In practice a combination of different techniques can be used. More research is needed on the interaction between the different techniques. In the next chapter we will discuss a technique which can be used for protecting secret keys. This technique is called white-box cryptography.

Chapter 4

White-Box Cryptography

4.1 Introduction

In the context of OMA-DRM, it is realistic to analyze an algorithm in an untrusted host environment, where an application is subject to attacks from the host machine itself. Therefore, we will use the white-box attack model where the attacker has total visibility into software implementation and execution. Cryptographic algorithms are used for protecting content, but the secret keys in the implementation are visible to the attacker in the white-box attack context. To prevent an attacker from finding secret keys in the implementation, the keys need to be hidden. This can be done with white-box cryptography.

In this chapter we start with describing a white-box implementation on AES as described by Chow et al. in [8]. After that, we will outline a white-box implementation on DES briefly. We focus on the implementation on AES because AES is used in OMA-DRM. The main goal of the implementations is to prevent the extraction of secret keys from the program.

4.2 Basic Idea

In [8] a method is described for protecting an AES-key in a white-box environment. In the white-box attack context the keys in an AES implementation are completely exposed to an attacker. In this chapter we will describe a technique which hides the AES keys in the implementation.

Consider a basic round of AES (for details on the AES algorithm see [11]). Each round is split into different components. These components will be encoded and represented by lookup tables.

Definition (Encoding):
Let X be a transformation from m to n bits. Choose an m-bit bijection F and an n-bit bijection G. Call $X' = G \circ X \circ F^{-1}$ an encoded version of X. F^{-1} is an input decoding and G is an output encoding.

Each encoding is decoded in another table, which results into a functionally equivalent AES computation. The idea is to make an implementation which consists entirely of encoded lookup tables.

Here we will explain the basic idea with a simple model. Suppose we have as components the following i mappings X which need to be hidden:

$$X_1 \circ X_2 \circ \ldots \circ X_i$$

The bijections $M_1, \ldots, M_{2i}$ are randomly chosen and are put around the components along with their inverses:

$$M_1^{-1} \circ M_1 \circ X_1 \circ M_2 \circ M_2^{-1} \circ M_3^{-1} \circ M_3 \circ X_2 \circ M_4 \circ M_4^{-1} \circ \ldots \circ M_{2i-1}^{-1} \circ M_{2i-1} \circ X_i \circ M_{2i} \circ M_{2i}^{-1}$$

Parts are taken together and put into separate tables. Each table is composed out of different mappings. Two mappings F^{-1} and G are put around it, otherwise the first and the last table are composed out of one mapping which is not secure. F and G^{-1} are also available to the user which he needs to decrypt the message.

$$\underbrace{F^{-1} \circ M_1^{-1}}_{table} \circ \underbrace{M_1 \circ X_1 \circ M_2}_{table} \circ \underbrace{M_2^{-1} \circ M_3^{-1}}_{table} \circ \underbrace{M_3 \circ X_2 \circ M_4}_{table} \circ \ldots \circ \underbrace{M_{2i-1} \circ X_i \circ M_{2i}}_{table} \circ \underbrace{M_{2i}^{-1} \circ G}_{table}$$

The result is that the different components are no longer visible on their own. A similar technique was proposed by Paul Gorissen et al. in [13].

4.3 A White-Box AES Implementation

The input to the AES encryption and decryption algorithm is a single 128-bit block. This block is represented by a 4×4 matrix consisting of bytes. AES consists of 10 rounds for AES-128. Each round updates a set of sixteen bytes which form the state of AES, thus each AES round processes 128 bits. AES-128 uses a key of 128 bits. This key serves as input for an algorithm which converts the key into different round keys of 128 bits. A basic round consists of four parts:

- `SubBytes`
- `ShiftRows`
- `MixColumns`
- `AddRoundKey`

Before the first round an extra `AddRoundKey` operation occurs, and from round ten the `MixColumns` operation is omitted. The only part that uses the key is `AddRoundKey`, the other three parts do nothing with the key.

In the implementation we change the boundaries of the rounds, because we want to compose the `AddRoundKey` step with the `SubBytes` step of the next round into one step. We let a round begin with `AddRoundKey` and `SubBytes` followed by `Shiftrows` and finally `MixColumns`.

4.3.1 Step 1: Hiding the Key in S-Boxes

First, we want to hide the key by composing the SubBytes step and the AddRoundkey together into one step. This makes the key no longer visible on its own. Because the key is known in advance, the operations involving the key can be pre-evaluated. This means that the standard S-Boxes which are used in the step `SubBytes` can be replaced with key-specific S-Boxes. To generate key-specific instances of AES-128, the key is integrated into the `SubBytes` transformations by creating sixteen 8×8 (i.e. 8-bit input, 8-bit output) lookup tables $T^r_{i,j}$ which are defined as follows:

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

where S is the AES S-box (an invertible 8-bit mapping), and $k^r_{i,j}$ is the AES subkey byte at position i, j of the 4×4 matrix which represents the round key for round r. These T-boxes compose the `SubBytes` step with the previous round's `AddRoundKey` step. The round 10 T-boxes absorb the post-whitening key as follows:

$$T^{10}_{i,j}(x) = S(x \oplus k^9_{i,j}) \oplus k^{10}_{sr(ij)} \qquad i = 0, \ldots, 3,\ j = 0, \ldots, 3,$$

where $sr(i, j)$ denotes the new location of cell i, j after the `ShiftRows` step.

In total we have $10 \times 16 = 160$ T-boxes. However, the key can easily be recovered from T-boxes because S^{-1} is publicly known:

$$\begin{aligned} T^r_{i,j}(x) &= S(x \oplus k^{r-1}_{i,j}) \\ S^{-1}(T^r_{i,j}(x)) &= S^{-1}(S(x \oplus k^{r-1}_{i,j})) \\ S^{-1}(T^r_{i,j}(x)) &= x \oplus k^{r-1}_{i,j} \\ k^{r-1}_{i,j} &= x \oplus S^{-1}(T^r_{i,j}(x)) \end{aligned}$$

The key $k^{r-1}_{i,j}$ can be calculated, by choosing an arbitrary x and calculating $x \oplus S^{-1}(T^r_{i,j}(x))$. This makes additional encodings necessary. Linear transformations are used for diffusing the inputs to the T-boxes. These linear transformations are called mixing bijections and can be represented as 8×8 matrices over GF(2). The mixing bijections are inverted by an earlier computation to undo their effect. In the following section we explain the use of mixing bijections in more detail.

4.3.2 Step 2: Inserting Mixing Bijections

An AES state is represented by a 4×4 matrix consisting of bytes. The `MixColumns` step operates on a column (four 8-bit cells) at a time. Consider a 32×32 matrix MC. If this is represented by a table, this table would cost $2^{32} \times 32 = 137438953472$ bits $= 16$ GB. In order to avoid such large tables the matrix is blocked into four sections.

MC is blocked into four 32×8 sections, MC_0, MC_1, MC_2, MC_3. Multiplication of a 32-bit vector $x = (x_0, \ldots, x_{31})$ by MC is done by four separate multiplications yielding four 32-bit vectors $(z_0, \ldots, z_3)$ (Figure 4.1). This is followed by three 32-bits *xors* giving the final 32-bit result z:

$MC \cdot (x_0, \ldots, x_{31})^T = (MC_0 \| MC_1 \| MC_2 \| MC_3) \cdot (x_0, \ldots, x_{31})^T = MC_0 \cdot (x_0, \ldots, x_7)^T \oplus MC_1 \cdot (x_8, \ldots, x_{15})^T \oplus MC_2 \cdot (x_{16}, \ldots, x_{23})^T \oplus MC_3 \cdot (x_{24}, \ldots, x_{31})^T = z_0 \oplus z_1 \oplus z_2 \oplus z_3 = z$

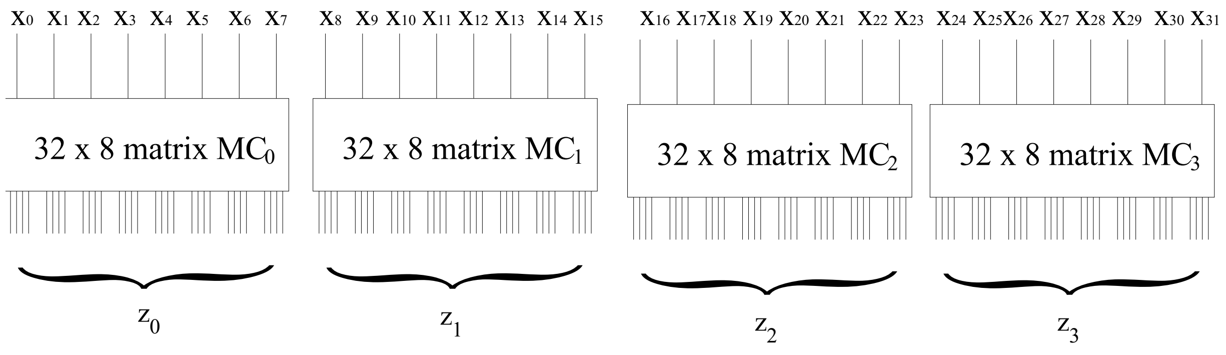


Figure 4.1: MC blocking

The four tables together only cost $4 \times 2^8 \times 32 = 32768$ bits $= 4$ KB.

The three *xors* will be divided into 24 4-bit *xors* with appropriate concatenation (e.g. $((z[0,0], z[0,1], z[0,2], z[0,3]) + (z[1,0], z[1,1], z[1,2], z[1,3])) \| ((z[0,4], z[0,5], z[0,6], z[0,7]) + (z[1,4], z[1,5], z[1,6], z[1,7])) \| \ldots)$. By using these strips and subdivided *xors*, each step is represented by a small lookup table. In particular, for $i = 0, \ldots, 3$ the z_i are computed using 8×32 tables, while the 4-bit *xors* become 24 8×4 tables. Input decodings and output encodings are put around the *xors*. These encodings are randomly chosen non-linear 4×4 bijections. The tables are called type IV tables (Figure 4.2). The type IV tables take as input 4 bits from each of two previous computations. The output encodings of those computations are matched with the input decodings for the type IV tables to undo each other.

The choice for 4×4 non-linear bijections depended on the size of the tables. In this situation a type IV table is only $2^8 \times 4 = 128$ bytes. We need 24 tables which cost together 3 KB. If we did not divide the *xors* we would need three *xor* tables which computed 32-bit *xors*. Ons such a table would cost 2^{24} KB. This is way too large to store.

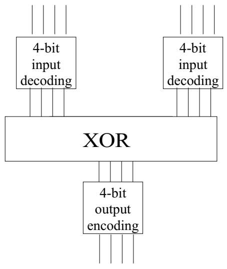


Figure 4.2: Type IV table

The T-boxes and the 8×32 tables could be represented as separate lookup tables. Instead, we compose them creating new 8×32 tables computing the `SubBytes` and `AddRoundKey` transformations as well as part of `MixColumns`. This saves both space (to store the T-boxes) and time (to perform the table lookups).

Before splitting MC into MC_i as above, MC will be left-multiplied by a 32×32 mixing bijection MB chosen as a non-singular matrix with 4×4 submatrices of full rank. The use of mixing bijections increases the number of possible constructions for a particular table.

$$\begin{gathered} MB \circ MC(x_0, \ldots, x_{31})^T = \\ MB \circ (MC_0 \| MC_1 \| MC_2 \| MC_3)(x_0, \ldots, x_{31})^T = \\ MB \circ MC_0(x_0, \ldots, x_7)^T \oplus \ldots \oplus MB \circ MC_3(x_{24}, \ldots, x_{31})^T \end{gathered}$$

We put everything together in an 8×32 type II table surrounded by 4×4 input decodings and 4×4 output encodings (Figure 4.3). These output encodings and input decodings are non-linear 4×4 bijections which must match the input decodings and output encodings of the type IV tables. The type II tables are followed by type IV tables.

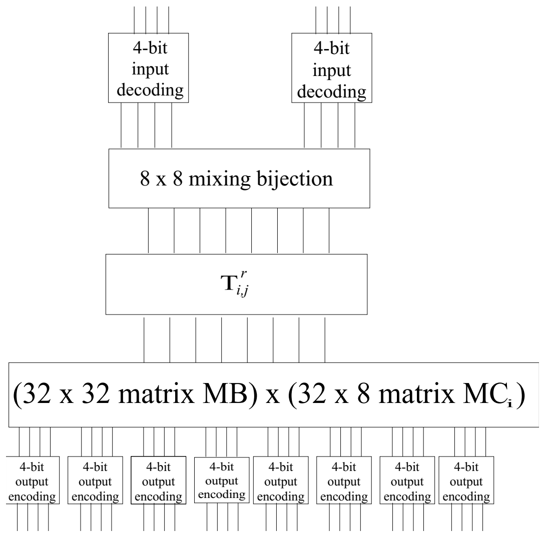


Figure 4.3: Type II table

In order to invert MB, an extra set of tables is used for calculating MB^{-1}. Let $(x'_0, \ldots, x'_{31})$ be the input to `MixColumns`, and let $(z_0, \ldots, z_{31})$ be the output after `MixColumns`. Let $(z'_0, \ldots, z'_{31})^T$ be the result after multiplication with MB. $(z'_0, \ldots, z'_{31})^T$ serves as input to the type III tables.

$$\begin{gathered} MB^{-1} \circ MB \circ MC(x_0, \ldots, x_{31})^T = \\ MB^{-1} \circ MB(z_0, \ldots, z_{31})^T = MB^{-1}(z'_0, \ldots, z'_{31})^T = \\ (MB_0^{-1} \| MB_1^{-1} \| MB_2^{-1} \| MB_3^{-1})(z'_0, \ldots, z'_{31})^T = \\ MB_0^{-1}(z'_0, \ldots, z'_7)^T \oplus MB_1^{-1}(z'_8, \ldots, z'_{15})^T \oplus MB_2^{-1}(z'_{16}, \ldots, z'_{23})^T \oplus MB_3^{-1}(z'_{24}, \ldots, z'_{31})^T \end{gathered}$$

Note that we ignored the input encodings and the output decodings here. This is done because the input decodings of the type III tables undo the output encodings of the type II tables.

In type III tables, MB^{-1} will be left-multiplied by the inverses of the four input mixing bijections of the next round's type II tables, and split into four 32×8 blocks:

$$\begin{gathered} mb^{-1} \circ (MB_0^{-1}(z'_0, \ldots, z'_7)^T \oplus \ldots \oplus MB_3^{-1}(z'_{24}, \ldots, z'_{31})^T) = \\ mb^{-1} \circ MB_0^{-1}(z'_0, \ldots, z'_7)^T \oplus \ldots \oplus mb^{-1} \circ MB_3^{-1}(z'_{24}, \ldots, z'_{31})^T \end{gathered}$$

where mb^{-1} has the following form:

$$mb^{-1} = \begin{pmatrix} mb_0^{-1} & & & \\ & mb_1^{-1} & & \\ & & mb_2^{-1} & \\ & & & mb_3^{-1} \end{pmatrix}$$

where the mb_i^{-1}'s are the inverses of the 8×8 mixing bijection for the next round's type II tables.

We put everything together in an 8×32 type III table surrounded by 4×4 non-linear input decodings and 4×4 non-linear output encodings (Figure 4.4). These tables are followed by corresponding type IV tables.

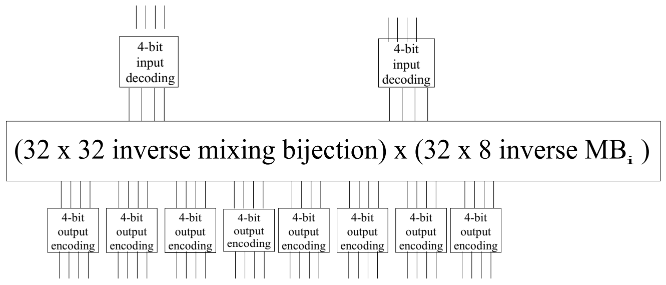


Figure 4.4: Type III table

In Figure 4.5 we show what happens in one round of white-box AES for one strip of 32 bits. One round consists of type II tables plus supporting type IV tables, followed by type III tables plus supporting type IV tables. The type II tables have an input of 8 bits and and output of 32 bits: $(p[i, 0], \ldots, p[i, 31])$ for $i \in 0, \ldots, 3$. The type III tables have an input of 8 bits and an output of 32 bits: $(q[i, 0], \ldots, q[i, 31])$ for $i \in 0, \ldots, 3$. The bits $(q[8i], \ldots, q[8i + 7])$ serve as input bits for the next round's type II tables for $i \in 0, \ldots, 3$.

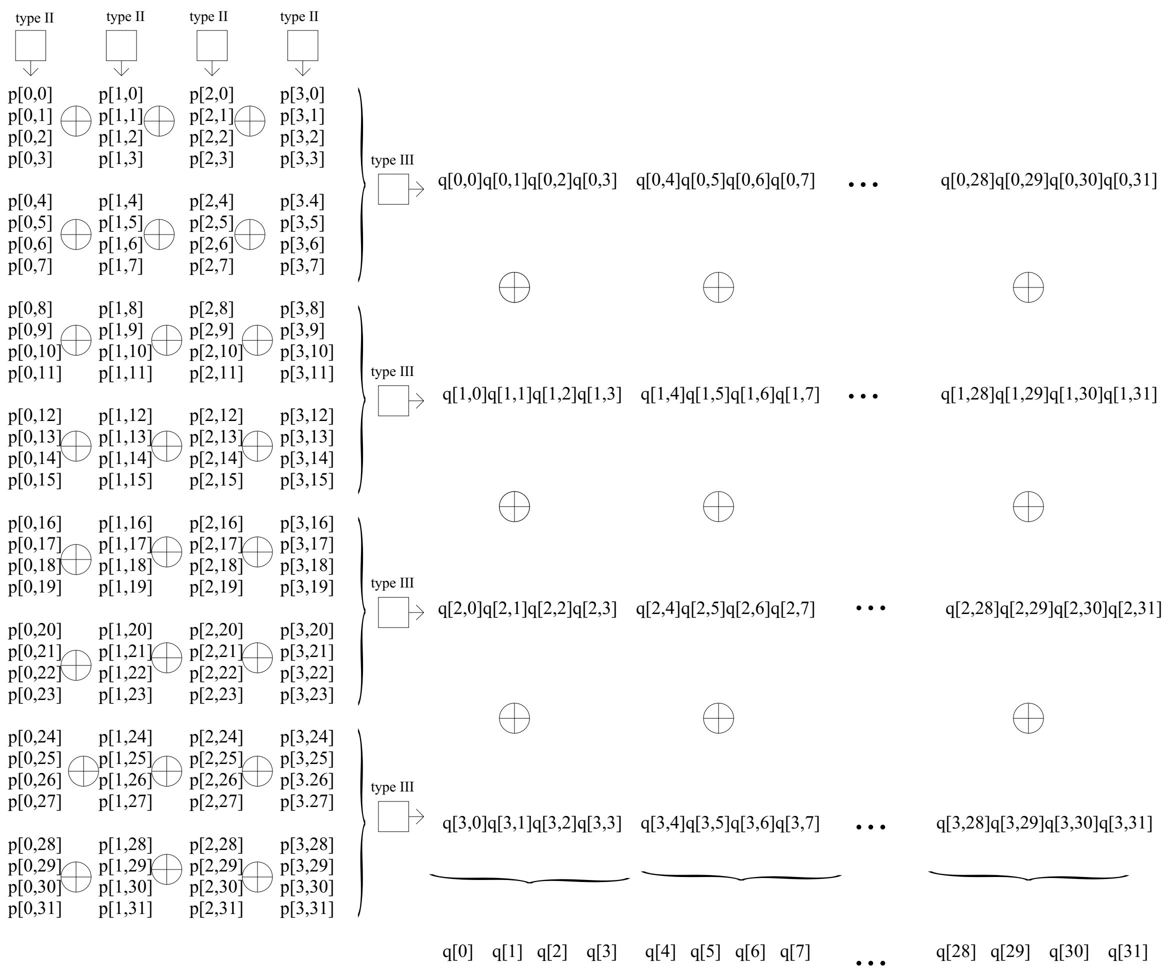

type II
type II
type II
type II
p[0,0] p[1,0] p[2,0] p[3,0]
p[0,1] p[1,1] p[2,1] p[3,1]
p[0,2] p[1,2] p[2,2] p[3,2]
p[0,3] p[1,3] p[2,3] p[3,3]
p[0,4] p[1,4] p[2,4] p[3.4]
p[0,5] p[1,5] p[2,5] p[3,5]
p[0,6] p[1,6] p[2,6] p[3,6]
p[0,7] p[1,7] p[2,7] p[3,7]
p[0,8] p[1,8] p[2,8] p[3,8]
p[0,9] p[1,9] p[2,9] p[3,9]
p[0,10] p[1,10] p[2,10] p[3,10]
p[0,11] p[1,11] p[2,11] p[3,11]
p[0,12] p[1,12] p[2,12] p[3,12]
p[0,13] p[1,13] p[2,13] p[3,13]
p[0,14] p[1,14] p[2,14] p[3,14]
p[0,15] p[1,15] p[2,15] p[3,15]
p[0,16] p[1,16] p[2,16] p[3,16]
p[0,17] p[1,17] p[2,17] p[3,17]
p[0,18] p[1,18] p[2,18] p[3,18]
p[0,19] p[1,19] p[2,19] p[3,19]
p[0,20] p[1,20] p[2,20] p[3,20]
p[0,21] p[1,21] p[2,21] p[3,21]
p[0,22] p[1,22] p[2,22] p[3,22]
p[0,23] p[1,23] p[2,23] p[3,23]
p[0,24] p[1,24] p[2,24] p[3,24]
p[0,25] p[1,25] p[2,25] p[3,25]
p[0,26] p[1,26] p[2,26] p[3.26]
p[0,27] p[1,27] p[2,27] p[3.27]
p[0,28] p[1,28] p[2,28] p[3,28]
p[0,29] p[1,29] p[2,29] p[3,29]
p[0,30] p[1,30] p[2,30] p[3,30]
p[0,31] p[1,31] p[2,31] p[3,31]
type III
q[0,0]q[0,1]q[0,2]q[0,3] q[0,4]q[0,5]q[0,6]q[0,7] ... q[0,28]q[0,29]q[0,30]q[0,31]
type III
q[1,0]q[1,1]q[1,2]q[1,3] q[1,4]q[1,5]q[1,6]q[1,7] ... q[1,28]q[1,29]q[1,30]q[1,31]
type III
q[2,0]q[2,1]q[2,2]q[2,3] q[2,4]q[2,5]q[2,6]q[2,7] ... q[2,28]q[2,29]q[2,30]q[2,31]
type III
q[3,0]q[3,1]q[3,2]q[3,3] q[3,4]q[3,5]q[3,6]q[3,7] ... q[3,28]q[3,29]q[3,30]q[3,31]
q[0] q[1] q[2] q[3] q[4] q[5] q[6] q[7] ... q[28] q[29] q[30] q[31]

Figure 4.5: Part of the tables for one round

4.3.3 Step 3: Inserting External Encodings

Two encodings F^{-1} and G are put around the white-box implementation. These encodings are called the external encodings. F^{-1} is composed of non-linear input decodings and a linear bijection U^{-1}. G is composed of a linear bijection V and non-linear output encodings.

The mixing bijections U^{-1} and V are randomly chosen as 128×128 linear bijections which consist of 1024 4×4 submatrices of full rank. U^{-1} is inserted prior to the first AES `AddRoundKey` operation. To undo the 8×8 mixing bijections for T^1, U^{-1} will be left-multiplied by the inverted input mixing bijections for T^1. The result is split into 128×8 strips, and is followed by 4-bit to 4-bit non-linear input decodings and output encodings (Figure 4.6). These output encodings have to be decoded, *xor*ed together and reencoded to complete the implementation. The output encodings of the last stage of type IV tables supporting U^{-1}invert the input decodings of the type II tables for round 1.

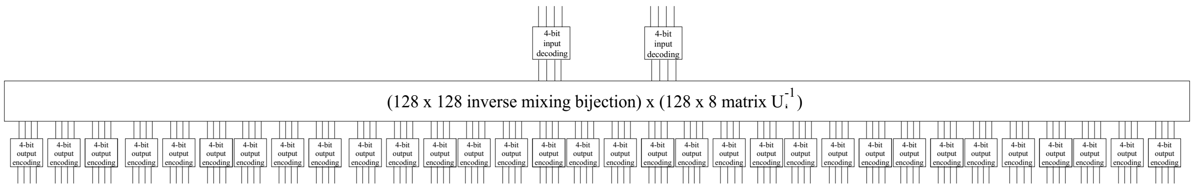

4-bit input decoding
4-bit input decoding
(128 x 128 inverse mixing bijection) x (128 x 8 matrix U_i^{-1})
4-bit output encoding

Figure 4.6: Type Ia table

V is inserted after the last AES `AddRoundKey` operation. V is split into 128×8 strips and is followed by 4-bit to 4-bit non-linear input decodings and output encodings (Figure 4.7).

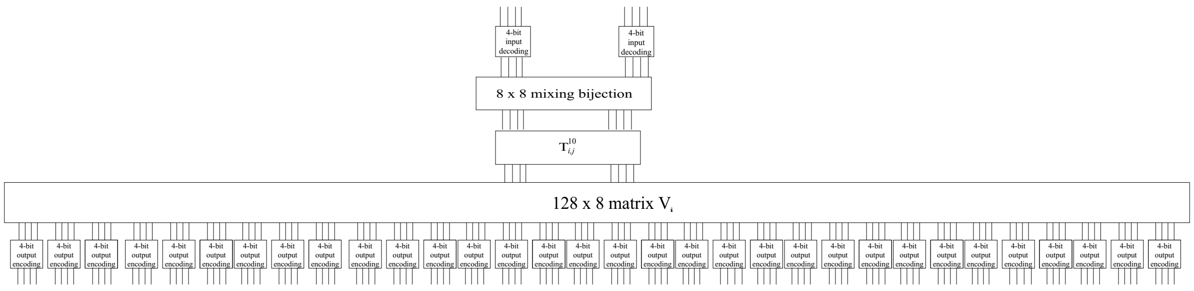


Figure 4.7: Type Ib table

See Appendix A for an overview of the white-box tables, and Appendix B for an overview of how the white-box tables are connected.

4.4 Size & Performance

The AES implementation consists entirely of lookup tables. Sixteen type Ia tables are used for performing the initial input mixing bijection followed by $32 \times 15 = 480$ supporting type IV tables. Each AES round is represented by a set of tables for each of the four 32-bit strips in an AES round. Each strip of the first nine rounds consists of four type II and four type III tables, followed by $3 \times 8 = 24$ type IV tables supporting the type II tables and $3 \times 8 = 24$ type IV tables supporting the type III tables. The type II tables followed by the corresponding type IV tables represent `AddRoundKey`, `SubBytes` and `MixColumns`. The type III tables followed by the corresponding type IV tables represent the mixing bijections which undo the mixing bijections at the end of the previous type II tables, and they undo the mixing bijections at the beginning of the type II tables for the next round.

`ShiftRows` is executed during the rounds by providing appropriately shifted data as input to the type III tables. Sixteen type Ib tables are used for performing the output mixing bijection followed by $32 \times 15 = 480$ supporting type IV tables. The size of the lookup tables in this implementation is:

Type Ia: $16 \times (2^8 \times 128) = 524288$ bits
Type IV supporting Type I: $480 \times (2^8 \times 4) = 491520$ bits
Type II: $9 \times 16 \times (2^8 \times 32) = 1179648$ bits
Type IV supporting Type II: $9 \times 4 \times 24 \times (2^8 \times 4) = 884736$ bits
Type III: $9 \times 16 \times (2^8 \times 32) = 1179648$ bits
Type IV supporting Type III:$9 \times 4 \times 24 \times (2^8 \times 4) = 884736$ bits
Type Ib: $16 \times (2^8 \times 128) = 524288$ bits
Type IV supporting Type Ib: $480 \times (2^8 \times 4) = 491520$ bits

The total size of the lookup tables is: 524288 bits $+491520$ bits $+1179648$ bits $+884736$ bits $+1179648$ bits $+884736$ bits $+524288$ bits $+491520$ bits $= 6160384$ bits $= 770048$ bytes

The described implementation makes white-box attacks more difficult by representing the different encoding steps by lookup tables. Although the security improves, the implementations make a program much larger and slower. The AES implementation of Daemen and Rijmen requires 4352 bytes for lookup tables [11], thus the expected increase in size is about $177\times$. The performance slowdown is approximately $55\times$ compared to a normal implementation of AES. However, this is also dependent on the layout of the tables in memory [8]. In §6.5 we look for possibilities to use less tables. Because of the increase in memory space and the performance slowdown, it is important to consider carefully where to apply white-box cryptography.

Note that in the article an AES-encryption implementation is described. However, the decryption algorithm of AES can be written in the same way as the encryption algorithm. The difference is that SubBytes, ShiftRows and MixColumns becomes InverseSubBytes, InverseShiftRows and InverseMixColumns.

4.5 Security

As with the different obfuscation techniques, it is difficult to claim something about the security. However, it is possible to count the number of different possibilities for the tables of each type. This enables us to say something about the possibility of a brute force attack. We start by giving the number of different possible tables without looking at the underlying structure.

The type Ia tables and the type Ib tables have an input of 8 bits and an output of 128 bits. This means that an upper bound for the number of possibilities for those tables is: $2^{2^8 \times 128} = 2^{32768}$. The type II tables and the type III tables have an input of 8 bits and an output of 32 bits. An upper bound for the number of different tables is $2^{2^8 \times 32} = 2^{8192}$. The type IV tables have an input of 8 bits and an output of 4 bits. The upper bound for the number of possibilities for those tables is $2^{2^8*4} = 2^{1024}$. We have 16 type Ia tables, 16 type Ib tables, 144 type II tables, 144 type III tables, 2688 type IV tables, and 16 type Ib tables. The total complexity becomes: $(2^{32768})^{32} \times (2^{8192})^{288} \times (2^{1024})^{2688} = 2^{6160384}$. This is much too high for a brute force attack to be effective. However, if we look at the underlying structure, the actual number of different tables can be much lower.

4.5.1 White-Box Diversity

We can also look at the number of distinct constructions which exist for the possible tables of a type. This metric is called white-box diversity [8]. All encodings are assumed to be random and independent. To obtain the white-box diversity the possible encoded steps need to be counted. If constructing a table requires n independent choices to be made, and the i'th choice has c_i alternatives, then the white-box diversity for the tables is:

$$\prod_{i=1}^{n} c_i$$

First, we calculate the number of constructions for the different parts of the tables:

- For each input decoding or output encoding of 4 bits to 4 bits there are $16!$ possibilities.
- The number of nonsingular $i \times i$ matrices can be calculated as follows:

  $$\#\text{nonsingular } i \times i \text{ matrices} = (2^i - 1) \times \prod_{j=1}^{i-1} (\; 2^i - 1 - \sum_{k=1}^{j} \binom{j}{k} \;)$$

  There are 20160 nonsingular 4×4 matrices. Because there fit 64 4×4 submatrices in a 128×8 matrix, an upper bound for the number of different constructions for the 128×8 matrices is 20160^{64}. The actual number is lower because not every composition of non-singular submatrices results in a non-singular matrix.
- The number of different constructions for the 8×8 mixing bijections is $2^{62.2}$.
- The number of possibilities for the T-boxes is $2^8 = 256$.
- There exist 20160 nonsingular 4×4 matrices. Sixteen 4×4 submatrices fit in a 32×8 matrix. The upper bound for the number of 32×8 matrices is thus 20160^{16}. The actual number is lower because not every composition of non-singular submatrices results in a non-singular matrix.

Upper bounds of the white-box diversity for the different table types are:

Type Ia: $(16!)^2 \times 20160^{64} \times (16!)^{32} \approx 2^{2419.7}$
Type II: $(16!)^2 \times 2^{62.2} \times 2^8 \times 20160^{16} \times 4 \times (16!)^8 \approx 2^{743.5}$
Type III: $(16!)^2 \times 20160^{16} \times 4 \times (16!)^8 \approx 2^{673.3}$
Type IV: $(16!)^2 \times 16! \approx 2^{132.8}$
Type Ib: $(16!)^2 \times 2^{62.2} \times 2^8 \times 20160^{64} \times (16!)^{32} \approx 2^{2489.9}$

These upper bounds can be lowered if we consider the method that is used for generating large non-singular matrices with 4×4 submatrices (see [19]). This method limits the number of possible constructions, by constructing the matrix inductively. The first step is finding a 4×4 non-singular matrix. The second step is to construct a 8×8 matrix, consisting of four 4×4 sub-matrices. The next step is to expand this matrix to a 12×12 matrix, by adding a row of blocks and a column of blocks to the 8×8 matrix. These rows and columns are chosen from the 8×8 matrix, which makes them dependent on this matrix. We only have to add one 4×4 block to create a 12×12 matrix. These steps can be repeated until we reach the size of the non-singular matrix we need.

An upper bound for the number of $i \times i$ non-singular matrices consisting of 4×4 non-singular submatrices can be calculated as follows:

$$\#\text{non-singular } i \times i \text{ matrices consisting of } 4 \times 4 \text{ submatrices} = 20160^4 \cdot \prod_{j=2}^{i/4-1} 20160 j^2$$

Figure 4.8 shows the construction of a non-singular 32×32 matrix. The white boxes are 4×4 nonsingular matrices which we can choose arbitrarily. The grey boxes are 4×4 nonsingular matrices which we choose from blocks that are chosen before:

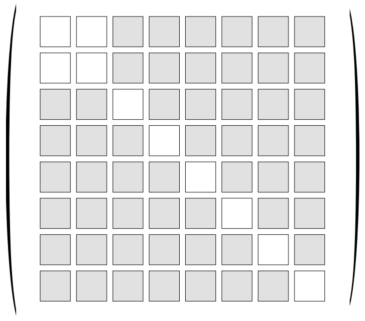


Figure 4.8: Generating a 32×32 matrix consisting of 4×4 submatrices

Now we have the following upper bounds of the white-box diversity:

Type Ia: $(16!)^2 \times 2^{22.2} \times (16!)^{32} \approx 2^{1483.5}$
Type II: $(16!)^2 \times 2^{62.2} \times 2^8 \times 2^{41.9} \times 4 \times (16!)^8 \approx 2^{556.6}$
Type III: $(16!)^2 \times 2^{41.9} \times (16!)^8 \approx 2^{486.4}$
Type IV: $(16!)^2 \times 16! \approx 2^{132.8}$
Type Ib: $(16!)^2 \times 2^{62.2} \times 2^8 \times 2^{22.2} \times (16!)^{32} \approx 2^{1596.9}$

4.5.2 White-Box Ambiguity

We discussed how many different constructions exist. However, some constructions produce the same tables. Thus the number of different constructions is higher than the number of different tables. Therefore, the authors of [8] introduced another metric: the white-box ambiguity of a table type. White-box ambiguity estimates the number of distinct constructions which produce exactly the same table. They have found the following estimates:

Type Ia: $2^{546.1}$
Type II: 2^{117}
Type III: 2^{117}
Type IV: $2^{48.2}$

Note that they did not give separate estimates for the type Ib tables. They consider the type Ib tables the same as the type Ia tables. The number of different tables can be calculated by dividing the number of distinct constructions (white-box diversity) by the number of distinct constructions which produce the same type of tables (white-box ambiguity):

Type Ia: $2^{1483.5}/2^{546.1} = 2^{937.4}$
Type II: $2^{556.6}/2^{117} = 2^{439.6}$
Type III: $2^{486.4}/2^{117} = 2^{369.4}$
Type IV: $2^{132.8}/2^{48.2} = 2^{84.6}$

Remember that we have 16 type Ia tables, 16 type Ib tables, 144 type II tables, 144 type III tables, and 2688 type IV tables. The total complexity becomes: $(2^{937.4})^{32} \times (2^{439.6})^{144} \times (2^{369.4})^{144} \times (2^{84.6})^{2688} = 2^{373898}$. This is still too much for a brute force attack.

4.6 White-Box Cryptography for DES

In [7] a technique is described of applying white-box cryptography to DES. DES consists of permutations, S-Box lookups and *xor* operations. The idea is to apply encodings to each of these steps. For S-Box lookups and *xor* operations, encoding each operation (along with its input and output) is performed in the same way as for the white-box implementation on AES. Applying encodings to the various permutations is more difficult. This is because these permutations are very simple, and it is difficult to hide the information being manipulated. To make the encodings more difficult, the DES permutations and the bitwise *xor* operations will be expressed as affine transformations. These transformations are still very simple because the permutations in DES have very sparse matrices (one or two 1-bits per row or column). In order to diffuse information over more bits, such a permutation P can be represented by $J \circ K$, where K is a mixing bijection and $J = PK^{-1}$, thereby replacing a sparse matrix with two non-sparse ones. These matrices are represented by lookup tables. The main idea is to make an implementation which consists entirely of look-up tables.

4.7 Conclusion

White-box cryptography can be used for hiding keys in an untrusted host environment. The result is a functionally equivalent program in which the key is no longer visible. This prevents malignant people for finding their own keys and distributing those keys. The advantage of distributing keys compared to distributing content is that keys are usually much smaller than content and they can be distributed more easily.

White-box cryptography also has some drawbacks. By using white-box cryptography more memory is needed and it causes a slowdown of the program. Whenever a key needs to be updated, the whole set of white-box tables needs to be updated too. This can be a problem if the amount of storage space is limited. Another drawback is that the whole white-box implementation can be used as a key. Moreover, the lack of both security metrics and security proof is also a drawback. More research is needed on this area.

Recently, an attack on the described white-box AES implementation was published in [4] by Billet et al.. They explain a technique for obtaining the key from the lookup tables. In the next chapter this attack will be described in more detail.

Chapter 5

Attack on a White-Box AES Implementation

5.1 Introduction

In the white-box attack context an attacker has total access to the implementation of cryptographic algorithms. Moreover, he can observe or manipulate the dynamic execution of the algorithm or parts of the algorithm. This makes it possible for an attacker to find keys easily. This can be prevented using white-box cryptography by hiding the keys in lookup tables. In [8] Chow et al. described an implementation of white-box cryptography for AES. Recently an attack on this implementation was published by Billet et al. in [4]. In this chapter the attack will be described in more detail.

5.2 Basic Idea of the Attack

In the described white-box AES implementation, the round keys are hidden in lookup-tables. Because it is very difficult to extract the keys by local inspections of the tables, it is more convenient to look at the input and the output of the composition of tables for a round. A round consists of type II tables, type III tables and supporting type IV tables. This can be represented as follows:

$$\underbrace{mb^{-1} \circ MB^{-1}}_{\text{in Type III tables}} \circ \underbrace{MB(z_0, \ldots, z_{31})^T}_{\text{in Type II tables}} = mb^{-1} \circ (z_0, \ldots, z_{31})^T = \begin{pmatrix} mb_0^{-1}(z_0, \ldots, z_7)^T \\ mb_1^{-1}(z_8, \ldots, z_{15})^T \\ mb_2^{-1}(z_{16}, \ldots, z_{23})^T \\ mb_3^{-1}(z_{24}, \ldots, z_{31})^T \end{pmatrix} = \begin{pmatrix} y_0 \\ y_1 \\ y_2 \\ y_3 \end{pmatrix}$$

where $(z_0, \ldots, z_{31})$ is the state of AES after `ShiftRows` and `Mixcolumns` and

$$mb^{-1} = \begin{pmatrix} mb_0^{-1} & & & \\ & mb_1^{-1} & & \\ & & mb_2^{-1} & \\ & & & mb_3^{-1} \end{pmatrix}$$

Note that we ignored the input decodings of the type III tables and the output encodings of the type II tables. This is because the input decodings of the type III tables undo the output encodings of the type II tables.

Because we look at the composition of the tables, MB and MB^{-1} cancel each other. See §4.3.2 voor details on the mixing bijections. As shown above the $xors$ supporting the type III tables are no longer needed.

Each round of AES consists of four mappings between four input bytes and four output bytes. These four mappings together form an encoded AES round. We put everything together in Figure 5.1 where one such mapping is shown. We assume that we can choose the input bits and we can only observe the corresponding output bits.

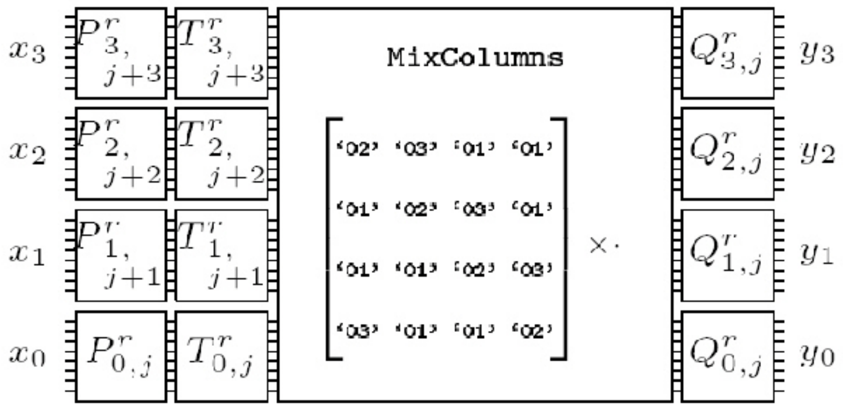


Figure 5.1: One of the four mappings for a single AES round

Here $P^r_{i,j}$ is constructed as the composition of two 4-bit to 4-bit input decodings, and one 8-bit to 8-bit mixing bijection of the type II tables. $Q^r_{i,j}$ is constructed as one 8-bit to 8-bit mixing bijection, and two 4-bit to 4-bit output encodings. The 8-bit to 8-bit mixing bijections are submatrices of the 32×32 mixing bijection for a type III table. Remember that the mixing bijection plus output encodings and input decodings for the next round and the mixing bijection for the next round cancel each other out. Thus each $Q^r_{i,j}$ is the inverse of $P^{r+1}_{i,j}$.

The goal of the attack is to extract the AES round keys. The attack proceeds in three steps:

1. Recovering the non-affine parts of $Q^r_{i,j}$ for round $r = 1, ..., 9$ (see Theorem 1). Since each $Q^r_{i,j}$ is the inverse of $P^{r+1}_{i,j}$, the non-affine parts of $P^{r+1}_{i,j}$ can also be recovered for round $r+1, r = 1, ..., 9$.

2. Recovering the affine parts of the $Q^r_{i,j}$'s.

3. Extracting the AES round keys.

5.3 Step 1: Recovering the Non-Affine Part of $Q^r_{i,j}$

We want to recover the non-affine part of the Q^r_i's in round $r = 1, ..., 9$. Consider y_0 as a function of (x_0, x_1, x_2, x_3) and fix x_1, x_2 and x_3 to some constants c_1, c_2 and c_3. $y_0(x, c_1, c_2, c_3)$ can be written as follows:

$$y_0(x, c_1, c_2, c_3) = Q^r_{0,j}(\alpha T^r_{0,j}(P^r_{0,j}(x)) \oplus \beta_{c_1,c_2,c_3})$$

and also:

$$y_0(x, c_1, c_2, c_3)^{-1} = (P^r_{0,j})^{-1}((T^r_{0,j})^{-1}\alpha^{-1}((Q^r_{0,j})^{-1}(x) \oplus \beta_{c_1,c_2,c_3}))$$

where α is a constant independent of c_1, c_2, and c_3, and β_{c_1,c_2,c_3} is another constant. Since x can only take $2^8 = 256$ values, it is possible to calculate all possible $y_0(x, c_1, c_2, c_3)$'s as well as their inverses, and to produce lookup tables of these functions.

If we vary one constant, say c_3, we get the following (we drop the r's and j's to in order to make it more readable):

$$\begin{gathered} y_0(y_0(x, c_1, c_2, c_3')^{-1}, c_1, c_2, c_3) = \\ Q_0(\alpha T_0(P_0(P_0^{-1}(T_0^{-1}\alpha^{-1}(Q_0^{-1}(x) \oplus \beta_{c_1,c_2,c_3'})))) \oplus \beta_{c_1,c_2,c_3}) = \\ Q_0(\alpha T_0(T_0^{-1}\alpha^{-1}(Q_0^{-1}(x) \oplus \beta_{c_1,c_2,c_3'})) \oplus \beta_{c_1,c_2,c_3}) = \\ Q_0(Q_0^{-1}(x) \oplus \beta_{c_1,c_2,c_3'} \oplus \beta_{c_1,c_2,c_3}) = \\ Q_0(Q_0^{-1}(x) \oplus \beta) \end{gathered}$$

where $\beta = \beta_{c_1,c_2,c_3'} \oplus \beta_{c_1,c_2,c_3}$.

Theorem 1 *Given a set of functions $S = \{Q \circ \oplus_\beta \circ Q^{-1}\}_{\beta \in GF(2^8)}$ given by values, where Q is a permutation of GF(2^8) and $\oplus_\beta$ is the translation by β in GF(2^8) , one can construct a particular solution $\overline{Q}$ such that there exists an affine mapping R so that $\overline{Q} = Q \circ R$.*

The proof of this theorem, and the algorithm to construct $\overline{Q}$ can be found in [1]. The theorem enables us to compute the non-linear part $\overline{Q}^r$ of Q^r with time complexity 2^{24}. Now we know that an affine mapping R^r exists such that $\overline{Q^r} = Q^r \circ R^r$. Since Q^r is the inverse of P^{r+1} we have:

$$P^{r+1} \circ Q^r(x) = x$$
$$P^{r+1} \circ \overline{Q}^r \circ (R^r)^{-1}(x) = x$$

Because $P^{r+1} \circ Q^r$ must be the identity, we know that $P^{r+1} \circ \overline{Q}^r$ must be R^r. R^r will be called $\widetilde{P}^{r+1}$, and $(R^r)^{-1}$, the affine part of Q^r, will be called $\widetilde{Q}^r$. y also changes:

$$Q^r(\ldots) = y$$
$$\overline{Q}^r \circ (R^r)^{-1}(\ldots) = y$$
$$(R^r)^{-1}(\ldots) = (\overline{Q}^r)^{-1}y$$
$$\widetilde{Q}^r(\ldots) = \widetilde{y}$$

where $\widetilde{y} = (\overline{Q}^r)^{-1}y$. We can construct $\overline{Q}$ according to Theorem 1, and therefore we also know $\widetilde{y}$. The original problem of recovering P and Q which are both non-linear is reduced to the problem of recovering the affine mappings $\widetilde{P}$ and $\widetilde{Q}$. Now we are in the same situation as in figure 5.1 except for the fact that P is changed to $\tilde{P}$, Q is changed to $\tilde{Q}$, and y is changed to $\tilde{y}$. In order to improve the readability whenever P, Q, and y are mentioned, we actually mean $\widetilde{P}$, $\widetilde{Q}$, and $\widetilde{y}$.

5.4 Step 2: Recovering the Affine Part of $Q^r_{i,j}$

The functions y_i can be written as:

$$y_i(x_0, x_1, x_2, x_3) = Q_i(\alpha_{i,0} \cdot T_0 \circ P_0(x_0) \oplus \alpha_{i,1} \cdot T_1 \circ P_1(x_1) \oplus \alpha_{i,2} \cdot T_2 \circ P_2(x_2) \oplus \alpha_{i,3} \cdot T_3 \circ P_3(x_3))$$

However, we do not know the values of the α's, because of the `Shiftrows` step.

Proposition 1. *For any pair (y_i, y_j) as introduced above, there exists a unique linear mapping L and a constant c such that,*

$$\forall x_0 \in \text{GF}(2^8), y_i(x_0, 00, 00, 00) = L(y_j(x_0, 00, 00, 00)) \oplus c.$$

The proof of the proposition can be found in [1]. By using that $y_i(x, 00, 00, 00) = Q_i(\alpha_{i,0} \cdot T_0(P_0(x)) \oplus c_i) = A_i(\alpha_{i,0} \cdot T_0(P_0(x)) \oplus c_i) \oplus q_i$, and $y_j(x, 00, 00, 00) = Q_j(\alpha_{j,0} \cdot T_0(P_0(x)) \oplus c_j) = A_j(\alpha_{j,0} \cdot T_0(P_0(x)) \oplus c_j) \oplus q_j$, we can conclude that L has the following form: $L = A_i \circ \Lambda_{\alpha_{i,0}/\alpha_{j,0}} \circ A_j^{-1}$.

If we vary the second, third, or fourth variable, and if we keep the other variables fixed, we can make an analogous statement.

Given two functions y_i and y_j, the corresponding (L, c) can be determined by solving the linear system of equations below:

$$\begin{pmatrix} x_0 & x_1 & x_2 & x_3 & x_4 & x_5 & x_6 & x_7 \\ x_8 & x_9 & x_{10} & x_{11} & x_{12} & x_{13} & x_{14} & x_{15} \\ x_{16} & x_{17} & x_{18} & x_{19} & x_{20} & x_{21} & x_{22} & x_{23} \\ x_{24} & x_{25} & x_{26} & x_{27} & x_{28} & x_{29} & x_{30} & x_{31} \\ x_{32} & x_{33} & x_{34} & x_{35} & x_{36} & x_{37} & x_{38} & x_{39} \\ x_{40} & x_{41} & x_{42} & x_{43} & x_{44} & x_{45} & x_{46} & x_{47} \\ x_{48} & x_{49} & x_{50} & x_{51} & x_{52} & x_{53} & x_{54} & x_{55} \\ x_{56} & x_{57} & x_{58} & x_{59} & x_{60} & x_{61} & x_{62} & x_{63} \end{pmatrix} \begin{pmatrix} y_{i,0} \\ y_{i,1} \\ y_{i,2} \\ y_{i,3} \\ y_{i,4} \\ y_{i,5} \\ y_{i,6} \\ y_{i,7} \end{pmatrix} \oplus \begin{pmatrix} c_0 \\ c_1 \\ c_2 \\ c_3 \\ c_4 \\ c_5 \\ c_6 \\ c_7 \end{pmatrix} = \begin{pmatrix} y_{j,0} \\ y_{j,1} \\ y_{j,2} \\ y_{j,3} \\ y_{j,4} \\ y_{j,5} \\ y_{j,6} \\ y_{j,7} \end{pmatrix}$$

where $y_{i,k}$ stands for the k'th element in y_i, and $y_{j,k}$ stands for the k'th element in y_j, $k = 0, \ldots, 7$. Because we know y_i and y_j we can form a highly overdefined linear system of $2^8 \times 8$ equations involving $64 + 8 = 72$ unknowns. According to Billet et al. [4], this equation can be solved with a time complexity much lower than 2^{16}.

Applying proposition 1 with $(i, j) = (0, 1)$, L can be determined by solving the linear system of equations. L has the following form: $L = A_0 \circ \Lambda_{\alpha_{0,0}/\alpha_{1,0}} \circ A_1^{-1}$. If A_0 is known and we know the right values of the α's, A_1 can be determined. Using the same argument with $(i, j) = (0, 2)$, A_2 can be determined, and with $(i, j) = (0, 3)$, A_3 can be determined. Thus from the knowledge of the linear part of Q_0, the linear parts of Q_1, Q_2 and Q_3 can be computed. Thus a linear set of equations needs to be solved three times with a total time complexity of $3 \cdot 2^{16}$. Now we only need to focus on the determination of Q_0.

Q_i is an affine mapping which can be decomposed into a linear and constant part: $Q_i(x) = A_i(x) + q_i$. If we apply proposition 1 with $(i, j) = (0, 1)$, we will get $L_0 = A_0 \circ \Lambda_{\alpha_{0,0}/\alpha_{1,0}} \circ A_1^{-1}$. Using a variant of proposition 1 with $(i, j) = (0, 1)$, but where one varies x_1 instead of x_0, we will obtain $L_1 = A_0 \circ \Lambda_{\alpha_{0,1}/\alpha_{1,1}} \circ A_1^{-1}$. Now we are able to compute:

$$L = L_0 \circ L_1^{-1} = A_0 \circ \Lambda_\beta \circ A_0^{-1}, \text{ where } \beta = \alpha_{0,0}\alpha_{1,1}/\alpha_{0,1}\alpha_{1,0}.$$

The values α stand for the `MixColumns` coefficients which take only the values 00, 01, 02, and 03. It can be checked that only 16 values of β remain possible:

{20, 8d, 30, f6, 40, cb, 60, 7b, 50, 52, a4, 8f, f7, 8c, 46, f5}

We have the following situation. We are able to determine L and we know that L has the following form: $L = L_0 \circ L_1^{-1} = A_0 \circ \Lambda_\beta \circ A_0^{-1}$. The correct β can be found by determining the characteristic polynomial of L, and the characteristic polynomial for Λ_β. These polynomials are the same. If we know β, we are able to determine A_0. Next will be described how to determine A_0.

Proposition 2. *Given an element β of GF(2^8) not contained in any subfield of GF(2^8) and its corresponding matric $L = A_0 \circ \Lambda_\beta \circ A_0^{-1}$, we can compute with time complexity lower than 2^{16}, a matrix $\overline{A}_0$ such that there exists a unique non-zero constant γ in GF(2^8), such that $\overline{A}_0 = A_0 \circ \Lambda_\gamma$.*

Next we will explain how to recover γ in order to determine A_0, the linear part of Q_0, and how to determine the constant part q_0 of Q_0, to recover Q_0 completely.

5.4.1 Determining γ

Proposition 3.*There exist unique pairs $(\delta_i, c_i)_{i=0,\ldots,3}$ of elements in GF(2^8) δ_i being non-zero, such that*

$$\begin{aligned}
P_0^* &: x \to (S^{-1} \circ \Lambda_{\delta_0} \circ \overline{A}_0^{-1})(y_0(x, 00, 00, 00) \oplus c_0),\\
P_1^* &: x \to (S^{-1} \circ \Lambda_{\delta_1} \circ \overline{A}_0^{-1})(y_0(00, x, 00, 00) \oplus c_1),\\
P_2^* &: x \to (S^{-1} \circ \Lambda_{\delta_2} \circ \overline{A}_0^{-1})(y_0(00, 00, x, 00) \oplus c_2),\\
P_3^* &: x \to (S^{-1} \circ \Lambda_{\delta_3} \circ \overline{A}_0^{-1})(y_0(00, 00, 00, x) \oplus c_3),
\end{aligned}$$

are affine mappings. Any pair (δ_i, c_i) can be computed with time complexity 2^{24}. Moreover, those mappings are exactly $P_i^*(x) = P_i(x) \oplus k_i$.

Now we show how we got this result for P_0^*:

$$\begin{gathered}
y_0(x, 00, 00, 00) = Q_0(\alpha T(P_0(x)) \oplus c)\\
y_0(x, 00, 00, 00) = A_0(\alpha T(P_0(x)) \oplus c) \oplus c'\\
y_0(x, 00, 00, 00) = A_0(\alpha T(P_0(x)) \oplus A_0(c) \oplus c'\\
y_0(x, 00, 00, 00) = A_0(\alpha T(P_0(x)) \oplus c_0\\
y_0(x, 00, 00, 00) \oplus c_0 = A_0(\alpha T(P_0(x))\\
y_0(x, 00, 00, 00) \oplus c_0 = \overline{A_0} \circ \Lambda_\gamma(\alpha T(P_0(x)))\\
\overline{A}_0^{-1}(y_0(x, 00, 00, 00) \oplus c_0) = \Lambda_\gamma(\alpha T(P_0(x))\\
\Lambda_{\gamma^{-1}} \overline{A}_0^{-1}(y_0(x, 00, 00, 00) \oplus c_0) = \alpha T(P_0(x))\\
\Lambda_{\alpha^{-1}\gamma^{-1}} \overline{A}_0^{-1}(y_0(x, 00, 00, 00) \oplus c_0) = S(P_0(x) \oplus k)\\
S^{-1}\Lambda_{\alpha^{-1}\gamma^{-1}} \overline{A}_0^{-1}(y_0(x, 00, 00, 00) \oplus c_0) = P_0(x) \oplus k\\
S^{-1}\Lambda_{\delta_0} \overline{A}_0^{-1}(y_0(x, 00, 00, 00) \oplus c_0) = P_0(x) \oplus k
\end{gathered}$$

For each possible pair of (δ_i, c_i) the corresponding mapping can be tested in order to check whether or not it is affine. There are 2^{16} possible pairs, from which four unique pairs (δ_i, c_i) with $\delta_i^{-1} = \gamma \cdot \alpha_{0,i}$ remain after the testing. Since two of the four $\alpha_{0,i}$'s are 01 and the other ones are 02 and 03, we know that two of the four δ's are the same, and for those δ's we know that: $\delta^{-1} = \gamma \cdot 01$, and thus $\gamma = \delta^{-1}$. Now that γ is known, we are able to determine $A_0 = \overline{A}_0 \circ \Lambda_\gamma$.

5.4.2 Determining q_o

Now that we know A_0, we only need to determine the constant q_o of the affine mapping Q_0 to recover Q_0 completely. Define $c_4 = y_0(00,00,00,00) = A_0(\bigoplus_{i=0}^{3} \alpha_{0,i} \cdot T_i \circ P_i(00)) \oplus q_0$

$$\begin{aligned}
c_0 &= y_0(x,00,00,00) \oplus A_0(\alpha_{0,0} \cdot T_0(P_0(x))),\\
c_1 &= y_0(00,x,00,00) \oplus A_1(\alpha_{0,1} \cdot T_1(P_1(x))),\\
c_2 &= y_0(00,00,x,00) \oplus A_2(\alpha_{0,2} \cdot T_2(P_2(x))),\\
c_3 &= y_0(00,00,00,x) \oplus A_3(\alpha_{0,3} \cdot T_3(P_3(x))),
\end{aligned}$$

which holds for each x, thus also for $x = 00$:

$$\begin{gathered}
q_0 = A_0(\bigoplus_{i=0}^{3} \alpha_{0,i} \cdot T_i \circ P_i(00)) + c_4\\
= A_0(\alpha_{0,0} \cdot T_0(P_0(00))) \oplus A_1(\alpha_{0,1} \cdot T_1(P_1(00))) \oplus A_2(\alpha_{0,2} \cdot T_2(P_2(00))) \oplus A_3(\alpha_{0,3} \cdot T_3(P_3(00))) + c_4\\
= c_0 \oplus y_0(00,00,00,00) \oplus c_1 \oplus y_0(00,00,00,00) \oplus c_2 \oplus y_0(00,00,00,00) \oplus c_3 \oplus y_0(00,00,00,00) \oplus c_4\\
= c_0 + c_1 + c_2 + c_3 + c_4
\end{gathered}$$

Thus the constant part of Q_0 is $q_0 = c_0 + c_1 + c_2 + c_3 + c_4$. At this point, we have determined Q_0 completely.

5.5 Step 3: Extracting the AES Round Keys

The final step is the extraction of the round keys. According to Proposition 3:

$$P^*_{i,j}(x) = P_{i,j}(x) \oplus k_{i,j}$$

Therefore we also know:

$$\begin{gathered}
P^{*r+1}_{i,j}(Q^r_{i,j}(x)) = P^{r+1}_{i,j}(Q^r_{i,j}(x)) \oplus k^{r+1}_{i,j}\\
P^{*r+1}_{i,j}(Q^r_{i,j}(x)) = x \oplus k^{r+1}_{i,j}
\end{gathered}$$

Thus $k^{r+1}_{i,j} = P^{*r+1}_{i,j}(Q^r_{i,j}(x)) \oplus x$ can be calculated for a certain round. However, we do not know the right order of the bytes. The algorithm which generates round keys of AES calculates a round key by using the round key of the previous round as input of the algorithm. This implies that we only need to find Q's for two consecutive round. In order to check the right order of the bytes, we need to calculate $k_{i,j}$ for the next round, and use the algorithm to compare bytes and put them in the right order. Thus recovering one round key correctly, enables us to recover each round key.

5.6 Summary and Conclusion

The goal of the attack described in [4] is to extract the AES keys. The attack will only work if all tables of the white-box AES implementation are available to an attacker. The attacker must be able to choose the input and observe the output of these tables. If these conditions are satisfied an attacker is able to find the key.

In order to find the key an attacker must know what the P's and the Q's are (see §5.2). Because Q is the inverse of P for the next round, we focus on determining the Q's. In the first part of the attack the non-affine part of the Q's for round 1 to 9 can be recovered. This can be done with a time complexity of 2^{24}. At the same time the non-affine parts of the P's for round 2 to 10 are recovered. In the second step the affine part of the Q's which consist of a linear part and a constant can be recovered. If P and Q are determined completely, the AES round keys can be determined. The total complexity of the attack is $4 \cdot 4 \cdot 2^{24} = 2^{28}$.

If the attacker only knows the type II tables, the attack will not work. The attack works because the type II tables and the type III tables are considered as one entity. In this way the mixing bijections in the middle cancel each other. This implies that the `MixColumns` matrix is no longer obfuscated by a mixing bijection. The `MixColumns` matrix is publicly known, and because of some specific characteristics of this matrix the attack works. If this matrix is obfuscated the attack will not work.

No general statement can be made about the possibility of the existence of a strong white-box implementation. However, the attack shows that the described implementation is not a strong one. More research is needed on the possibilities of constructing better white-box implementations. Nevertheless the described white-box AES implementation is not useless. In the next chapter we look for possibilities to use the AES implementation in such a way that the attack cannot be carried out.

Chapter 6

Using White-Box Cryptography in Practice

6.1 Introduction

In this chapter we will look for possibilities to use white-box cryptography for AES in a secure way. We discuss an application of white-box cryptography in which we split the set of white-box tables into a dynamic part and a static part in §6.2. The result is that whenever a key needs to be updated, no longer the whole set of tables needs to be updated. In §6.3 different possibilities for using external encodings are described. We discuss what the best possibility is. In §6.4 we discuss the problem of the storage space and in §6.5 we look for possibilities to use less tables to save storage. Finally, in §6.6 we look for possibilities to use white-box cryptography for OMA-DRM.

6.2 Splitting the White-Box Tables

We want to be able to update the key which is hidden in the white-box tables. This can be done by splitting the tables. The part of the tables which is dependent on the key is sent to the client and the other part of the tables which is not dependent on the key is stored on the client's device. When we want to update the key, only part of the tables needs to be sent to the client. Therefore, less data needs to be transmitted. The tables that are sent by the server to the client can be updated and are called the dynamic tables. The tables that are stored on a clients's device cannot be updated and are called the static tables. If an attacker taps the ciphertext plus part of the tables, he is not able to decrypt the ciphertext, because he also needs the other tables. The idea to make part of the tables dynamic was proposed by Paul Gorissen et al. in [13].

The following needs to be considered:

- It is important that each client receives different static tables to ensure that each client uses a unique combination of static and dynamic tables. If this is not ensured, then someone could tap the dynamic tables which were sent to another client and use these dynamic tables in combination with his own static tables to decrypt the content.

- It is important that the static tables cannot be copied. Otherwise a client could publish his static tables and his dynamic tables which together could be used for decrypting content. This can be done by locking the static tables on the device (nodelocking).

However, the question remains which tables need to be transmitted and which tables can be stored.

There are five types of tables: type Ia, II, III, IV, and Ib. The tables that are dependent on the key are the type II and the type Ib tables. We want to be able to update this key, therefore these tables cannot be fixed on the client's device. The least amount of data a server needs to send are the tables which are dependent on the keys and those are the type II and type Ib tables.

There are several possibilities for partitioning the set of tables into a set of dynamic tables and a set of static tables:

1. Dynamic tables: II, Ib (208 KB)
   Static tables: Ia, III, IV(544 KB)
2. Dynamic tables: Ia, II, Ib (272 KB)
   Static tables: III, IV (480 KB)
3. Dynamic tables: II, III, Ib (352 KB)
   Static tables: Ia, IV (400 KB)
4. Dynamic tables: II, IV, Ib (544 KB)
   Static tables: Ia, III (208 KB)
5. Dynamic tables: Ia, II, III, Ib (416 KB)
   Static tables: IV (336 KB)
6. Dynamic tables: II, III, IV, Ib (688 KB)
   Static tables: Ia (64 KB)
7. Dynamic tables: Ia, II, IV, Ib (608 KB)
   Static tables: III (144 KB)
8. Dynamic tables: Ia, II, III, IV, Ib (752 KB)
   Static tables: -

Partition 8 is the original situation in which all the tables are sent to the client. If an attacker has access to all the tables, the attack can be executed. Therefore, it is not recommended to transmit all the tables over the line.

The server wants to send the least amount of data. Therefore, the server only wants to send tables which it wants to update, like the tables which are dependent on the key or the tables which represent the external encodings. Two partitions remain:

- Dynamic tables: II, Ib (208 KB)
  Static tables: Ia, ,III, IV (544 KB)
  (see Figure 6.1)

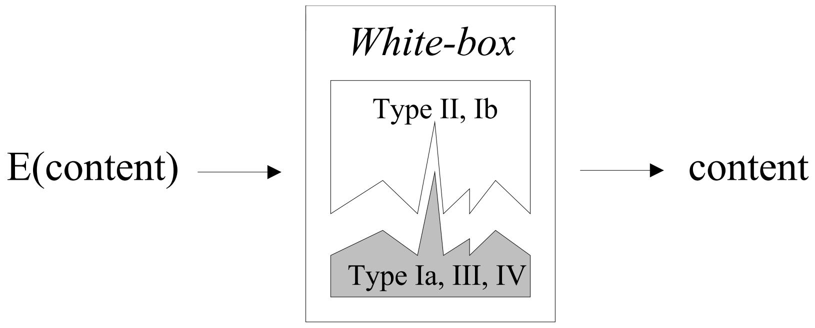


Figure 6.1: One of the four mappings for a single AES round

- Dynamic tables: Ia, II, Ib (272 KB)
  Static tables: III, IV (480 KB)
  (see Figure 6.2)

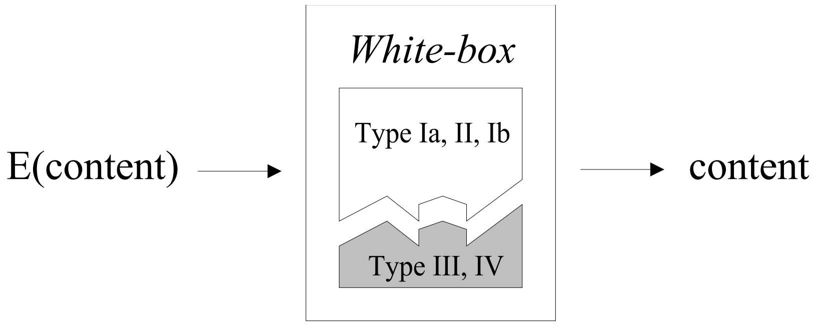


Figure 6.2: One of the four mappings for a single AES round

The static keys can be seen as personalization keys, which are unique for each client. The choice between the two possible partitions depends on the choice for the external encodings. This will be explained in §6.3.

6.3 External Encodings

In §6.2 two possible partitions of the set of tables into a set of dynamic tables and a set of static tables remained. The choice between these two possible partitions depends on the choice for the external encodings which will be discussed in this section.

Suppose a ciphertext C corresponding to a plaintext P is sent to a client who wants to obtain P. The white-box tables which are used to decrypt C are represented by $\boxed{G \cdot AES_d \cdot F^{-1}}$, where G and F^{-1} are the external encodings and AES_d represents the decryption with AES in the white-box implementation. Note, AES_e represents the encryption with AES in the white-box implementation.

There are four possibilities for the external encodings:

1. $C = F \cdot AES_e \cdot G^{-1}(P) \Rightarrow \boxed{G \cdot AES_d \cdot F^{-1}} \rightarrow P$

   In the first case the server sends the ciphertext $C = F \cdot AES_e \cdot G^{-1}(P)$ plus the white-box tables $\boxed{G \cdot AES_d \cdot F^{-1}}$ to the client. The ciphertext serves as input to the white-box tables. The client can use the white-box tables to decrypt the ciphertext to obtain the plaintext P.

   The server will put the information concerning G in the dynamic tables. Moreover, the information concerning F can also be put in the dynamic tables. This implies that the server can send the same ciphertext to each client, or send a different ciphertext to each client by varying F and G. This can be seen as personalization.

   If the server wants F, G^{-1}, and the key to be dynamic, it has to send both the type Ia and the type Ib tables plus the type II tables:

   Dynamic tables: Ia, II, Ib (272 KB)
   Static tables: III, IV (480 KB)

   The advantage of this method is that F and G can be varied. A possible disadvantage of this method is that P comes available to the client, which he can distribute. However, in case P is very large, this may not be convenient. In that case it becomes more interesting to extract the key, because the key can be distributed more easily than a large plaintext. With the key people can decrypt the content themselves.

2. $C = F \cdot AES_e(P) \Rightarrow \boxed{G \cdot AES_d \cdot F^{-1}} \rightarrow G(P) \Rightarrow \boxed{\boxed{G^{-1} \rightarrow P}}$

   In the second case the server sends the ciphertext $C = F \cdot AES_e(P)$ plus the white-box tables $\boxed{G \cdot AES_d \cdot F^{-1}}$ to the client. The ciphertext serves as input to the white-box tables. The client can use the white-box tables to decrypt the ciphertext to obtain $G(P)$. On the client's device G^{-1} is stored in a renderer which is assumed to be non-accessible. The decryption of $G(P)$ is done in the renderer and P will never be exposed.

   The server can put the information concerning F in the dynamic tables. This implies that the server can send the same ciphertext to each client, or send a different ciphertext to each client by varying F for each client. G^{-1} is stored on a client's renderer and is fixed. G^{-1} should be different for each client. Otherwise a client could publish $G(P)$ on the internet. Someone who downloads $G(P)$ is able to calculate $G^{-1} \circ G(P) = P$ by using his own G^{-1}. This implies that the server needs to know for each client what kind of G he has.

   If the server wants F and the key to be dynamic, it has to send both the type Ia and the type Ib tables, plus the type II tables. Although G is fixed, the tables which represent G need to be updated because those tables also contain the key.

   Dynamic tables: Ia, II, Ib (272KB)
   Static tables: III, IV (480 KB)

   The advantage of this method is that F can be varied. The disadvantage is the assumption that the renderer is completely secure. This assumption does not fit in the white-box attack model where we assume that the implementation is completely visible to the attacker.

3. $C = AES_e \cdot G^{-1}(P) \Rightarrow \boxed{F} \rightarrow F \cdot AES_e \cdot G^{-1}(P) \Rightarrow \boxed{G \cdot AES_d \cdot F^{-1}} \rightarrow P$

   In the third method the server sends the ciphertext $C = AES_e \cdot G^{-1}(P)$ plus the white-box tables $\boxed{G \cdot AES_d \cdot F^{-1}}$ to the client. The client encrypts the ciphertext with a stored F. The new ciphertext serves as input to the white-box tables. The client can use the white-box tables to decrypt the ciphertext to obtain the plaintext P.

   If the server wants G and the key to be dynamic, it has to send the type Ib tables plus the type II tables:

   Dynamic tables: II, Ib (208 KB)
   Static tables: Ia, III, IV (544 KB)

The advantage of this method is that the server has to send less data. However, it is not a good idea to use the local encryption because it weakens the security.

Suppose F is hidden well, and we have 128 linearly independent ciphertexts $C_1, \ldots C_{128}$ such that $F \cdot C_i$ is known for each i. Then, we can also determine $F \cdot e_i$ for each $i \in \{1, \ldots, 128\}$ with e_i the i-th unit vector. However, $F \cdot e_i$ is the i-th column of F which implies that F can be determined. Thus, if local encryption is used, then it is not safe to assume that F can be kept secret. If the client knows F, then also the mixing bijections of the type Ia tables and the type II tables can be determined. This will probably make it easier to extract the keys from the tables. More research is needed to determine if knowledge of the bijections will make it easier to extract the keys from the tables.

4. $C = AES_e(P) \Rightarrow \boxed{F} \rightarrow F \cdot AES_e(P) \Rightarrow \boxed{G \cdot AES_d \cdot F^{-1}} \rightarrow G(P) \Rightarrow \boxed{\boxed{G^{-1} \rightarrow P}}$

   In the fourth method the server sends the ciphertext $C = AES_e(P)$ plus the white-box tables $\boxed{G \cdot AES_d \cdot F^{-1}}$ to the client. The client encrypts the ciphertext with a stored F. The new ciphertext serves as input to the white-box tables. The client can use the white-box tables to decrypt the ciphertext to get $G(P)$. On the client's device G^{-1} is stored on a renderer which is assumed to be non-accessible. The decryption of $G(P)$ is done in the renderer and P will never be exposed.

   F and G are fixed. Therefore, the server only has to send the type Ib tables plus the type II tables to update the key:

   Dynamic tables: II, Ib (208 KB)
   Static tables: Ia, ,III, IV (544 KB)

   The advantage of this method is that the server has to send less data. However, it is not a good idea to use the local encryption because it weakens the security (see method three). Another advantage is that the sever sends the same ciphertext $C = AES_e(P)$ to each client. This is also a disadvantage because once an attacker has found the AES key, he could publish it and everybody could use that key to decrypt the ciphertext. In the other methods the ciphertext is personalized for each user which prevents this from happening.

Method one and two are the best methods and three are four are weaker variants which should not be used. The choice between the first and the second method depends on the assumption of security. If a secure renderer can exist the second method is better because P is never available to the client. However, the idea of a secure renderer contradicts the idea of a white-box attack model which we use throughout this document. Thus, using the first method is recommended.

6.4 Storage Problems

The use of white-box cryptography can cause some storage problems because the size of the implementation can be too large. The total size of the tables might be too large to send or to store on a mobile phone. On a mobile phone, each time DRM content is accessed the content needs to be decrypted with the AES key. If white-box cryptography is used, then each time content is accessed, the white-box tables will be used. In case of one key this does not have to be a problem, but when a large amount of content is stored in encrypted form, they can cost too much memory space. The static part of the tables is the same for each piece of content and costs 480 KB of memory space, whereas the dynamic part is different for each piece of content and costs 272 KB of memory space. For example, if we use white-box cryptography for K_{CEK}, for each new content an extra 272 KB needs to be stored on the device.

If the part of the memory space taken by the tables is relatively small compared to the part of the memory space taken by the content (think of broadcasting a movie), then this is not a problem. Otherwise, it seems best to use white-box cryptography only for keys that remain constant over a longer period of time.

6.5 Optimizing the Total Size of the Tables:

Because the total size of the tables might be too large to send or to store on a particular device, we look for possibilities to decrease the total size of the tables.

It is possible to use less tables, for example by using some tables more than once. However, this weakens the security. For each type of table we will look for possibilities to reduce the total size the tables:

- Type Ia: The external encoding is represented by a 128×128 matrix U^{-1}. This matrix is invertible and rows $8i + 1$ to $8i + 8$ form a block of the external encoding for $i \in 0, \ldots, 15$. Each of these blocks is represented by a type Ia table. U^{-1} is invertible and thus the 32 blocks are different. This implies that no two type Ia tables coincide. Thus we cannot store less type Ia tables.
- Type II: The type II tables are dependent on the key and they are all different. Thus we cannot store less type II tables.
- Type III: The type III tables represent mixing bijections which are chosen randomly for each table. However, it is also possible to keep the mixing bijections the same inside a round. Another radical possibility is use only one 32×32 mixing bijection and one 8×8 mixing bijection for all rounds. The 32×32 bijection which is the result of the product of the mixing bijections can be represented by four tables. Thus it suffices to use only four type III tables. The total size is: $4 \times 2^8 \times 32 = 32768$ bits $= 4$ KB.

- Type IV: We can choose to keep the type IV tables the same. Then only one type IV table is needed. $2^8 \times 4 = 1024$ bits.
- Type Ib: The type Ib tables are dependent on the key and they are all different. Thus we cannot store less type Ib tables.

As shown above, only the number of the type III and type IV tables can be reduced. Thus the size of the dynamic tables stays unaffected. Reducing the size of the static type III and type IV tables as indicated, implies that we can reduce the total size of the static tables from 480 KB to $32768 + 1024 = 33792$ bits $= 4224$ Bytes. However, if a lot of encrypted content is transmitted, then the amount of memory space taken by the static tables will be relatively small compared to the amount of memory space taken by the dynamic tables. To store the static tables we need 4224 bytes, whereas for each key we want to store we need an extra 272 KB for the dynamic tables. In that case, reducing the size of the static tables will not be significant.

If an attacker knows that only four type III tables and one type IV tables are used, he can try to determine the type IV table with a brute force attack. There are $2^{84.6}$ possible construction for a type IV table. However, with knowledge of the type IV table the attacker cannot do anything. The type III tables are more interesting. However, there are $2^{369.4}$ possible construction for a type III table which is too much for a brute force attack.

6.6 Using White-Box Cryptography in the OMA-DRM Context

In a white-box attack context, an attacker has full access to the decryption software and has control over the execution environment. This enables him to find the keys, which he can distribute over the internet. In this section we look for possibilities for applying white-box cryptography in the context of OMA-DRM to protect keys. For more details on the keys see chapter 2. The following keys will be considered:

- K_{CEK} : The content is encrypted with K_{CEK}. If someone knows K_{CEK}, he is able to decrypt the content. He could also publish K_{CEK} to enable other people to decrypt the content. Therefore, it is important that K_{CEK} cannot easily be extracted. Hiding K_{CEK} in white-box tables is the best way to prevent unauthorized people for decrypting the content. K_{CEK} can be hidden in white-box tables as follows:

  1. The first possibility is to send WB$_{K_{CEK}}$ to the client instead of $C =$ AES_WRAP(K_{REK}, K_{CEK}), where WB$_{K_{CEK}}$ are the dynamic white-box tables in which K_{CEK} is hidden. K_{REK} was used for encrypting K_{CEK}, but now K_{CEK} is protected by the white-box tables. Therefore, K_{REK} is no longer needed, and can be removed from C^*.

2. Another possibility is to send $C =$AES_WRAP$(K_{REK}$,WB$_{K_{CEK}})$ instead of $C =$ AES_WRAP(K_{REK}, K_{CEK}), where WB$_{K_{CEK}}$ are the dynamic white-box tables in which K_{CEK} is hidden. Now K_{REK} is needed to obtain the white-box tables. The dynamic white-box tables together with the static white-box tables are needed to decrypt the content.

3. It is also possible to send $C =$AES_WRAP$(K_{REK}$,AES$^K(K_{CEK})$ instead of $C =$ AES_WRAP(K_{REK}, K_{CEK}), where AES$^K(K_{CEK})$ is the AES-encryption of K_{CEK} with a key K which is unique for each client. We also have to send WB$_K$ which are the dynamic white-box tables in which the key K is hidden. In this way K_{CEK} can be stored in encrypted form with key K.

4. Another possibility is to store K_{CEK} in encrypted form instead of decrypted form on the device. We encrypt K_{CEK} with a unique key K. This key K is stored in white-box tables and never visible. Whenever we need K_{CEK}, we can decrypt the encrypted K_{CEK} with the white-box tables. After that, we can decrypt the content. An advantage is that we only have to store one set of white-box tables. Another advantage is that the specifications of OMA-DRM are still respected, because there are no specifications which state how keys need to be stored.

- K_{REK} : K_{REK} is used for encrypting K_{CEK}. If somebody knows K_{REK} he is able to calculate K_{CEK}. K_{REK} could also be hidden in white-box tables. However, with these tables we are able to obtain K_{CEK}, and distribute K_{CEK}. Therefore, it is better to hide K_{CEK} in white-box tables than K_{REK}. For a user who wants to have illegal content it is more convenient to obtain K_{CEK} than K_{REK} because this needs one calculation step less.

- K_{MAC} : K_{MAC} is used for integrity checking. It is not used for encryption. Therefore, it is not necessary to apply white-box cryptography to K_{MAC}.

- K_D : If somebody knows K_D he is able to calculate K_{REK}, and then also K_{CEK}. K_D could also be hidden in white-box tables, and these tables could be used for obtaining K_{REK}, K_{CEK}, and the content. K_{CEK} becomes visible, and can be distributed. Therefore, it is not enough protecting only K_D.

- KEK : KEK cannot be protected with white-box cryptography because KEK can be calculated with KDF.

- $K_{Private}$: Currently, $K_{Private}$ is stored on the client's device by the manufacturer of the device and cannot be updated. Suppose, it would be possible to update $K_{Private}$. This can be accomplished if each client has a unique AES-key which is hidden in white-box tables and stored on the device. The server is able to update $K_{Private}$ by sending the new private key, which is encrypted with the client's unique AES-key. The client can use the white-box tables for decryption (Figure. 6.3).

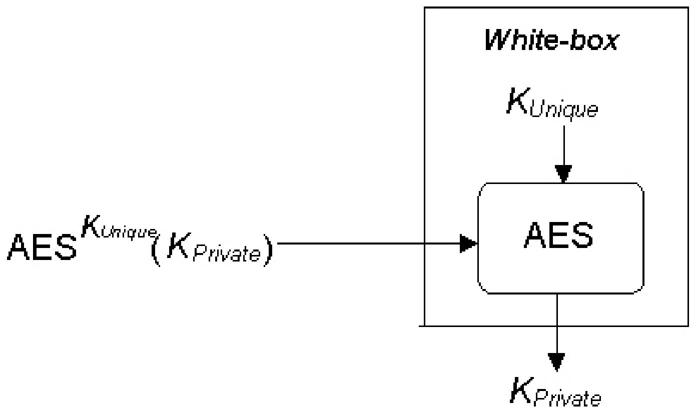


Figure 6.3: Updating $K_{Private}$

> The difference with the original situation is that $K_{Private}$ can be updated. However, the unique AES-key is fixed and hidden in the tables. Once the unique AES-key is found, the private key $K_{Private}$ can be calculated. The problem changes from protecting $K_{Private}$ to protecting the unique AES-key. The static white-box tables are the most important secrets now. The advantage is that this AES-key is hidden, whereas in the original case, $K_{Private}$ was not hidden. A disadvantage is that the server needs to send more data.

Summarizing, we found five possibilities for hiding keys: four possibilities for hiding K_{CEK} and one possibility for hiding $K_{Private}$. However, not each possibility does fit in the OMA-DRM specifications.

The first two possibilities for hiding K_{CEK} cost too much storage, because for each piece of content new white-box tables need to be stored. The possibilities do also not fit in the OMA-DRM specifications. If K_{CEK} is hidden as described in the third and the fourth possibility, then only one set of white-box tables needs to be stored which contains the unique key K. The storage space is no longer a problem, but the third possibility does not fit in the OMA-DRM specifications. Therefore, we recommend the fourth possibility. However, the slowdown can still be a problem. In the fifth possibility where $K_{Private}$ is updated, we only have to store one set of white-box tables which hide K_{Unique}. Therefore, we do not have a storage problem, and the slowdown is also not a problem because $K_{Private}$ does not need to be updated regularly.

6.7 Conclusion

In this chapter we looked for possibilities to use white-box cryptography for AES in a secure way. We recommended to split the set of white-box tables into a static part and a dynamic part. In this way each client has a unique set of static tables which can only be used in combination with a unique set of dynamic tables. The security also increases by sending a different ciphertext to each client.

We suggested different possibilities for applying white-box cryptography for OMA-DRM. Because of the total size of the tables and the slowdown we recommended using white-box cryptography only for keys which are fixed over a longer period of time. For example, white-box cryptography can be used to update the private key. White-box cryptography can also be used to store keys on the client's device.

Overall, we can say that white-box cryptography ensures that the keys are no longer visible. Nevertheless, we are still able to publish the decrypted content. For example we can put a lot of effort in hiding a key which can be used to decrypt an encrypted ringtone, but as soon as the ringtone is decrypted on our mobile phone we could tap it and distribute it on the internet. The same can be said in the case of broadcasting an encrypted movie or a soccer match via a satellite. However, for instance in case of a soccer match, people want to see it live. In this situation people would be more interested in obtaining the key. If they have the key, they could tap the encrypted soccer match and use the key for decryption and watch the soccer match live. In this case it is more valuable to have the key, which implies that it is very important to protect the key securely by for example white-box cryptography.

Chapter 7

Conclusion

The aim of this Master's Thesis was to analyze how digital content for mobile phones can be protected in an effective way in a white-box attack context. We started by analyzing several software protection techniques and looked for possibilities to apply these techniques in the context of OMA-DRM. From these techniques we chose to focus on a relatively new technique: white-box cryptography. In the white-box attack context, the attacker has total visibility into software implementation and execution. To prevent an attacker from finding secret keys, the keys can be hidden in the implementation with white-box cryptography. The result is a functionally equivalent program in which the key is no longer visible. However, white-box cryptography increases the amount of storage space for the white-box tables, and it causes a performance slowdown.

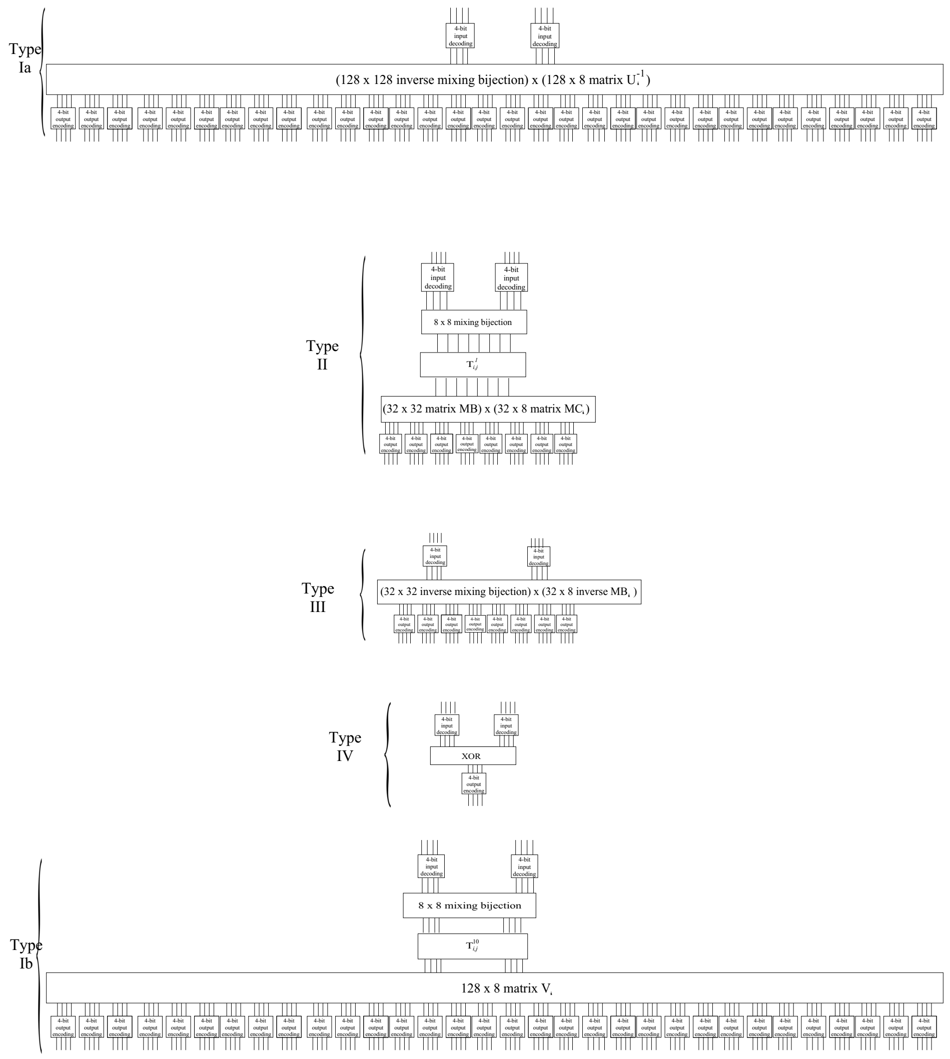
We discussed two drawbacks of white-box cryptography. The first drawback is that whenever the key needs to be updated, the whole set of white-box tables needs to be updated too. We solved this problem by splitting the set of tables into a dynamic part and a static part. Each client has a unique set of static tables which can only be used in combination with a unique set of dynamic tables which are sent to him. Because only the dynamic tables are dependent on the key, the server only has to update the dynamic tables when it wants to update the key. This is also a way to obtain software diversity, because each client needs a unique combination of static and dynamic tables for decryption. A second drawback is that the whole white-box implementation can be used as a key. If an attacker knows to the complete white-box implementation, he can use the white-box tables to decrypt the content. Therefore, it is important that the static tables cannot be copied. This can be done by locking the static tables on the hardware (nodelocking). More research is needed on the possibility of locking the static tables on hardware.

After we started to analyze an implementation on white-box cryptography for AES, an attack on this implementation was published. We analyzed the attack and we looked at its implications. The attack can be carried out by an attacker if he has access to all the white-box tables. Therefore, it is important that an attacker does not have access to all the white-box tables. The attack enables the attacker to find the hidden AES keys. In the original situation proposed in [8] the whole set of tables is transmitted to the client whenever a key needs to be updated. An attacker can eavesdrop the transmitted tables and extract the keys. In the new situation the set of tables is not transmitted totally. Therefore, the positive part is that in the new situation an attacker cannot perform the attack by just eavesdropping on the communication line between the server and the client. For the static tables the attacker has to look on the device. More research is needed on the possibility of constructing a white-box implementation for which the attack does not work.

Finally, we looked for applications of white-box cryptography in the context of OMA-DRM. Because of the total size of the tables and the slowdown we recommended using white-box cryptography only for keys which are fixed over a longer period of time. For example, white-box cryptography can be used to update the private key. More research is needed on the possibility of a white-box implementation for hiding RSA keys. White-box cryptography can also be used to store keys on the client's device. In the future, the size of the tables and the slowdown, might no longer be a problem. Mobile phones will be able to store more data, and process information faster. In that case, white-box cryptography can also be used effectively for storing keys which change regularly, like the content encryption key.

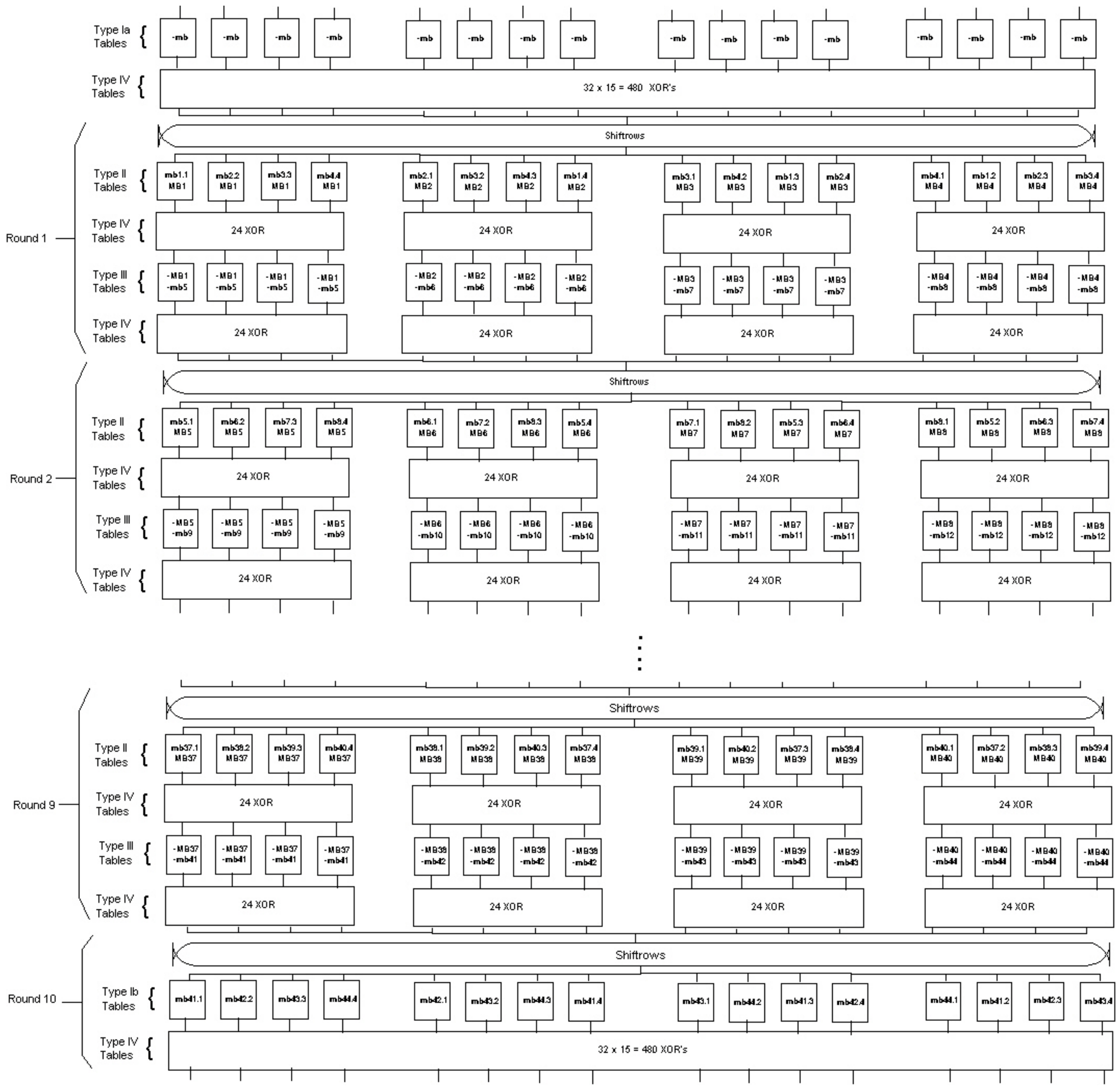
Appendix A

Overview of the White-Box Tables



Appendix B

Overview of Connecting White-Box Tables



where

$$-mb = mb^{-1} = \begin{pmatrix} mb_{1.1}^{-1} &&&&&&&&&&&&&&& \\ & mb_{1.2}^{-1} &&&&&&&&&&&&&& \\ && mb_{1.3}^{-1} &&&&&&&&&&&&& \\ &&& mb_{1.4}^{-1} &&&&&&&&&&&& \\ &&&& mb_{2.1}^{-1} &&&&&&&&&&& \\ &&&&& mb_{2.2}^{-1} &&&&&&&&&& \\ &&&&&& mb_{2.3}^{-1} &&&&&&&&& \\ &&&&&&& mb_{2.4}^{-1} &&&&&&&& \\ &&&&&&&& mb_{3.1}^{-1} &&&&&&& \\ &&&&&&&&& mb_{3.2}^{-1} &&&&&& \\ &&&&&&&&&& mb_{3.3}^{-1} &&&&& \\ &&&&&&&&&&& mb_{3.4}^{-1} &&&& \\ &&&&&&&&&&&& mb_{4.1}^{-1} &&& \\ &&&&&&&&&&&&& mb_{4.2}^{-1} && \\ &&&&&&&&&&&&&& mb_{4.3}^{-1} & \\ &&&&&&&&&&&&&&& mb_{4.4}^{-1} \end{pmatrix}$$

Bibliography

[1] D. Aucsmith, *Tamper Resistant Software and Implementation*, Proc. 1st International Information Hiding Workshop (IHW), Cambridge, U.K. 1996, Springer LNCS 1174, pp. 317-333 (1997).

[2] B. Barak, *Can We Obfusacate Programs?*, http://www.math.ias.edu/ boaz/Papers/obf_ informal.html.

[3] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. Vadhan and K. Yang, *On the (Im)possibility of Obfuscating Programs*,pp 1-18, Advances in Cryptology - Crypto 2001, Springer LNCS 2139 (2001)

[4] O. Billet, H. Gilbert, C. Ech-Chatbi, *Cryptanalysis of a White-box AES Implementation*, SAC 2004.

[5] H. Chang, M. Atallah, *Protecting Software Code by Guards*, Proc. 1st ACM Workshop on Digital Management (DRM 2001), Springer LNCS 2320, pp.160-175 (2002).

[6] Y. Chen, R. Venkatesan, M. Cary, R. Pang, S. Sinha, M. Jacubowski, *Oblivious Hashing: A stealthy Software Integrity Verification Primitive*, Proc. 5st Information Hiding Workshop (IHW), Netherlands (October 2002), Springer LNCS 2578, pp.400-414.

[7] S. Chow, P. Eisen, H. Johnson, P.C. van Oorschot, *A White-Box DES implementation for DRM Applications*, pp. 1-15, Proceedings of DRM 2002 - 2nd ACM Workshop on Digital Rights Management (DRM 2002), Springer LNCS 2696 (2003).

[8] S. Chow, P. Eisen, H. Johnson, P.C. van Oorschot, *White-Box Cryptography and an AES implementation*, pp. 250-270, Proceedings of the Ninth Workshop on Selected Areas in Cryptography (SAC 2002), Springer LNCS 2595 (2003).

[9] F. Cohen, *Operating System Protection Through Program Evolution*, Computers and Security 12(6), 1 Oct. 1993, pp. 565-584.

[10] C. Collberg, C. Thomborson, and D. Low. *A Taxonomy of Obfusacting Transformations*. Technical Report 148, Department of Computer Science, University of Auckland, July 1997.

[11] J. Daemen, V. Rijmen, *AES Proposal: Rijndael*, http://csrc.nist.gov/encryption/aes/rijndael/ Rijndael.pdf, 1999.

[12] S. Forest, A. Somayaji, D. H. Ackley, *Building Diverse Computer Systems*, pp.67-72, Proc. 6th Workshop on Hot Topics in Operating Systems, IEEE Computer Society Press, 1997.

[13] P. Gorissen, J. Trescher, *Key Distribution in Unsafe Evironments*, Philips Research Laboratories Eindhoven, to be published.

[14] B. Horne, L. Matheson, C. Sheehan, R. Tarjan, *Dynamic Self-Chacking Techniques for Improved Tamper Resistance*, Proc, 1st ACM Workshop on Digital Rights Management (DRM2001), Springer LNCS 2320, pp.141-159 (2002).

[15] National Institute of Standards and Technology (NIST). *AES Key Wrap Specification*, November 2001. Available at csrc.nist.gov/encryption/kms/key-wrap.pdf

[16] P.C. van Oorschot, *Revisiting Software Protection*, In Proc. of 6th International Information Security Conference (ISC 2003), pages 1-13. Springer-Verlag LNCS 2851, 2003. Bristol, UK, October 2003.

[17] Open Mobile Alliance, *DRM Specification V2.0*, Open Mobile Alliance Ltd, 2004, La Jolla (CA), USA.

[18] C. Wang, *A security Architecture for Sirvivability Mechanisms*, Ph. D. thesis, University of Virginia (Oct. 2000).

[19] J. Xiao, Y. Zhou, *Generating Large Non-Singular Matrices over an Arbitrary Field with Blocks of Full Rank*, Cryptology ePrint Archive (http://eprint.iacr.org), no. 2002/096.

[20] Draft ANSI X9.44, *Public Key Cryptography for the Financial Services Industry - Key Establishment Using Integer Factorization Cryptography*, Draft 6, 2003.