Universitat Autònoma
de Barcelona

UNIVERSITAT
ROVIRA I VIRGILI


Universitat de les
Illes Balears

Trabajo Final de Máster

Automation of White-Box Cryptography attacks in Android applications

Máster Interuniversitario en Seguridad de las Tecnologías de la Información y de las Comunicaciones

Author: Víctor Sánchez Ballabriga

Sponsor company: Brightsight B.V.
Company mentor: Adrián García Ramirez
Consultant: Arnau Vives Guasch
Professor: Victor Garcia Font

UOC December 2018

Automation of White-Box Cryptography attacks in Android applications

ABSTRACT

Cryptography is increasingly deployed in applications that are executed on open devices (such as PCs, tablets or smartphones). The open nature of these systems makes the software extremely vulnerable to attacks, since the attacker has complete control over the execution platform and the software implementation itself. This means that an attacker can easily analyze the binary code of the application, and the corresponding memory pages during execution.

Therefore, White-Box Cryptography (WBC) is an essential technology in any software protection strategy. In practice, WBC is a cryptographic algorithm, usually simmetric, that embeds the key in it and obfuscates the process to make it more difficult to reverse.

This technology allows to perform cryptographic operations without revealing any portion of confidential information such as the cryptographic key. Without this, attackers could easily grab secret keys from the binary implementation, from memory, or intercept information that would lead to disclosure at execution time.

Parallely, new attack paths appeared with this technology, attacks that in the past were used to attack cryptography implemented in hardware, such as Differential Power Analysis (DPA) or Differential Fault Analysis (DFA) found their equivalents in software. This equivalent attacks are based on the analysis of the memory accesses or the values stored in the registers at every cycle of the processor. In order to retrieve these traces, dynamic binary instrumentation and emulation of the binaries are the best approaches.

However, in order to protect WBC implemented in software, usually other security countermeasures are applied such as code obfuscation, which can make unfeasible the task to find the functions that are executing the encryption/decryption tasks, making unfeasible to trace them to retrieve valid traces for a successful attack.

This project has two main objectives. The first one is to facilitate the search of the WBC functions in obfuscated binaries, making it easier for the security analysts that want to test the strength of the different WBC implementations in real products. Researching and going deep into the different techniques that could be used to identify the fingerprint that this type of cryptography may leave in a binary which is using it, its identification might become in a semiautomatic task.

The second objective of the project is to implement an ARM processor emulator with capabilities to collect the necessary traces for the attacks mentioned above, generating also the necessary faults in the execution when needed.

Keywords: White-Box Cryptography, Android, Attack automation

RESUMEN

La criptografía es cada vez más usada en aplicaciones que se ejecutan en dispositivos que no fueron creados para ser seguros como meta principal (como ordenadores personales, tablets o smartphones). La naturaleza de estos sistemas hace que el software sea extremadamente vulnerable a los ataques, ya que el atacante tiene un control completo sobre la plataforma de ejecución y la implementación del software en sí. Esto significa que un atacante puede analizar fácilmente el código de la aplicación así como su memoria correspondiente durante la ejecución de estas.

Por este motivo White-Box Cryptography (WBC) es una tecnología esencial en cualquier estrategia de protección de software. En la práctica, WBC es un algoritmo criptográfico, generalmente simétrico, que diluye la clave en él y ofusca el proceso para dificultar su entendimiento por parte de potenciales atacantes.

Esta tecnología permite realizar operaciones criptográficas sin revelar ninguna parte de la información confidencial, como puede ser la clave criptográfica. Sin esta tecnología los atacantes podrían recuperar fácilmente las claves secretas del binario, de la memoria o interceptarla durante su uso en ejecución.

Paralelamente, las nuevas rutas de ataque aparecieron con esta tecnología, los ataques que en el pasado se usaban para atacar criptografía implementada en hardware, como el Differential Power Analysis (DPA) o el Differential Fault Analysis (DFA) encontraron sus equivalentes en software. Estos ataques equivalentes se basan en el análisis de los accesos de memoria o los valores almacenados en los registros en cada ciclo del procesador. Para recuperar esta información, la instrumentación dinámica de los binarios y la emulación son los mejores enfoques.

Sin embargo, para proteger la WBC en el software, generalmente se aplican otras medidas de seguridad sobre ella, como la ofuscación de código, que puede hacer casi imposible la tarea de encontrar las funciones que ejecutan las tareas de cifrado/descifrado, lo que dificulta rastrearlas para recuperar la información necesaria para un ataque exitoso.

Este proyecto tiene dos objetivos principales que pretenden, como producto final, por un lado facilitar la búsqueda de funciones que implementan criptografía de tipo White-Box, así como su posterior instrumentación, lo que facilitaría la tarea de los analistas de seguridad que deben probar la robustez de estas implementaciones en productos reales.

Al investigar y profundizar en las diferentes técnicas que podrían usarse para identificar la huella que este tipo de criptografía puede dejar en un binario que lo está utilizando, su identificación podría convertirse en una tarea semiautomática.

Por otro lado, el segundo objetivo de este proyecto es la implementación de un emulador de procesador en arquitectura ARM. Este emulador tiene las capacidades para recopilar las trazas necesarias para los ataques mencionados anteriormente, así como la capacidad de generar faltas en la ejecución cuando sea necesario. Todo esto crea un entorno de ejecución controlado donde los ataques pueden ser llevados a cabo bajo el control y supervisión de un técnico.

Keywords: White-Box Cryptography, Android, Automatización de ataques

Contents

List of Figures

Acronyms

AES Advanced Encryption Standard.

API Application Programming Interface.

APK Android Package Kit.

CCA Correlation Computational Analysis.

CPA Correlation Power Analysis.

CTF Capture The Flag.

DBI Dynamic Binary Instrumentation.

DCA Differential Computational Analysis.

DES Data Encryption Standard.

DFA Differential Fault Analysis.

DPA Differential Power Analysis.

I/O Input/Output.

IP Intellectual Property.

PC Program Counter.

R&D Research & Development.

SP Stack Pointer.

WBC White-Box Cryptography.

Chapter 1

Introduction

1.1 Problem statement

White-Box Cryptography (WBC) is a cryptographic technique that aims to dissolve the secret key of symmetric encryption algorithms inside the code, making more difficult to retrieve it.

The use of White-Box Cryptography (WBC) in software solutions increased considerably during the last years due to their effectiveness to protect the encryption key against attackers in untrusted environments if they are combined with other techniques to protect it.

In high security software, such as the mobile HCE payment applications evaluated in Brightsight, the necessary effort to find, isolate, and trace the execution of the WBC implementation used by the application is commonly higher.

The extra time necessary to test the strength of the WBC implementation and its protections ends in a redistribution of the given penetration testing time, assigning more time to this task than other. When this happens, the security evaluators in charge of the penetration testing of the other components of application have the risk to run out of time without a clear evidence of the possible vulnerabilities present in the product, or may make the necessary evaluation so long, losing competitiveness.

If the necessary time to isolate and trace the WBC in a given software could be reduced, would imply better performance on the evaluation of products with this type of encryption.

1.2 Software-based security evaluations and the role of WBC

The new technological environment and the inclusion of powerful and portable devices such as laptops and smartphones is pushing the development of portable solutions that

are able to work in this type of devices. The idea of the portability and the compatibility of the different types of devices in the market makes the software based solutions perfect to achieve this goal. However, when due to its purpose the software has to manage sensitive assets, a new technology such as White-Box Cryptography (see Section 2.1) has to be developed to protect those assets without the help of trusted hardware.

In order to assess the level of protection that a particular WBC implementation gives to a software solution, penetration test is necessary to check the strength of it in a real attack scenario. Therefore a complete attack tool is necessary to perform the state of the art attacks that could be faced in a real case scenario.

1.3 Motivation

Both the commercial and built in Brightsight tools used in the security evaluations until now can help to the security evaluators to identify and isolate the WBC used by the applications through reverse engineering and dynamic binary instrumentation or emulation. However, these tools require long time of manual effort by the evaluators.

Taking into consideration that the average time assigned for each security evaluation is between two and four weeks of penetration testing effort, simply applying strong obfuscation to the binaries of the application tested can make that the evaluator will not have enough time to reverse them to identify and further attack the WBC. This extra time spent implies that the rest of the specific countermeasures implemented to protect this WBC might not be tested as deep as could be required.

A tool which could automatize the attack process, or at least partially, would help to save time during evaluations, redirecting the effort of the evaluators to other penetration tests that also have to be done, and raising the efficiency of the Brightsight's Software Based Security department.

1.4 Objective

The objective of this project is to design, develop and test a tool or set of tools to automatize as much as possible the process of identification, isolation and attack of WBC implementations integrated in highly secured Android applications, such as mobile payment applications.

The desired final result of the project is a tool that, taking as input the whole APK file or the library which is implementing WBC, would be able to automatize the identification, instrumentation and attack of it, giving as output the secret key.

However, due to the constraints coming from the time assigned to this thesis, the objective of the thesis is the implementation of two separated tools that would be able to achieve this main objective with the minimum human intervention possible, leaving as future work the automation of the steps in between these two tools.

The first tool would be able to return the best candidates to be WBC functions, discovering also their respective parameters. And the second one would emulate the

candidate functions trying to apply Differential Computational Analysis (DCA) and Differential Fault Analysis (DFA) attacks.

1.5 Methodology

During the research part of the project, all the necessary information available for the correct detection of WBC in binaries have been collected and analyzed, gaining knowledge about how it is protected in those binaries, and which type of marks or fingerprint may it leave in them. In parallel, information about Unicorn Engine and how an emulator has to be implemented using it was also collected.

Once the knowledge obtained was considered enough, the detection and emulation tools have been designed, deciding the best approach and the main features that these tools should have.

Then, the development phase started, focusing first on the creation of the binary emulation tool. Once the tool had a minimum level of functionality, it was tested to debug and improve the tool. At the point that the capabilities of the tool were the ones necessary for the objective described in Section 1.4, it was tested first using binaries without strong protections, and jumping to more protected ones progressively, finalizing in the first release of the tool.

Due to time constraints it was not possible to finalize the improvement of the WBC detection tool. However, the design phase has been finished and the development is already planned as future work (see Chapter 5 for more detailed explanation).

The last part of the project has been the redaction of the final report, which had several revisions to obtain a good quality on it.

Chapter 2

Background

In this chapter, basic concepts for the understanding of this thesis report are defined. It is intended to give an introduction to the different characteristics that will be addressed in later chapters, such as White-Box Cryptography basics, some of the available attacks against it and emulation of binary code.

2.1 White-Box Cryptography

The main idea behind White-Box Cryptography is to embed both the static key (in the form of data but also in the form of code) and random data (instantiated at compilation time) in a composition from which it is hard to derive the original key.

One of the main design principles in cryptography is the Kerckhoffs' principle [Pet11], which states that a cryptosystem should be secure even if everything about the system, except for the key, is public knowledge. White-box Cryptography aims to withstand similar public scrutiny: not only does the attacker have full access to the implementation, he also knows which algorithm is implemented and what white-box protection techniques are applied; the security relies on the confidentiality of the secret key and the random data.

The first white-box implementations were presented by Chow et al. [CEJvO03a] in 2002 on the DES and the AES respectively. Their white-box techniques transform a cipher into a series of key-dependent lookup tables. The secret key is hard-coded into the lookup tables and protected by randomization techniques that are applied. One such method that is deployed is the injection of random annihilating encodings which are merged together with the lookup tables such that the lookup tables and the data flow in between is randomized, while retaining the overall semantic functionality of the implementation.

2.2 Differential Computational Analysis Attack

DCA attack [BHMT15] is the software derived version of the Differential Power Analysis (DPA) attack that is used in hardware cryptography to attack smartcards, chips, etc.

Since the late 1990s it is publicly known that the statistical analysis of a power trace obtained when executing a cryptographic primitive might correlate to, and hence reveal information about the secret key material used by the algorithm [KJJ99].

Later on, it was also discovered that the electromagnetic emanations of the processor during processing may also leak this information.

Using the same principle, tracing I/O memory instructions during the encryption or decryption of different input values, it is possible to obtain the same type of correlations and therefore retrieve the key embedded in the software WBC implementation.

For this type of attack it is necessary to obtain different executions of the cryptographic algorithm with different inputs in order to find the correlations between the different input/output pairs.

2.3 Differential Fault Analysis Attack

This type of attack [TH16] also comes from a hardware background, and it is based in the induction of faults during the execution of a cryptographic algorithms to reveal their internal states.

The statistical analysis of an original trace (execution trace of a successful encryption of a given input) together with traces obtained using the same input and injecting faults during its execution can give as result the secret key of the software WBC implementation.

In the hardware version of this attack the faults can be injected through different techniques such as power glitches or using a laser [PBR17]. In the software version the faults are injected using dynamic binary instrumentation, which requires control over the execution environment.

For this type of attack it is required to execute the cryptographic algorithm several times with the same input in order to obtain information leakages when faults at different points of the execution are injected.

2.4 Binary emulation

Binary emulation is the execution of a given binary, or a piece of it, in an emulated environment, which can imitate the whole real environment, such as QEMU [Bel05], or just a processor, such as Unicorn Engine [AQHV15] in a host machine.

The created emulated environment give access to every level of it, from the processor and the register at every moment of the execution, to the binary and the memory that is allocating it and is used by it.

Chapter 3

State of the art

White-Box Cryptography technology is a solution to hide the secret keys in the cryptographic algorithm that, as explained in [VC17], is becoming more used everyday due to the necessity to protect pure software solutions against external attacks using a controlled environment to retrieve the secret key.

Several papers have been published explaining how WBC works and how to attack isolated implementations of them, such as [AFG+15], [JW16] and [SMdH15]. However, not so many papers talk about the problem that a security analyst may face if he tries to attack a WBC implementation that has other type of countermeasures, such as strong obfuscation, anti-debugging, anti-hooking and anti-emulation.

The first problem that a penetration tester has to face in a highly secured software solution is the identification and isolation of the WBC in the binary, mainly due to the strong obfuscation techniques used to hide it. Some public papers are available talking about this topic, but they are usually focused on traditional cryptography, trying to detect cryptographic primitives typically such as [LGF15] or [GWH11].

Secondly, in case that the WBC implementation is found, the next step in the penetration testing is the tracing and further attack, and for that attacks like DCA or DFA have a proven effectiveness [TH16]. For this purpose is necessary to trace the execution of the WBC, with possibility to instrument the execution in case of DFA. It is possible to find public tools such as the Side Channel Marvels[1], and also commercial tools such as esDynamic[2].

Another option, and is one of the objectives of this thesis, is to create an emulator to trace the execution of the binary. For that different emulation frameworks are available such as Unicorn Engine[3], Valgrind[4] or PIN[5], being the last one only for x86.

This project covers the implementation of a proprietary emulation and tracing tool

[1] https://github.com/SideChannelMarvels
[2] https://www.eshard.com/esdynamic-white-box-crypto-framework/
[3] https://www.unicorn-engine.org/
[4] http://valgrind.org/
[5] https://software.intel.com/en-us/articles/pin-a-dynamic-binary-instrumentation-tool

to perform DCA and DFA attacks, which is something already implemented, but the advantage of an in-house developed tool is the knowledge gained about the topic, the integration with other tools of the company and the flexibility to focus on the most used features. However, it also aims to find a reliable way to find WBC functions in highly secured binaries, which is a topic not really developed publicly until now.

Chapter 4

White-Box Cryptography

This chapter explains the characteristics of White-Box Cryptography technology in detail, pointing which type of marks it leaves when it resides inside of a binary. Additionally, this chapter also aims to point what is the approach that is gonna be used to detect the functions that are used as entry point to the use of this technology, and the functional requirements that the emulator used for the attacks has to fulfill in order to make it usable by the security evaluators that will have to work with it.

4.1 White-Box Cryptography in software based security

As explained in Section 2.1, the objective of White-Box Cryptography is to hide the keys used by a typically symmetric encryption algorithm in the binary code, making the necessary effort to retrieve them much higher.

It is difficult to specify how a WBC is implemented due to the abstractness of the idea. Figure 4.1 shows the general concept of this idea, converting a traditional cryptographic algorithm with a static secret key into a White-Box version of it, embedding this algorithm and static key into a complex loop of look-up tables.

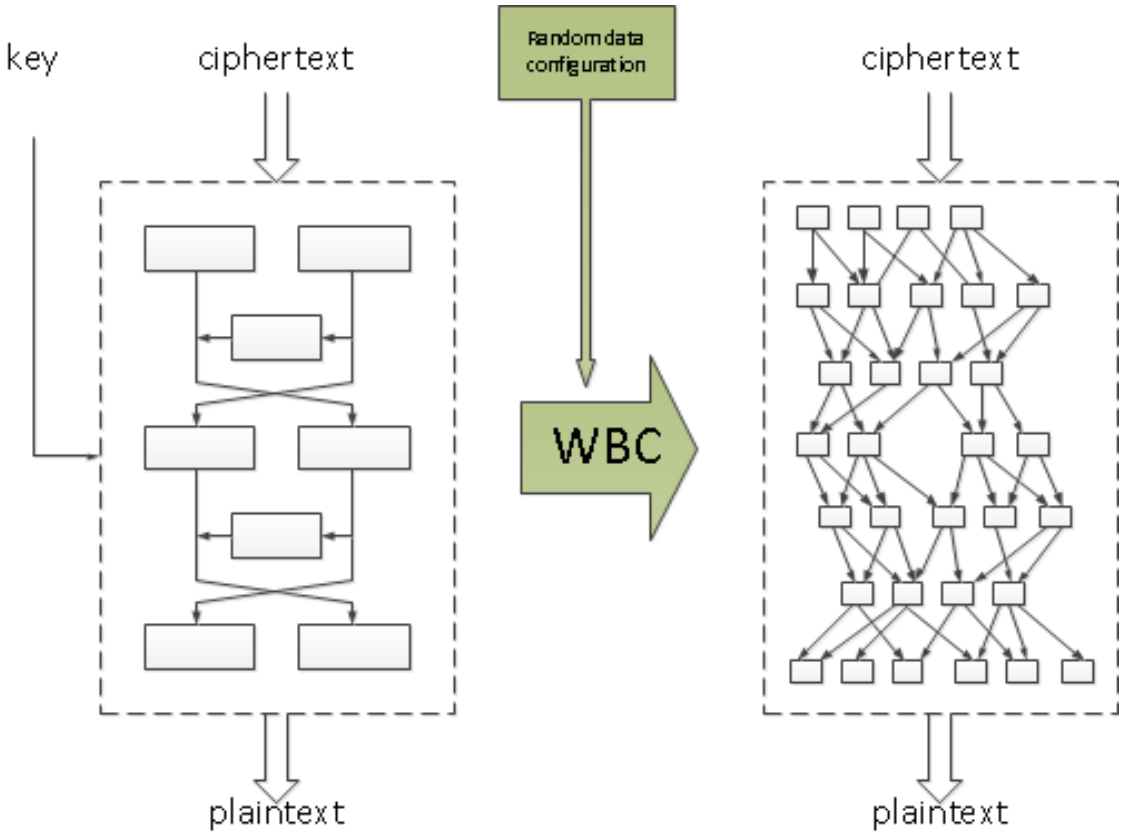

Figure 4.1: Basic idea behind the term White-Box Cryptography [SOF12].

Several implementations of WBC have been published since this new technology started such as Chow-AES [CEJvO03a] or the dozens of implementations that are presented to contests like CHES[1]. However, until now all of them have been broken. It is important to mention that these implementations are academic versions of the algorithm, which means that they are isolated implementations of the WBC, without any other countermeasure.

Pointing to the scenario of highly secured software, typically White-Box Cryptography implementations have other types of countermeasures around them such as code obfuscation or anti-instrumentation. These countermeasures aim to protect the WBC against external attacks.

In these high security scenarios some characteristic features that are linked to the nature of WBC designs may be present in the protected binary. However, preliminary research could not find any paper talking about them, or trying to find WBC with other different techniques.

The experience obtained working with different WBC implementations gives the fol-

[1]https://whibox-contest.github.io/

lowing points of interest about them.

- In Android, they are typically implemented in the native layer of the application in order to improve the efficiency and make more difficult its understanding through reverse engineering.
- The size of the look-up table is huge in comparison with other structures.
- A lot of I/O memory operations are used during encryption/decryption.
- Since the environment where WBC is executed is considered untrusted, implementations are usually monolithic, making them bigger and non dependant of external functions.
- Usually they have a lot of internal jumps, creating complex loops.

4.2 Proposed hypothesis for WBC detection

In order to detect WBC implementations embedded in highly secured software, concretely in Android applications, the features described in Section 4.1 will be used to determine the best candidates to be cryptographic functions inside of the shared library files.

The main idea is to apply these heuristics through each function of a given library, using an appropriate weighing for each one, and determine which functions are the best candidates to be WBC functions. After a reasonable list of potential candidates is found, the evaluator would have to apply his knowledge and skills to determine if one of those candidates is WBC.

A problem of this WBC detection mechanism is the application of these heuristics to hidden functions. The use of inline functions as obfuscation technique might make impossible the identification of a portion of code as a function, and therefore the application of any measurement over it to determine if it is a good candidate of WBC.

The validation of this hypothesis is not trivial due to the complexity of the target binaries. In order to validate the effectiveness of a list of candidates a reverse engineer should reverse and understand each function and only after gaining this knowledge, give feedback about the accuracy of the list, which may be difficult in the time frame of a security evaluation.

4.3 Unicorn Engine emulation framework

Unicorn Engine [AQHV15] is a QEMU[2] based lightweight multi-platform, multi-architecture CPU emulation framework. The fact that Unicorn Engine is a CPU emulator is a useful characteristic for this project, because of the flexibility that it provides

[2]https://www.qemu.org/

to execute code from shared libraries, which do not have `main` function, starting and finishing the execution at any point.

This framework also has other interesting features such as an architecture-neutral API, which might be useful to port the tool to other architectures in the future if ARM is not the main mobile architecture in the future; the high performance due to Just-In-Time compilation technology and the bindings to several languages that offers the possibility to make it non dependant of the host machine architecture.

On the other hand, Unicorn Engine also has some negative features, at least in the context of this project. These cons are the lack of documentation, which force to check the source code trying to find the function(s) useful for each situation; and also the lack of any context such as operative system or minimal environment to support the execution of the binaries might be problematic in certain scenarios.

4.4 Functional requirements for the developed emulator

Due to the nature of Unicorn Engine, a minimal knowledge about the piece of code that will be emulated is necessary to provide the initial memory and registers setup to the developed tool in order to simulate the correct state of them before to start the emulation. Therefore, a user-friendly configuration is required for the tool in order to make the configuration process as smooth as possible, saving with it working time.

On the other hand, and according to the testing standards used in Brightsight, the emulator shall be able to emulate binary code compiled for ARM 32-bit architecture, including also the emulation of some basic features provided by the Android operative system such as `libc` calls or canary checks, giving more flexibility to the tool if the emulated WBC functions contain any external call before or during the encryption/decryption process.

Finally, a mechanism to perform WBC attacks has to be implemented. For this thesis, only DCA and DFA attacks will be developed, leaving other types of attacks as future work. In order to give these capabilities to the emulation tool, two features have to be developed:

- A mechanism to collect clear and meaningful execution traces is necessary for both types of attacks. For this purpose both registers values in every clock cycle or data being read or written from/to memory could be collected. Due to the requirements of DCA attack, memory I/O traces will be collected, leaving the register values as a secondary target for tracing.

- A mechanism to inject faults in the intermediate values that the WBC algorithm manages is necessary for the DFA attack. This means that mechanism to modify the values managed by the WBC at any point of the execution have to be altered at binary level, giving an incorrect final result, which will be call faulty output henceforth.

Chapter 5

Design of WBC detector

This part of the project is focus on the improvement of an IDA Pro script used in Brightsight to detect White-Box Cryptography in Android shared libraries. Therefore, an IDA Pro license is necessary, which is only available in the company. Due to the limited access to IDA Pro and time constraints was not possible to finalize the development of the planned improvements. However, the planned design for it is explained in this chapter.

5.1 Current version of the tool

The current WBC detection tool uses the IDA API to detect every function declared in a specific segment of the library, applying different heuristics to each one in order to identify potential WBC functions. The different measurements used are explained below:

- The function should be monolithic, which means that it does not have calls to external functions such as operative system functions. This measurement uses the idea of the no trust in the operative system in the case of an Android device. This idea points that, since the device is controlled by a non expert user and it might contain malware, a high security application should not use possibly compromised calls to the operative system.
- The size of the function is other metric used. Based on the previous idea and the obfuscated design of a WBC, the function that is implementing it should be much bigger than other simpler functions.
- The memory stack size is also measured. Typically WBC functions abuse of the stack to store intermediate values during the encryption/decryption process.
- The number of I/O memory operations inside a WBC function is typically huge due to its design. Lookup tables, intermediate values and the use of the stack mentioned above force the use of memory, which implies a lot of I/O memory operations.

- The last metric is the number of internal jumps in the function. A lot of different operations and loops are executed during encryption/decryption process due to the design of WBC. Therefore, it may be assumed that a lot of `JUMP` operations (including every variant of it such as `BRANCH`) have to be used. However, in practice it was demonstrated that this heuristic is not always relevant depending on the type of obfuscation techniques applied on top of the WBC implementation. The code flattening technique can remove a lot of jumps from the binary and create huge decision trees.

All of these metrics are controlled in the script via thresholds. Since every highly secured binary and therefore every WBC implementation is different in practice, the thresholds have to be adjusted for each binary. Since it is not possible to gain knowledge about a binary without reverse engineering, and the objective of this detection tool is to save reverse engineering effort, the thresholds are manipulated by try-error methodology. The correct adjustment of the thresholds should give a list of good candidates to be WBC functions, reducing the subsequent reverse engineering effort only to the functions listed.

5.2 Addition of inline functions detection

The focus of this project about the WBC detection tool is the addition of inline detection functions. As explained in Section 5.1, the current version of the tool iterates over the different functions of a given code segment and applies the explained heuristics on each one. However, IDA Pro is only able to detect functions that are declared as functions in the binary.

The purpose of an inline function is to remove the function declaration in the binary during compilation by inserting the function code at the address of each call. This feature of C/C++ is used by some obfuscation tools to make more difficult the identification of certain sensitive functions in a binary. In case of the detection tool, it was demonstrated that this obfuscation technique is enough to spoil the results, hiding the inline functions and therefore the WBC.

After research about this topic, it was seen that the automatic detection of inline functions is difficult due to the nature of the problem. Some interesting approaches have been taken by other researches such as the use of semantic flow graphs [AWD16], a combination of static and dynamic analysis [CMJMS10] or symbolic execution to generate semantic information and learn the function recognition model through machine learning techniques [WWW17].

In the other hand, an interesting IDA Pro script has been found. This plugin is called GraphSlick [RPB13]. This plugin aims to automatize the detection of inlined functions. According to its documentation [RPB13] the plugin offers the following features:

- Automatically detection of inlined functions.
- Matching of equivalent inlined functions.

- Simplified visualization and interaction in IDA Pro.
- Rewriting of the binary to outline inlined functions.

Figure 5.1 obtained form the plugin documentation shows how GraphSlick shows in the GS panel 6 basic blocks detected per inlined function call, grouping them at same time. All the groups belonging to the same parent group have the same color but with different shade.

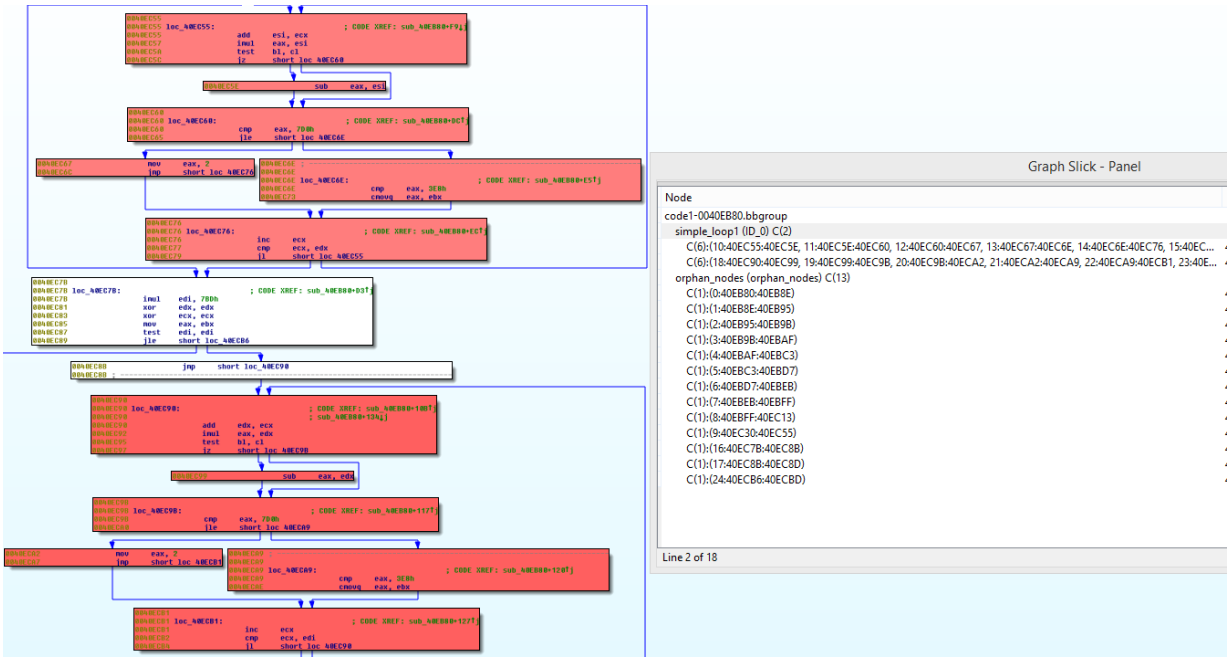

Figure 5.1: Result of a demo execution of GraphSlick.

A priori the use of this plugin is not as powerful as other more sophisticated approaches explained above, but it has the advantage that is easier to implement and since it is for IDA Pro, like the script already used for WBC detection, if it is possible to make them work together with acceptable results would be a functional and cheap solution in terms of time.

The decided approach was the validation of the GraphSlick plugin with real life binaries, and if it would be able to detect a good enough percentage of inline functions, the relative addresses showed by the plugin in the binary and tagged as inline functions could be collected. Then, modifying the current WBC detection script to pass a list of these addresses, it could be possible to perform the same measurements explained in Section 5.1 plus the evaluation of functions without declaration, forcing to the script to start from the relative address given by GraphSlick as it would do it at the beginning of any other function.

As conclusion of the design, it looks feasible to integrate GraphSlick for the detection of inline functions with the current WBC detection script. If this integration works as expected, the objective proposed for this project will be achieved. If that is not the case, additional research or self-development will be necessary to find a new approach.

Due to the necessary time to reverse engineer manually the possible inline functions indicated by the tool and the complexity of them it was not possible to develop the design further. However, as explained in Chapter 9, the development of this idea is listed as priority future work, aiming to improve the efficiency of the current WBC detection tool.

Chapter 6

Basic usage and configuration of Unicorn-Engine

This chapter tries to show how a basic configuration is set up to emulate a simple binary. This is the most basic level that is needed to prior to start the development of a tool such as the one implemented in this project.

6.1 Configuration of the emulation core

Two important references used during the development of the emulation tool based on Unicorn-Engine were the unofficial API [Nao] (facing the lack of an official one) and a guidance about the framework published by Eternal Stories [Sto].

Following the second reference given, a basic configuration of Unicorn-Engine is presented below to illustrate how a simple binary can be executed:

```
1  from unicorn import *
2  from unicorn.arm_const import *
3  import struct
4
5  def read(name):
6      with open(name) as f:
7          return f.read()
8
9  # Function used to create unsigned 32 integers
10 def u32(data):
11     return struct.unpack("I", data)[0]
12
13 # Function used to create 32b pointers
14 def p32(num):
15     return struct.pack("I", num)
```

```
16
17  # Configuration of Unicorn-Engine to emulate ARM32
18  # In this mode Unicorn is also able to emulate Thumb at the same
    ↪  time, switching between this two architectures like a normal ARM
    ↪  CPU would do it
19  mu = Uc (UC_ARCH_ARM, UC_MODE_32)
20
21  BASE = BASE_ADDRESS     # Base address of the memory that the
    ↪  emulator will use
22  STACK_ADDR = 0x0    # Starting address of the Stack
23  PAGE_SIZE = 0x1000  # Definition of the size of a page of memory
24  STACK_SIZE = PAGE_SIZE * 2 # Size of the Stack
25
26  mu.mem_map(BASE, 1024*1024) # Giving memory to Unicorn to allocate
    ↪  the binary
27  mu.mem_map(STACK_ADDR, STACK_SIZE)  # Giving memory to Unicorn to
    ↪  allocate the Stack
28
29
30  mu.mem_write(BASE, read(BINARY_NAME)) # Writing in memory of the
    ↪  emulator the binary
31  mu.reg_write(UC_ARM_REG_SP, STACK_ADDR + STACK_SIZE - 0x4)    #
    ↪  Giving initial value to the Stack Pointer
32
33  mu.emu_start(START_ADDRESS, END_ADDRESS)    # Start of the emulation,
    ↪  Unicorn will emulate from <START_ADDRESS> to <END_ADDRESS> of the
    ↪  loaded binary
```

6.2 Configuration of a basic hook

Hooks in Unicorn-Engine are the name that receive the functions that are in charge of the binary instrumentation of the emulated code. Unicorn supports several types of hooks, that are executed in specific situations. The most used ones and used also in the development of this project are:

- `UC_HOOK_CODE`: This constant indicates that the hook will be executed before each emulated instruction is executed.
- `UC_HOOK_INSN`:This constant indicates that the hook will be executed for a particular instruction.
- `UC_HOOK_INTR`: This constant indicates that all interrupt/syscall events will be hooked.

- `UC_HOOK_MEM_FETCH`: This constant indicates that every memory fetch for execution events will be hooked.
- `UC_HOOK_MEM_UNMAPPED`: This constant indicates that the hook will be executed just before the emulated binary will try to access to a memory address that was not mapped into the emulator.
- `UC_HOOK_MEM_INVALID`: This constant indicates that the hook will be executed in every illegal memory access
- `UC_HOOK_MEM_READ`: This constant indicates that the hook will be executed before each time the emulated code read something from memory.
- `UC_HOOK_WRITE`:This constant indicates that the hook will be executed before each time the emulated code write something from memory.

For instance, an Unicorn-Engine hook that would perform a basic tracing of the executed instructions of the emulator, giving information about which one were executed and the size of it, would have a structure similar to the one given below:

```
def hook_code(mu, address, size, user_data):
    print('>>> Tracing instruction at 0x%x, instruction size = 0x%x'
     ↪  %(address, size))

mu.hook_add(UC_HOOK_CODE, hook_code)
```

Chapter 7

Development of the emulator

This chapter explains the followed process to create the first functional version of the emulator using Unicorn Engine framework as base for it.

7.1 Design decision

First of all, it was necessary to choose the programming language that would have to be used to build the emulator. Checking Unicorn´s documentation [AQHVa] and GitHub repository [AQHVb], it was decided to use the Python binding instead of the pure C code. The reason behind this decision was the flexibility and multi-platform capabilities of Python. Just having a Python project would not be necessary to compile it for different architectures, and would give more opportunities to other engineers to modify something specific that could improve the performance of the emulator in a specific situation when facing a binary.

After a research process about Unicorn Engine emulation framework, and facing the lack of official documentation, it was decided to use the unofficial documentation reference [Nao] as main guidance for the implementation, digging into the source code of the framework to find more specific details if needed.

7.2 Code organization

In order to make tidy and scalable implementation, the code of the tool has been split into different modules. Each module has a different main purpose, and it is represented by a `.py` file. This structure is shown in Figure 7.1 and defined as follows:

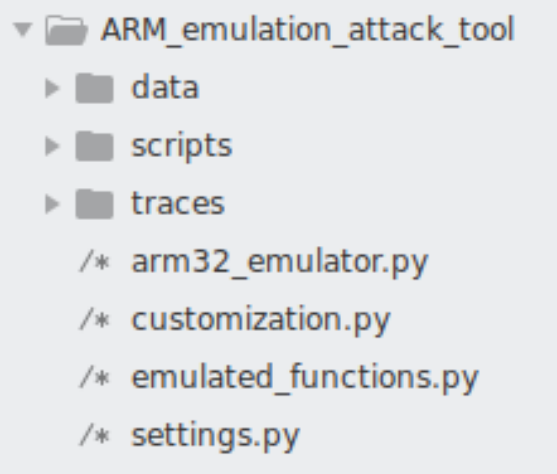

Figure 7.1: Code structure of the emulator.

- `arm32_emulator.py`: This file is the core of the tool. It contains the `main` function and implements the parameters processing, sets up Unicorn to emulate ARM 32 bits, loads the library in memory and attaches to the emulator the hooks explained in 7.4 to the emulator.

- `settings.py`: This file contains the global variables that are used by the different modules of the tool during emulation and the banner that is shown to the user when the emulator is invoked.

- `customization.py`: This file contains two functions that specify the way that the emulator will set up the initial configuration of memory before starting the emulation, and how it will retrieve the output of the emulated piece of code at the end. This file is meant to be modified by the user for each binary that he/she wants to emulate, because since Unicorn is a CPU emulator, the work of the dynamic linker to set up the initial values of memory before to start an execution has to be done manually.

- `emulated_functions.py`: This file contains a pool of the most common `libc` library functions that a WBC could call in case of a non-monolithic design. They are called by `arm32_emulator.py` when it detects an external call, then they are emulated in Python and finally the return process is emulated to return normally to the binary code.

- `data` folder: It contains a pool of files with a pair *input* - *output* in each file. This data is generated in each emulation, and it is used to store the *plaintext* - *ciphertext* pairs that are needed for the different types of attacks.

- `traces` folder: It contains the memory I/O trace files created by the emulator

during the execution of a binary. Each file corresponds to a complete trace of the binary with a given input.

- `scripts` folder: This folder is meant to store auxiliary scripts for different purposes, such as DCA and DFA analysis scripts.

The resulting code organization is shown in Figure 7.1.

Using this structure is pretended to make the development easier, at the same time that allows an easier access to the code in the future for further improvements of the tool.

7.3 Emulation core

A previous R&D project developed in Brightsight was used as a starting point for this Thesis. In the previous implementation, a basic core to emulate small binaries and trace operations with memory was implemented by an intern in Brightsight as part of his Master Thesis called "Unicorn Toolchain for attacks on White-Box cryptosystems using Host Card Emulation" [Sur17] (the Thesis is not public-domain available, but it is available internally in the company). This implementation was studied, and the useful parts of it were taken for the development of this tool.

This emulation core corresponds basically to the configuration of Unicorn-Engine to emulate ARM32 binaries (as shown in Chapter 6), to map memory for the emulation, load the binary into this memory, and create a simple hook to retrieve information after every memory I/O instruction.

It is important to mention that the tool developed during this project only contains a basic configuration of Unicorn-Engine to emulate ARM32 (a similar version of the code shown in Chapter 6) as inheritance of the previous project mentioned above. The way to allocate memory for the binaries of the previous project, which was emulating the way that an ARM dynamic linker does it, that is loading only the segments with the "`PT_LOAD`" flag, resulted in errors during execution for bigger and more complex binaries. The basic hook that was implemented in the previous project was used as base and example to build a group of more complex hooks (see Section 7.4) to perform more complex and specific tasks.

7.4 Instrumentation functions

As explained in Chapter 6.2, Unicorn has the capability to dynamically instrument the emulated binaries through hooks. The emulation tool implements several types of hooks that are used to perform different tasks, explained below:

- Two hooks are loaded for every execution:
  - `hook_instrace`: This hook is executed before every instruction is executed. It is in charge of disassembling the instruction that is going to be executed,

detecting if the instruction contains a call to an external function, redirecting the execution to an emulated function in Python in case that it corresponds to one of the `libc` functions implemented in `emulated_functions.py`, or skipping the jump in case that the function is not present, preventing an immediate crash of the emulation.

The list of the emulated functions that are implemented in the tool nowadays can be seen in Appendix B.

- `hook_mem_access_dca`: This hook is executed every time that the emulated code tries to access to memory. The code of this hook takes the data read-/written from/to memory and add to the trace file with the format:

  ```
  HEX_COUNTER, PC: <R/W> MEMORY_ADDRESS DATA_SIZE VALUE
  ```

- Two more hooks are loaded if the emulator has to perform a DFA attack:
  - `inject_fault_on_mem_access`: Used to inject the fault during the emulation of the binary.
  - `hook_mem_unmapped`: Used to prevent emulation errors after the injection of faults.

  These two hooks are explained in detail in Section 7.7.

  - Finally, one more hook is loaded if the emulator has to perform a memory dump at a specific point of the execution of the binary:
    - `hook_memory_dump`: This hook is invoked every time that a new instruction is going to be executed. If the hex counter of the instruction specified through parameters is equal to the one that will be executed, the emulation of the binary is stopped and the memory range specified also by parameters is dumped into a file in the folder "*data*" with name "*dump_MEM-ADDRESS-START_MEM-ADDRESS_END.dmp*".

The piece of source code 7.1 shows the section of the code where the different hooks explained above are inserted into the emulation tool.

```
        # Adding hooks to Unicorn
        if settings.FI_RANGE:
            mu.hook_add(UC_HOOK_MEM_READ,
             ↪  inject_fault_on_mem_access, user_data=fd_trace)
            mu.hook_add(UC_HOOK_MEM_READ_UNMAPPED |
             ↪  UC_HOOK_MEM_WRITE_UNMAPPED |
             ↪  UC_ERR_WRITE_UNMAPPED | UC_ERR_READ_UNMAPPED,
             ↪  hook_mem_unmapped, user_data=fd_trace)
        if settings.DUMP:
            mu.hook_add(UC_HOOK_CODE, hook_memory_dump,
             ↪  user_data=0)
        mu.hook_add(UC_HOOK_CODE, hook_instrace,
         ↪  user_data=fd_trace)
        mu.hook_add(UC_HOOK_MEM_READ | UC_HOOK_MEM_WRITE,
         ↪  hook_mem_access_dca, user_data=fd_trace)
```

Source code 7.1: Inclusion of the hooks in the tool.

7.5 Memory tracing

The tracing of the memory accesses of the execution of the binary with different input/output values is the base of the DCA attack. In order to retrieve this information, binary instrumentation is used through a hook attached to the emulator. This hook is `hook_mem_access_dca` and its behaviour is explained in Section 7.4.

Setting up the customization script to generate a different input to the binary for each execution, and giving to the emulator the number of execution traces that the user wants to collect using the `-n | --num-traces` parameter, it is possible to obtain a big enough pool of traces to perform a successful DCA attack by searching for correlation in those traces, thus obtaining the secret key of the WBC.

7.6 Plotting traces

An important part in an attack against White-Box Cryptography is the visual analysis of the execution traces looking for potential patterns that could leak the time-frame of the execution where the cryptographic algorithm is executed. Therefore the Python library `matplotlib` was used for that purpose.

In this implementation only the instruction counter, a number starting from `0x0` and used to uniquely identify each executed instruction, and the address of the instruction where a memory I/O operation was performed is plotted.

During execution, if the "*plot*" flag was set as parameter, an empty array is created before to start the emulation, and it is filled tuples of {instruction counter-PC} of the

instruction performing memory I/O operation. As mentioned before, this instruction counter is not the Program Counter, instead is a simple counter that starts from `0x0` with an increment of 1 for every memory I/O operation. Within this information, the user could identify relevant patterns in the execution trace, what could lead to gain knowledge about the approximate point where the execution of the cryptographic algorithm starts and end inside the WBC. An example of a trace plotted with the tool can be seen in Figure 7.2, where an execution of the Chow AES 128-bit WBC public-domain implementation [CEJVO03b] has been emulated and plotted, showing a clear 10-loop pattern.

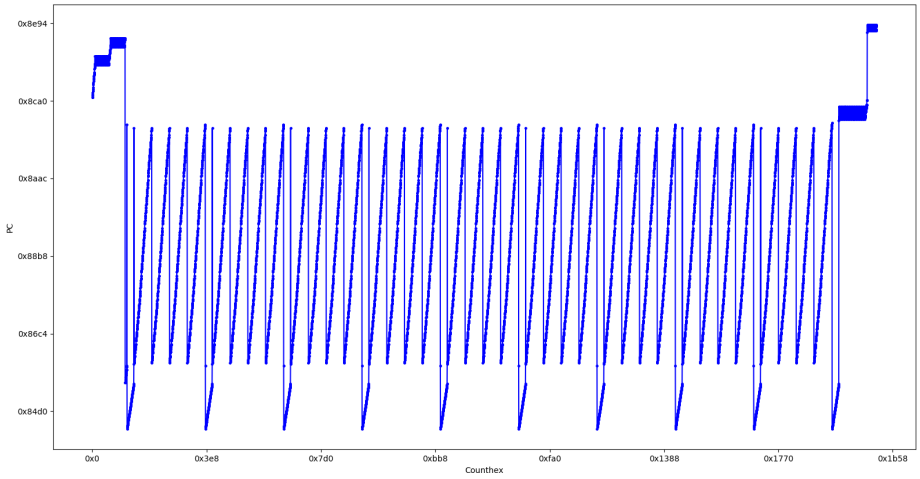

Figure 7.2: Chow AES-128 execution plot.

Additionally, `matplotlib` library offers the possibility to inspect the trace deeply, having zoom in and out options, move the trace in the chart, and having the coordinates of the mouse cursor always in the screen, making easier the process to identify a concrete range of the trace.

7.7 Fault injection

The way that the tool has to inject faults in the execution of a binary is the modification of the values read from memory. In order to do this, binary instrumentation is used through the implementation of two hooks injected to Unicorn as explained in Section 7.4.

The hooks implemented for this purpose are listed and explained below:

- `inject_fault_on_mem_access`: This hook is executed when the emulated code tries

to read from memory. If the executed instruction is in the specified range of the hexadecimal counter used to identify each memory I/O instruction, the value which is going to be read is modified with the objective of creating a fault.

The way that the tool is able to inject the fault is variable, and it has to be specified in the parameters before to start the emulation. These different ways to create a fault are listed below:

- Random fault: A randomly chosen bit is switched.
- Specific bit: A specified bit through parameters is switched.
- Fault applying a mask: A specified mask through parameters is used to modify the value read from memory. The mask has different values that corresponds with different behaviours, differentiating between uppercase letters to apply the mask to the whole byte and lowercase letters to apply the mask to a specific bit. The function of each character of the mask is described as follows:
  - `x|X`: Equivalent to do not modify the value of the bit/byte.
  - `r|R`: Modify the value of a bit/byte with a random value between `0` and `1`.
  - `f|F`: Flip the value of a bit/byte.
  - `0|1`: Fix the value of a specific bit to `0` or `1`.

- `hook_mem_unmapped`: This hook is triggered when the emulator tries to access to a memory position that is not mapped. This may happen because of the nature of the fault injection attack. When this attack injects faults in a range of instruction during execution, it is possible that the instruction could be modified in a way that the behaviour of the binary is corrupted.

  This hook was implemented to act as a `try/catch` statement, redirecting the execution of the emulator to it in order to finalize the execution in a controlled manner, giving at the same time about the wrong fault that was injected.

Additionally, using the `-ef | --extra-faults` flag in the parameters is possible to inject additional faults in concretes moments of the execution, after the injection of the first one. This feature is implemented in order to allow the pentesters to bypass double calculations implemented as security countermeasure to check if one of the encryption/decryption processes were affected by a fault or anomaly.

Injecting these faults, and with the help of the retrieval of the generated output with the help of the customization script, it is possible to capture the faulty outputs generated by the binary, and later on, by using the analysis tools, it is possible to perform the mathematical attack to find the key embedded in the WBC.

7.8 Additional utilities

In addition to the emulation tool created as core of this project, additional scripts have been added and created to make the work easier for different aspects.

The scripts added to the tool that were not developed during this project belong to the Side-Channel Marvels[1] GitHub repository, concretely the scripts from the JeanGrey module. These scripts are used for the mathematical analysis of the traces on DFA attacks on AES-128 bit implementations. As future work, in-house developed scripts for this purpose will be implemented.

An script implemented during this project is `install_dependencies.sh`, which installs all the necessary Linux packages and Python libraries for the correct execution of the tool.

At least one more script will be implemented during December before finishing this project. The idea of this script is to plot already collected traces from files, removing the need to emulate the same binary again just to plot the trace.

7.9 Usability improvements

In order to make the tool as much easy as possible for the final users and/or other security specialists that would want to make improvements on it, some usability decisions were taken during design and development of the tool.

The most relevant ones are explained in the following subsections.

7.9.1 Modularity

As explained in Section 7.2, the code is organized in a modular way that the code is easier to understand to the people that were not involved in the development of the tool.

7.9.2 Parameters processing

In order to process the parameters passed to the tool previous to each emulation, the `argparse` Python library was used. Within this library the declaration and processing of these parameters is made quite simple, moreover creating the *help* message.

The piece of source code 7.2 shows how the parameters are configured in the tool, concretely in the `arm32_emulator.py` file, and Figure 7.3 shows how the *help* message is shown in terminal when is invoked.

```
# Arguments parser
parser = argparse.ArgumentParser(description='Unicorn emulation tool
 ↪  to obtain traces from a ARM 32 bits binary WBC
 ↪  implementation.\nFor a better performance, it is recommended to
 ↪  redirect the output of the script to a file.',
 ↪  formatter_class=argparse.RawTextHelpFormatter)
```

[1] https://github.com/SideChannelMarvels

```
parser.add_argument('-F', '--file', action='store', dest='conf_file',
 ↪  type=str, help='Configuration file for the emulation.\nThis file
 ↪  can contain one assignation per line (VAR_NAME = var_value) for
 ↪  the next variables:\n + PC_LOG_RANGE\n + FI_RANGE\n +
 ↪  RDM_FAULT\nBITS_FAULT\n + DATA_TYPE\n + NUM_TRACES\n + STACK\n +
 ↪  STACK_SIZE\n + ENTRY\n + RET\n + FREE_MEMORY_ZONE\n + PAGE_SIZE\n
 ↪  + EXTRA_FAULTS')
parser.add_argument('-e', '--entry-point', action='store',
 ↪  dest='entry_point', type=auto_int, help='Entry point for the
 ↪  emulation')
parser.add_argument('-r', '--return-point', action='store',
 ↪  dest='return_point', type=auto_int, help='Return point for the
 ↪  emulation')
parser.add_argument('-s', '--load-sections', help="Load also sections
 ↪  of the binary in memory", action="store_true")
parser.add_argument('--pc-log-range', action='store', nargs=2,
 ↪  dest='pc_log_range', type=auto_int, default=None, help='PC range
 ↪  to log traces', metavar=('INITIAL_PC', 'FINAL_PC'))
parser.add_argument('-t', '--measure-time', action='store_true',
 ↪  help='Measure the execution time')
parser.add_argument('-f', '--fault-injection', action='store',
 ↪  nargs='+', dest='fi_range', type=auto_int, default=None,
 ↪  help='Instruction number range to generate faults',
 ↪  metavar=('INITIAL_INS', 'FINAL_INS'))
parser.add_argument('-ef', '--extra-faults', action='store',
 ↪  nargs='+', dest='extra_faults', type=auto_int, default=None,
 ↪  help='Instructions where to inject extra faults')
parser.add_argument('--rdm-fault', action='store', nargs=2,
 ↪  dest='rdm_fault', type=auto_int, help='Fault injection affecting
 ↪  NUMBER random bits at the BYTE_AFFECTED byte (0 for all bytes)',
 ↪  metavar=('RDM_BITS', 'BYTE_AFFECTED'))
parser.add_argument('-b', '--bits-fault', action='store', nargs='+',
 ↪  dest='bits_fault', type=int, help='Fault injection affecting a
 ↪  list of concrete bits', metavar='bitN')
parser.add_argument('-c', '--custom-fault', action='store',
 ↪  dest='custom_fault', type=str, help='Fault injection applying a
 ↪  custom mask:\n + x/X => Do not touch that bit/byte\n + f/F =>
 ↪  Flip that bit/byte\n + r/R => Randomize that bit/byte\n + 0/1 =>
 ↪  Fix value for that bit', metavar='MASK')
```

```
parser.add_argument('--dump', action='store', nargs=3, dest='dump',
↪  type=auto_int, help='Memory dump when the instruction counter has
↪  the value <INSTRUCTION_COUNTER> for the memory space beginning at
↪  <STARTING_ADDRESS> for <SIZE>', metavar=('INSTRUCTION_COUNTER',
↪  'STARTING_ADDRESS', 'SIZE'))
parser.add_argument('-d', '--data-type', action='store',
↪  dest='data_type', type=str, default='u32', help='Data type to be
↪  written in memory after the fault injection [u32, p16, p32]')
parser.add_argument('-n', '--num-traces', action='store',
↪  dest='num_traces', type=int, default=1, help='Number of traces
↪  generated')
parser.add_argument('-p', '--plot', action='store_true', help='Plot
↪  instructions counter and PC in a chart')
parser.add_argument('-v', '--verbose', help="Enable debug messages",
↪  action="store_true")
parser.add_argument(action='store', type=str, dest='input_file',
↪  help="Binary file to be emulated")
arguments = parser.parse_args(sys.argv[1:])
```

Source code 7.2: Parameters processing of the emulator.

```
> python arm32_emulator.py -h
usage: arm32_emulator.py [-h] [-F CONF_FILE] [-e ENTRY_POINT]
                         [-r RETURN_POINT] [-s]
                         [--pc-log-range INITIAL_PC FINAL_PC] [-t]
                         [-f INITIAL_INS [FINAL_INS ...]]
                         [-ef EXTRA_FAULTS [EXTRA_FAULTS ...]]
                         [--rdm-fault RDM_BITS BYTE_AFFECTED]
                         [-b bitN [bitN ...]] [-c MASK]
                         [--dump INSTRUCTION_COUNTER STARTING_ADDRESS SIZE]
                         [-d DATA_TYPE] [-n NUM_TRACES] [-p] [-v]
                         input_file

Unicorn emulation tool to obtain traces from a ARM 32 bits binary WBC implementation.
For a better performance, it is recommended to redirect the output of the script to a file.

positional arguments:
  input_file            Binary file to be emulated

optional arguments:
  -h, --help            show this help message and exit
  -F CONF_FILE, --file CONF_FILE
                        Configuration file for the emulation.
                        This file can contain one assignation per line (VAR_NAME = var_value) for the next variables:
                         + PC_LOG_RANGE
                         + FI_RANGE
                         + RDM_FAULT
                        BITS_FAULT
                         + DATA_TYPE
                         + NUM_TRACES
                         + STACK
                         + STACK_SIZE
                         + ENTRY
                         + RET
                         + FREE_MEMORY_ZONE
                         + PAGE_SIZE
                         + EXTRA_FAULTS
  -e ENTRY_POINT, --entry-point ENTRY_POINT
                        Entry point for the emulation
  -r RETURN_POINT, --return-point RETURN_POINT
                        Return point for the emulation
  -s, --load-sections   Load also sections of the binary in memory
  --pc-log-range INITIAL_PC FINAL_PC
                        PC range to log traces
  -t, --measure-time    Measure the execution time
  -f INITIAL_INS [FINAL_INS ...], --fault-injection INITIAL_INS [FINAL_INS ...]
                        Instruction number range to generate faults
  -ef EXTRA_FAULTS [EXTRA_FAULTS ...], --extra-faults EXTRA_FAULTS [EXTRA_FAULTS ...]
                        Instructions where to inject extra faults
  --rdm-fault RDM_BITS BYTE_AFFECTED
                        Fault injection affecting NUMBER random bits at the BYTE_AFFECTED byte (0 for all bytes)
  -b bitN [bitN ...], --bits-fault bitN [bitN ...]
                        Fault injection affecting a list of concrete bits
  -c MASK, --custom-fault MASK
                        Fault injection applying a custom mask:
                         + x/X => Do not touch that bit/byte
                         + f/F => Flip that bit/byte
                         + r/R => Randomize that bit/byte
                         + 0/1 => Fix value for that bit
  --dump INSTRUCTION_COUNTER STARTING_ADDRESS SIZE
                        Memory dump when the instruction counter has the value <INSTRUCTION_COUNTER> for the memory space beginning at <STARTING_ADDRESS> for <SIZE>
  -d DATA_TYPE, --data-type DATA_TYPE
                        Data type to be written in memory after the fault injection [u32, p16, p32]
  -n NUM_TRACES, --num-traces NUM_TRACES
                        Number of traces generated
  -p, --plot            Plot instructions counter and PC in a chart
  -v, --verbose         Enable debug messages
```

Figure 7.3: Help message generated by the tool.

7.9.3 Configuration script

As explained in Section 7.2, one of the modules of the tool is the `customization.py` and its meant to be modified by the user before the execution of a binary in order to specify how the tool has to prepare the emulation environment before to start the execution of the binary.

After studying different options to customize how to setup the emulation environment, the conclusion was that this mechanism was considered the most suitable one, giving two Python functions to set up the initialization and the output retrieval of the execution in

a isolated file.

7.9.4 Debugging information

In order to make easier the task of debugging during the initial implementation of the tool and future improvements, the emulation tool has several debugging messages spread in its code and meant to give useful information about the execution. This debugging messages are created with the `logging` Python library, and can be invoked using the `-v | --verbose` parameter.

Chapter 8

Validation of the emulator implementation

This chapter explains the procedures that were followed to validate the correct emulation of the tool and later on how the attacking features were validated.

8.1 Sample White-Box Cryptography implementations chosen

During development, an small implementation of Chow AES-128 bit [CEJVO03b] implementation have been used. The implementation of this WBC is an adapted version of a public repository[1].

It was decided to use this design of WBC because of several reasons:

- The first one is the simplicity and the small size of its implementation, avoiding problems with memory space, allocation, etc.
- The second one is that this implementation is known and broken for a long time, having a lot of information publicly available in Internet.
- The third and last one maybe is not relevant from the technical side, but historically this was one of the first WBC-AES implementation, being his WBC-DES the first White-Box Cryptography implementation of the history. Therefore it was considered appropriated to start the first emulations of this tool with the first WBC algorithm.

A trace obtained from this Chow-AES implementation obtained at the end of the development is shown in Figure 7.2.

[1] `https://github.com/0vercl0k/stuffz/tree/master/wbaes_attack/wbaes128`

In order to validate the correct performance of the tool once it was completed, a public repository from the *WhiBOx contest*[2] CTF challenge of the conference CHES 2017 was downloaded. All of these WBC implementations are AES-128.

The different implementations that were presented to the contest were cross-compiled for ARM 32-bit architecture.

8.2 Validation experiments

8.2.1 Binary execution

As first step, a `main` function was created in order to make them standalone executable. As part of the contest rules, every binary has the same function skeleton to perform encryption, therefore the same `main` function could be used. The implemented function can be seen in the piece of source code 8.1.

```
int main (int argc, char *argv[]) {
    unsigned char ciphertext[16];
    unsigned char *ciphertext_ptr = ciphertext;
    unsigned char *plaintext_ptr = argv[1];

    AES_128_encrypt(ciphertext_ptr, plaintext_ptr);

    return 4;
}
```

Source code 8.1: `main` function created to call to WBC.

Previous emulation of a binary, it is necessary to set up the `customization.py`, as explained in Section 7.9.3. Using a decompilation tool the relative address of the beginning of the `main` function and the end of it, or in this case the address of instruction just after the call to `AES_128_encrypt`.

Then, different binaries were executed using the tool with promising results. No unexpected errors appeared, the emulated output is correct in comparison with a normal execution and both the plot of the traces and the tracing of the memory I/O operations and its later processing and saving in files were successful.

Figures 8.1 and 8.2 shows two examples of these emulations and its results.

[2] `https://whibox-contest.github.io/dashboard`

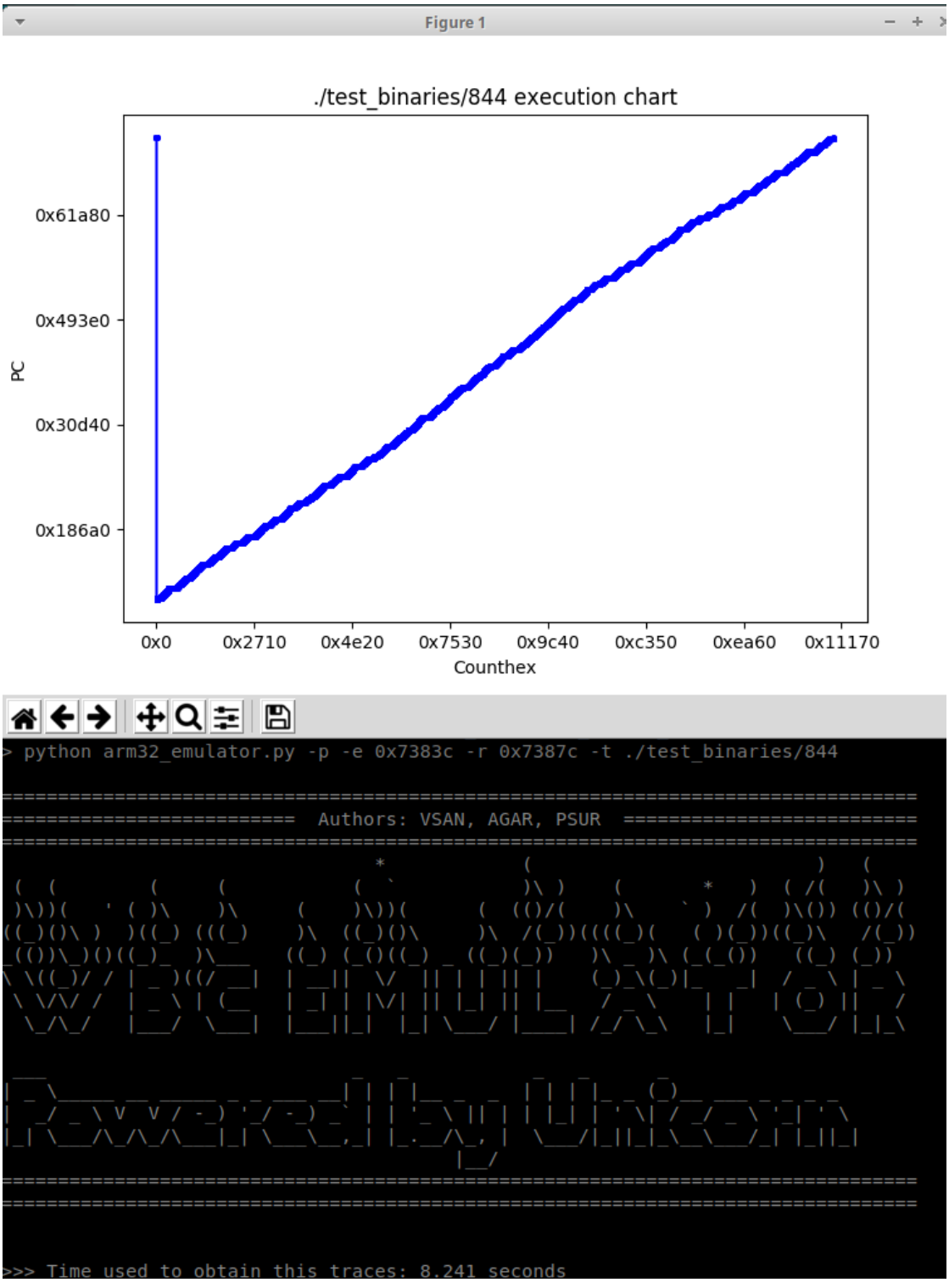

Figure 8.1: Results of the execution of the binary with name 844[3].

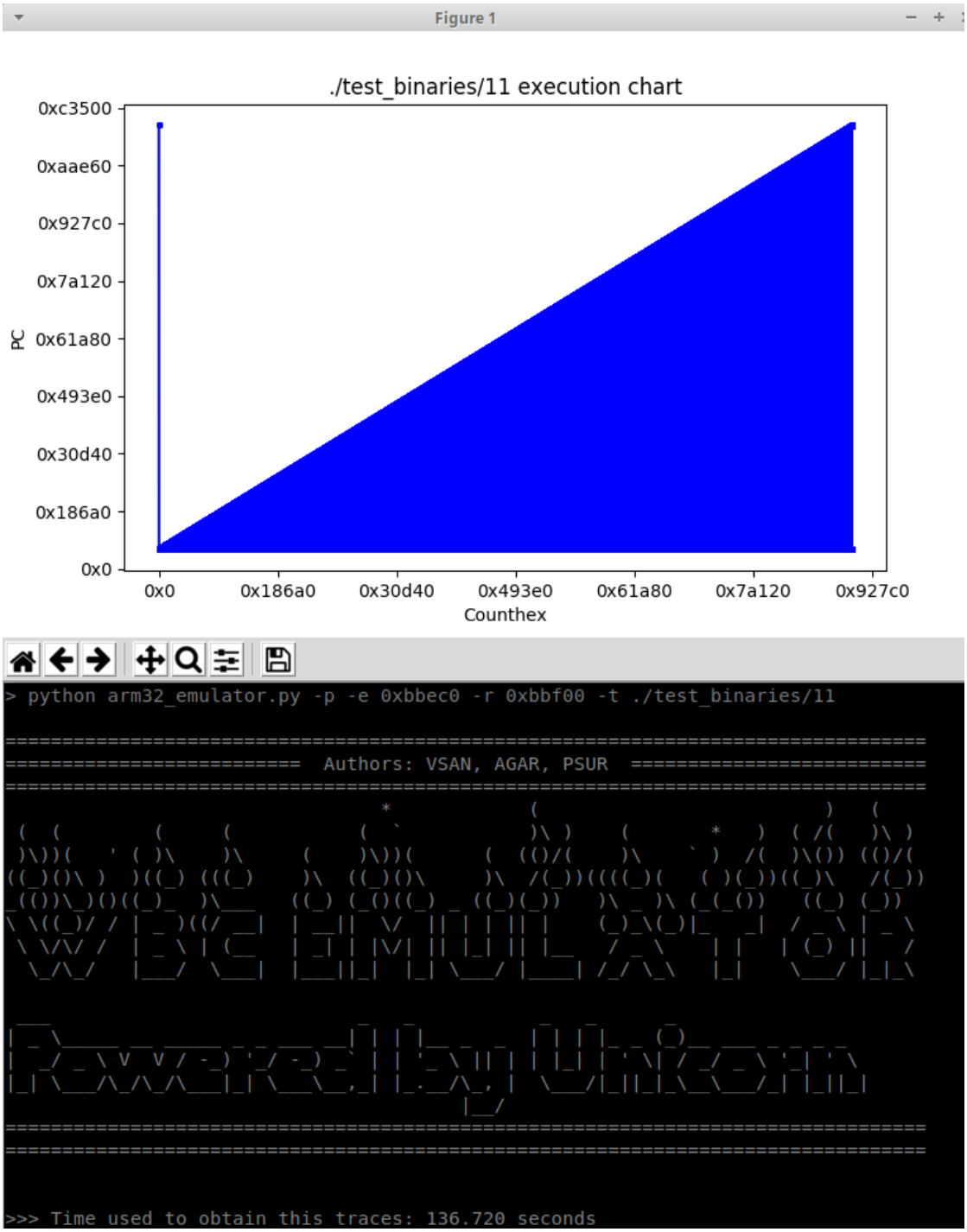

Figure 8.2: Results of the execution of the binary with name 11[4].

[3] https://whibox-contest.github.io/show/candidate/844

The overhead on the execution time created by the emulator is in some cases reasonable, giving good execution times. However other times, probably because of the features of the WBC emulated, the emulation takes a long time. Preliminary analysis of the problem indicates that the biggest overloading is in the stack due to the huge amount of calls implemented in the WBC. It is important to take into consideration that the WBC implementations of the contest are design to be complex and difficult to attack, and not to be fast, therefore maybe part of the problem is the WBC itself. However, this is something that has to be studied deeply in order to find the origin of the problem and fix it if possible.

8.2.2 DCA memory tracing

As explained in Section 7.5, during every emulation the tracing hook is attached dumping the trace to a file. Therefore, with the same validation binaries shown in the previous Section the tracing capabilities of the tool were validated.

Checking the trace files after both emulations was assessed that if the emulation is successful, the trace file is populated correctly. An example of one of these resultant trace files, Figure 8.3 shows the file created with the emulation of the binary with name *11* and input `"ABCDEFGHIJKLMNOP"`.

```
0x1, 0xbbec0: W 0xa7fff800  4 0x28185702
0x2, 0xbbec0: W 0x00000000  4          0
0x3, 0xbbecc: W 0x00000002  4        0x2
0x4, 0xbbed0: W 0xa7fffa00  4 0x28185707
0x5, 0xbbed8: W 0xa7fff7e4  4 0x28185702
0x6, 0xbbedc: R 0xa7fff7dc  4 0x28185707
0x7, 0xbbee0: R 0xa7fffa04  4 0x28185704
0x8, 0xbbee4: W 0xa7fff900  4 0x28185704
0x9, 0xbbee8: R 0xa7fff7f8  4 0x28185702
0xa, 0xbbeec: R 0xa7fff7f4  4 0x28185704
0xb, 0x8718: W 0xa7fff800  4 0x28185702
0xc, 0x8718: W 0x000bbef4  4   0x769780
0xd, 0x8724: W 0xa7fff7e4  4 0x28185702
0xe, 0x8728: W 0xa7fff900  4 0x28185704
0xf, 0x872c: R 0xa7fff7cc  4 0x28185704
0x10, 0x8730: R 0xa7fff900  1       0x65
0x11, 0x8734: R 0x00009734  4  0x6617284
0x12, 0x873c: W 0x00000041  4       0x65
```

Figure 8.3: Part of the trace file resultant of the emulation of the binary 11.

In the portion of the shown trace can be seen how the beginning of the input value

[4] `https://whibox-contest.github.io/show/candidate/11`

is read from memory (`"A" = 0x65, "B" = 0x66`, etc.).

8.2.3 Fault injection

A second emulation of the two binaries mentioned above (*844* and *11*) was executed, indicating this time to the tool that a fault has to be injected at a particular point of the execution.

It is important to mention that since the objective of this test was to check the correct execution of the tool according to the expected behaviour, the injection of these faults did not have as goal the obtainment of a successful faulty output usable in a DFA attack, and therefore errors in the emulation were caused by them. However, just cross-comparing two execution traces: one without any modification and the second one injecting a fault, was enough to check that the behaviour of the tool is correct, forcing to the emulated code to read a modified value from memory at a specific moment of the execution.

For instance, the call to the tool to create a faulty emulation of the binary *11*, injecting a random fault of 1 bit in the first byte (less significant byte) of the memory read in the instruction identified with the counter `0x10` can be seen in Figure 8.4.

Figure 8.4: Call to the tool to provoke a faulty emulation.

Then, comparing the traces saved by the tool of these emulation, it is visible that the value read is different in the faulty execution (see Figures 8.5 and 8.6).

```
0x1, 0xbbec0: W 0xa7fff800  4 0x28185702
0x2, 0xbbec0: W 0x00000000  4          0
0x3, 0xbbecc: W 0x00000002  4        0x2
0x4, 0xbbed0: W 0xa7fffa00  4 0x28185707
0x5, 0xbbed8: W 0xa7fff7e4  4 0x28185702
0x6, 0xbbedc: R 0xa7fff7dc  4 0x28185707
0x7, 0xbbee0: R 0xa7fffa04  4 0x28185704
0x8, 0xbbee4: W 0xa7fff900  4 0x28185704
0x9, 0xbbee8: R 0xa7fff7f8  4 0x28185702
0xa, 0xbbeec: R 0xa7fff7f4  4 0x28185704
0xb, 0x8718: W 0xa7fff800  4 0x28185702
0xc, 0x8718: W 0x000bbef4  4   0x769780
0xd, 0x8724: W 0xa7fff7e4  4 0x28185702
0xe, 0x8728: W 0xa7fff900  4 0x28185704
0xf, 0x872c: R 0xa7fff7cc  4 0x28185704
0x10, 0x8730: R 0xa7fff900  1       0x65
0x11, 0x8734: R 0x00009734  4  0x6617284
0x12, 0x873c: W 0x00000041  4       0x65
```

Figure 8.5: 12 first instructions saved in the trace file in a normal emulation.

```
0x1, 0xbbec0: W 0xa7fff800  4 0x28185702
0x2, 0xbbec0: W 0x00000000  4          0
0x3, 0xbbecc: W 0x00000002  4        0x2
0x4, 0xbbed0: W 0xa7fffa00  4 0x28185707
0x5, 0xbbed8: W 0xa7fff7e4  4 0x28185702
0x6, 0xbbedc: R 0xa7fff7dc  4 0x28185707
0x7, 0xbbee0: R 0xa7fffa04  4 0x28185704
0x8, 0xbbee4: W 0xa7fff900  4 0x28185704
0x9, 0xbbee8: R 0xa7fff7f8  4 0x28185702
0xa, 0xbbeec: R 0xa7fff7f4  4 0x28185704
0xb, 0x8718: W 0xa7fff800  4 0x28185702
0xc, 0x8718: W 0x000bbef4  4   0x769780
0xd, 0x8724: W 0xa7fff7e4  4 0x28185702
0xe, 0x8728: W 0xa7fff900  4 0x28185704
0xf, 0x872c: R 0xa7fff7cc  4 0x28185704
0x10, 0x8730: R 0xa7fff900  1       0x97
0x11, 0x8734: R 0x00009734  4  0x6617284
0x12, 0x873c: W 0x00000061  4       0x97
```

Figure 8.6: 12 first instructions saved in the trace file in a faulty emulation.

The obtained results were good, showing that the faults were injected were pointed. However, it is not possible to validate completely this feature of the tool before its use in several DFA real attacks, to verify that the injection of this faults is affecting only to the specific instruction or range of instruction and in the way that is indicated by the user.

Chapter 9

Conclusions and Future Work

This chapter summarizes the objectives that have been completed during this Thesis, as well as the tasks that have to be finished in the future and the reasons for it.

9.1 Achieved objectives

During this project several objectives have been achieved, leaving the final state of the future final tool closer. The first one is the knowledge gained about the state of the art in White-Box Cryptography detection in binaries, emulation techniques and two of the most important attacks for this type of cryptography such as DCA and DFA.

Another one is the implementation of the Unicorn-based CPU emulator, giving full control over the emulated binary, and with capabilities to trace the I/O operations in memory during execution and the possibility to inject arbitrary faults. This tool gives to the security evaluators the capability to perform the first part of a DCA/DFA attack, giving an output that can be used later on directly in a mathematical analysis tool for the rest of the attack.

Something that is considered at this moment as a partial achievement is the design of the proposed improvement for the WBC detection tool. After the research task, it was decided that a good option to integrate inlined functions detection in this tool is the use of the IDA Pro plugin called GraphSlick. If the integration of the current tool with the features of this plugin is possible, the capabilities to detect hidden function in strongly obfuscated binaries for its later processing will be highly improved with an acceptable cost in terms of development time. This task will be developed during 2019 as an internal R&D project in the company. This project will have priority in the R&D queue due to the important reverse engineering help that it could provide during security evaluations.

9.2 Future work

Due to time constraints and the complexity of the project some objectives were not achieved in time or were moved to future development iterations.

The improvement of the White-Box Cryptography detection tool was planned to be part of this project. The original version of this tool, taken as starting point for this project, is an IDA Pro script. Due to the lack of available time to work on this task and the limited access to IDA Pro only during working hours, made unfeasible the development of these improvements.

Other relevant task to be done in the future is the further research about the long emulation times obtained during the validation of the tool. Some of the binaries taken for validation showed that the tool takes several hours to complete one execution. It is true that the WBC implementation of the contest are not designed to be fast but would be good to understand the problem in order to gain more knowledge about the internal emulation environment of Unicorn.

After the two previous tasks are finished, the next step will be the research about how feasible is to combine and automatize both tools to create a complete tool that is able to handle the detection and emulation of WBC, and automatize the process as much as possible. If it is possible, the integration of both tools will be performed together with the automation of the process as planned at the beginning of the project.

The last task that would be interesting to research about is the implementation of the Correlation Computational Analysis (CCA) attack. This attack takes a special advantage of the characteristics of the ARM processors in the case of Android. Since every value has to pass through registers to be processed, it can be assumed that the intermediate values of the cryptographic algorithm will be in a register at some point of the execution. Therefore, applying correlation analysis between the values in every register in every CPU clock cycle and pre-calculated values of the different steps of a cryptographic algorithm such as SBOX transformations or output values per round could lead in the leakage of the correct value, and with enough intermediate values is possible to retrieve the complete WBC key.

Bibliography

[AFG+15] Julien Allibert, Benoit Feix, Georges Gagnerot, Ismael Kane, Hugues Thiebeauld, and Tiana Razafindralambo. Chicken or the egg - computational data attacks or physical attacks. Cryptology ePrint Archive, Report 2015/1086, 2015. `https://eprint.iacr.org/2015/1086`.

[AQHVa] Nguyen Anh Quynu and Dang Hoang Vu. Unicorn engine documentation. `https://www.unicorn-engine.org/docs`.

[AQHVb] Nguyen Anh Quynu and Dang Hoang Vu. Unicorn engine github repository. `https://github.com/unicorn-engine/unicorn`.

[AQHV15] Nguyen Anh Quynh and Dang Hoang Vu. *Unicorn Engine. The ultimate CPU emulator*, 2015. `https://www.unicorn-engine.org/`.

[AWD16] Saed Alrabaee, Lingyu Wang, and Mourad Debbabi. Bingold: Towards robust binary analysis by extracting the semantics of binary code as semantic flow graphs (sfgs). *Digital Investigation*, 18:S11 – S22, 2016.

[Bel05] Fabrice Bellard. Qemu, a fast and portable dynamic translator. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference*, ATEC '05, pages 41–41, Berkeley, CA, USA, 2005. USENIX Association.

[BHMT15] J.W. Bos, C. Hubain, W.P.A.J. Michiels, and P. Teuwen. *Differential computation analysis : hiding your white-box designs is not enough.* Cryptology ePrint Archive. IACR, 2015.

[CEJvO03a] Stanley Chow, Phil Eisen, Harold Johnson, and Paul C. van Oorschot. A white-box des implementation for drm applications. In Joan Feigenbaum, editor, *Digital Rights Management*, pages 1–15, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[CEJVO03b] Stanley Chow, Philip Eisen, Harold Johnson, and Paul C. Van Oorschot. White-box cryptography and an aes implementation. In Kaisa Nyberg and Howard Heys, editors, *Selected Areas in Cryptography*, pages 250–270, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[CMJMS10] Juan Caballero, Noah M. Johnson, Stephen McCamant, and Dawn Song. Binary code extraction and interface identification for security applications. 10 2010.

[GWH11] Felix Gröbert, Carsten Willems, and Thorsten Holz. Automated identification of cryptographic primitives in binary programs. In Robin Sommer, Davide Balzarotti, and Gregor Maier, editors, *Recent Advances in Intrusion Detection*, pages 41–60, Berlin, Heidelberg, 2011. Springer Berlin Heidelberg.

[JW16] Michael J. Wiener. Evolution of white-box cryptography: From table-based implementations to recent designs, 2016. `https://www.cryptoexperts.com/whibox2016/slides-whibox2016/EvolutionofWhite-BoxCryptography.pdf`.

[KJJ99] Paul Kocher, Joshua Jaffe, and Benjamin Jun. Differential power analysis. In Michael Wiener, editor, *Advances in Cryptology — CRYPTO' 99*, pages 388–397, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[LGF15] Pierre Lestringant, Frédéric Guihéry, and Pierre-Alain Fouque. Automated identification of cryptographic primitives in binary code with data flow graph isomorphism. In *Proceedings of the 10th ACM Symposium on Information, Computer and Communications Security, ASIA CCS '15, Singapore, April 14-17, 2015*, pages 203–214, 2015.

[Nao] Tomori Nao. Unicorn engine reference (unofficial). `https://hackmd.io/s/rJTUtGwuW#`.

[PBR17] Roberta Piscitelli, Shivam Bhasin, and Francesco Regazzoni. *Fault Attacks, Injection Techniques and Tools for Simulation*, pages 27–47. 01 2017.

[Pet11] Fabien A. P. Petitcolas. Kerckhoffs' principle. In Henk C. A. van Tilborg and Sushil Jajodia, editors, *Encyclopedia of Cryptography and Security (2nd Ed.)*, page 675. Springer, 2011.

[RPB13] Ali Rahbar, Ali Pezeshk, and Elias Bachaalany. Ida plugin - graphslick, 2013. `https://github.com/lallousx86/GraphSlick`.

[SMdH15] Eloi Sanfelix, Cristofaro Mune, and Job de Haas. Unboxing the white-box: Practical attacks against obfuscated ciphers, 2015. `https://www.blackhat.com/docs/eu-15/materials/eu-15-Sanfelix-Unboxing-The-White-Box-Practical-Attacks-Against-Obfuscated-Ciphers-wp.pdf`.

[SOF12] KEYS IN SOFTWARE. White-box cryptography : Hiding keys in software. 2012.

[Sto] Ethenal Stories. Unicorn engine tutorial. http://eternal.red/2018/unicorn-engine-tutorial/.

[Sur17] Prasanna Suryanarayanan. Unicorn toolchain for attacks on white-box cryptosystems using host card emulation. Master's thesis, Technological University of Eindhoven, Netherlands, 2017.

[TH16] Philippe Teuwen and Charles Hubain. Differential fault analysis on white-box aes implementations, 2016.

[VC17] Terugu Venkat Chalapathi. How white-box cryptography is gradually eliminating the hardware security dependency, 2017. https://medium.com/engineering-ezetap/how-the-white-box-cryptography-gradually-eliminating-the-hardware-security-dependency-40622d516e02.

[WWW17] Shuai Wang, Pei Wang, and Dinghao Wu. Semantics-aware machine learning for function recognition in binary code. pages 388–398, 09 2017.

Appendix A

Temporary extension of the project

The temporary distribution of this project has been more dispersed than normal due to the fact that I am working, which together with the subjects of the semester did not leave a lot of free time to work on the thesis, therefore my daily dedication to TFM during some periods did not exceed 2 hours per day. On the other hand, Brightsight assigned 8 hours of work every two weeks to work on it, however due to the high load of projects it was not possible to use this time many days, accumulating at the end of this project 50 hours of work at the office.

Talking now about the organization of the project, it followed the initial schedule planned at the beginning of it, with the exception of the implementation of the WBC detection tool improvements, that due to time constraints and the lack of time to work with IDA Pro at the office was not possible to finish it.

About the emulation tool, which after this project is ready to be used as a stable release, the process was really similar to a any other development project. After enough research effort to understand the state of the art of White-Box Cryptography and documentation review to understand Unicorn-Engine, the implementation started. Once a new piece of code was finished it was tested to prevent unexpected bugs in the final composite, ending with another round of tests to be sure that the final result works as expected. After every bug found was fixed, a set of WBC implementations from CHES 2017 contest were emulated in order to validate the correct emulation and the correct injection of faults in them.

The total time expent for this project is 225 hours of work. Figure A.1 shows the temporary extension of the project in a Gantt diagram.

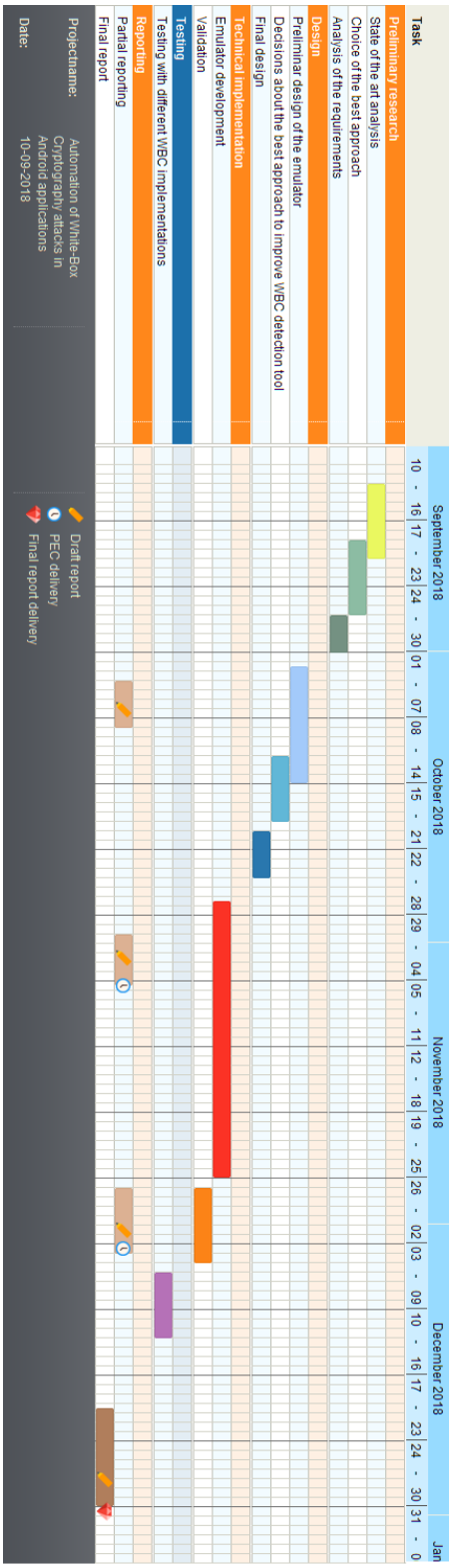

Figure A.1: Gantt diagram of the project.

Appendix B

Emulated functions supported by the emulation tool

As explained during this report, the assumption used during the development of this tool is that a secure implementation of a White-Box should be monolithic, with exceptional cases where calls to common libraries such as `libc` could be eventually used. In those exceptional cases the emulation tool is able to intercepts the jumps to the `GOT` table of the binary before it redirects the execution to any external function. Then, if the external call is to a function that is emulated by the tool, the emulation is stopped, the parameters are taken from the registers and the Stack, and the function is emulated in Python. Finally, the return value is set in the register `R0` like an ARM CPU would do it and the emulation is restarted at the next instruction after the external call.

The following list shows the functions that can be emulated by the tool using Python. These functions are nowadays only coming from `libc` library.

- `memcpy`
- `malloc`
- `strcpy`
- `strcmp`
- `strlen`
- `sscanf`
- `printf`