University of Twente
Faculty of Electrical Engineering, Mathematics and Computer Science (EEMCS)

Master Thesis

Behavioral Analysis of Obfuscated Code

Federico Scrinzi
1610481
f.scrinzi@student.utwente.nl

Graduation Committee:

Prof. Dr. Sandro Etalle (1st supervisor)

Dr. Emmanuele Zambon

Dr. Damiano Bolzoni

UNIVERSITEIT TWENTE.

Abstract

Classically, the procedure for reverse engineering binary code is to use a disassembler and to manually reconstruct the logic of the original program. Unfortunately, this is not always practical as obfuscation can make the binary extremely large by overcomplicating the program logic or adding bogus code.

We present a novel approach, based on extracting semantic information by analyzing the behavior of the execution of a program. As obfuscation consists in manipulating the program while keeping its functionality, we argue that there are some characteristics of the execution that are strictly correlated with the underlying logic of the code and are invariant after applying obfuscation.

We aim at highlighting these patterns, by introducing different techniques for processing memory and execution traces.

Our goal is to identify interesting portions of the traces by finding patterns that depend on the original semantics of the program. Using this approach the high-level information about the business logic is revealed and the amount of binary code to be analyze is considerable reduced.

For testing and simulations we used obfuscated code of cryptographic algorithms, as our focus are DRM system and mobile banking applications. We argue however that the methods presented in this work are generic and apply to other domains were obfuscated code is used.

Acknowledgments

I would like to thank my supervisors Damiano Bolzoni and Eloi Sanfelix Gonzalez for their encouragement and support during the writing of this report. My work would have never been carried out without the help of Ileana Buhan (R&D Coordinator at Riscure B.V.) and all the amazing people working at Riscure B.V., that gave me the opportunity to carry out my final project and grow professionally and personally. They provided excellent feedback and support throughout the development of the project and I really enjoyed the atmosphere in the company during my internship. I would also like to thank my friends and fellow students of the EIT ICTLabs Master School for their encouragement during this two years of studying and all the fun moments spent together.

Contents

CHAPTER 1

Introduction

In the last years, obfuscation techniques became popular and widely used in many commercial products. Namely, they are methods to create a program P' that is semantically equivalent to the original program P, but “unintelligible” in some way and more difficult to interpret by a reverse engineer. There are different reasons why a software engineer would prefer to protect the result of his or her work against adversaries, some examples include the following:

- Protecting intellectual property (IP): as algorithms and protocols are difficult to protect with legal measures [1], also technical ones needs to be employed to ensure unauthorized creation of program *clones*. Examples of software that include additional protection are iTunes, Skype, Dropbox or Spotify.

- Digital Rights Management (DRM): DRM are employed to ensure a controlled spreading of media content after sale. Using this kind of technologies, the data is usually offered encrypted and the distribution of the key for decrypting is controlled by the selling entity (e.g.: the movie distributor or the pay-tv company). Sometimes the usage of proprietary hardware solutions that implement DRM technologies is possible but often it is not. In these situations there is the need of implementing everything in software. Nevertheless, in both cases technical measures for protecting against reverse engineering are employed, in order to protect algorithm implementations and cryptographic keys.

- Malware: criminals that produce malware to create botnets, receive ransoms or steal private information, as well as agencies that offer their expertise on the development of surveillance software, need to protect their products against reversing. This is important in order to keep being effective, undetected by anti-viruses and act undisturbed.

These use-cases have all a common interest: research and invention of more and more powerful techniques to prevent reverse engineering.

The job of understanding what a binary, output of a common compiler, does is not always a trivial task. When additional measures to harden the process are in place this could become a nightmare. Reverse engineers strive to find new and easier ways of achieving their final goal: understanding every or most of the details of what a program is doing when is running on our CPUs. In the last years, an arms race has been going on between developers, willing to protect their software, and analysts, willing to unveil the algorithm behind the binary code.

There are different reasons why it would be interesting or useful to understand how effective these techniques are and how it would be possible to break them and somehow retrieve an understandable pseudocode from an obfuscated binary. The most obvious one is in the case of malware: as security researchers the public safety is important and we want to protect Internet users from criminals that illegally take control of other people's machines. Understanding how a malware works means also preventing its spreading.

On the other hand one could think that in general de-obfuscation of proprietary programs is unethical or even criminal [2], but this in not always the case. There are good and acceptable reasons to break the protections employed by commercial software. One example is to prove how secure the protection is and how much effort it requires to be broken, through security evaluations. This is useful especially for the developers of DRM solutions. Another interesting use case for reverse engineering of protected commercial software is to know if it includes backdoors, critical vulnerabilities or is simply doing operations that could be considered malicious. For a concrete example we could refer to the Sony BMG scandal: between 2005 and 2007 the company developed a *rootkit* that infected every user that inserted an audio CD distributed by Sony in a Windows computer. This rootkit was preventing any unauthorized copy of the CD but was also modifying the operating system and was later even exploited by other malware [3].

1.1 Research objectives

State-of-the-art obfuscators can add various layers of transformations and heavily complicate the process of reverse engineering the semantics of binary code. In most cases it is unpractical to obtain a complete understanding of the underlying logic of a program. For an analyst, there is often the need to first collect high-level information and identify interesting parts, in order to restrict the scope of the analysis.

From our experiments we observed that there are distinctive high-level patterns in the execution that are strictly bounded to the underlying logic of the program and are invariant after most transformation that preserve semantic equivalency, such as obfuscation. We argue that it is possible to highlight these patterns by analyzing the behavior of an execution.

The objective of this thesis is to develop a novel methodology for reverse engineering obfuscated binary code, based on the analysis of the behavior of the program. As a program can be defined as a sequence of instructions that perform computation using memory, we can describe its behavior by recording in which sequence the instructions are executed and which memory accesses are performed. These traces can be collected using dynamic analysis methods. Thus, we aim at processing these traces and extract insightful information for the analyst.

Analysis of the behavior of obfuscated code is a new method for extracting information from the output of dynamic analysis, therefore to understand the strength of this approach we test its effectiveness against sample programs. Next, to show the invariance after obfuscation: we compare the observed behavior of state-of-the-art obfuscated samples with the one of the same samples in a non-obfuscated form.

1.2 Outline

This report is organized as follows: in Chapter 2, a classification of obfuscation techniques will be presented, introducing state-of-the-art-research in the protection of software. Then, advances in its counterpart, de-obfuscation, will be discussed. In Chapter 3, techniques for analyzing memory and execution traces in order to extract semantic information of the target program will be presented. Chapter 4 will introduce an evaluation benchmark for these methods and results will be discussed. Finally, Chapter 5 will present some final remarks and observations for future developments.

CHAPTER 2

State of the art

2.1 Classification of Obfuscation Techniques

Even though an ideal obfuscator is proven by Barak et al. not to exist [4], many techniques were developed to try to make the reversing process extremely costly and economically challenging. Informally speaking we can say that a program is difficult to analyze if it performs a lot of instructions for a simple operation or it's flow it's not logical for a human. These descriptions however lack of rigorousness and are dubious. For these reasons many theoreticians tried to categorize these techniques and several models were proposed to describe both an obfuscator and a de-obfuscator [5, 6].

For our purposes we will base our categorization on the work of Collberg et al. from 1997 [6], augmenting it with more recent developments in the field [7, 8, 9, 10]. First we will introduce *control-based* and *data-based* obfuscation. Later more advanced *hybrid techniques* will be presented.

2.1.1 Control-based Obfuscation

By basing the analysis on assumptions about how the compiler translates common constructs (*for* and *while* loops, *if* constructs, etc.), it is often possible to reliably obtain an higher level view of the control flow structure of the original code. In a pure compiled program spatial and temporal locality properties are usually respected: the code belonging to the same basic block will in most cases be sequentially located and basic blocks referenced by other ones are often close together. Moreover we can infer additional properties: a prologue and epilogue will probably mean the beginning and the end of a function, a *call* instruction will generally invoke a function while a *ret* will most likely return to the caller.

Control flow obfuscation is defined as altering “the flow of control within the code, e.g. reordering statements, methods, loops and hiding the actual control flow behind irrelevant conditional statements” [11], therefore the assumptions mentioned earlier do not hold anymore.

The following are examples of control-based obfuscation techniques.

Ordering transformations Compiled code follows the principle of spatial locality of logically related basic blocks. Also, blocks that are usually executed near in time are placed adjacent in the code. Even though this is good for performance reasons thanks to caching, it can also provide useful clues to a reverse engineer. Transformations that involve reordering and unconditional branches break these properties.

Clearly this does not provide any change in the semantics of the program, however the analysis performed by a human would be slowed down.

Opaque predicates An opaque predicate is a special conditional expression whose value is known to the obfuscator, but is difficult for an adversary to deduce statically. Ideally its value should be only known at obfuscation time. This construct can be used in combination with a conditional jump: the correct branch will lead to semantically relevant code, the other one to junk code, a dead end or uselessly complicated cycles in the control graph. In practice, a conditional jump with an opaque predicate looks like a conditional jump but in practice it acts as an unconditional jump. For implementing these predicates, complex mathematical operations or values that are fixed, but are only known at runtime, can be used.

Functions In/Out-lining As from a call graph it is possible to infer some information on the underlying logic of the program, it is sometimes desirable to confuse the reverse engineer with an apparently illogic and unmeaningful graph. Functions inlining is the process of including a subroutine into the code of its caller. On the other hand function outlining means separating a function into smaller independent parts.

Control indirection Using control flow constructs in an uncommon way is an effective way for making a control graph not very meaningful to an analyst. For example instead of using a *call* instruction it is possible to dynamically compute the address at runtime and *jump* there, also *ret* instructions can be used as branches instead of returns from functions.

A more subtle approach is to use exception or interrupt/trap handling as control flow constructs. In detail, first the obfuscated program triggers an exception, then the exception handler is called. This can be controlled by the program and perform some computation, or simply redirect the instruction pointer somewhere else or change the registers.

It is also possible to further exploit these features: Bangert et al. developed a Turing-complete machine using the page faults handling mechanisms, switching from MMU to CPU computation using control indirection techniques [12].

2.1.2 Data-based Obfuscation

This category of techniques deals with the obfuscation of data structures used by the program. The following are examples of data-based obfuscation techniques.

Encoding For many common data types we can think of "natural" encodings: for example for strings we would use arrays of bytes using ASCII as a mapping between the actual byte and a character, on the other hand for an integer we would interpret 101010 as 42. Of course these are mere conventions that can be broken to confuse the reverse engineer. Another approach is to use a custom mapping between the actual values and the values processed by the program. It is also possible to use *homomorphic* mappings, so we can perform computation on the encoded data and decode it later [13].

Constant unfolding While compilers, for efficiency purposes, substitute calculations whose result is known at compile time with the actual result, we can use the very same technique in the reverse way for obfuscation. Instead of using constants we can substitute them with a possibly overcomplicated operation whose result is the constant itself.

Identities For every instruction we can find other semantically equivalent code that makes them look less "natural" and more difficult to understand. Some examples include the use of "*push addr; ret*" instead of a "*jmp addr*", "*xor reg, 0xFFFFFFFF*" instead of "*not reg*" or arithmetic identities such as "$\sim -x$" instead of "$x + 1$"

2.1.3 Hybrid techniques

For clarity and orderliness first control-based and data-based obfuscation techniques were presented. In practice these techniques are combined to reach higher levels of obfuscation and make the reversing process more and more difficult.

The following sections will present some advanced techniques, employed in the real world in many commercial applications.

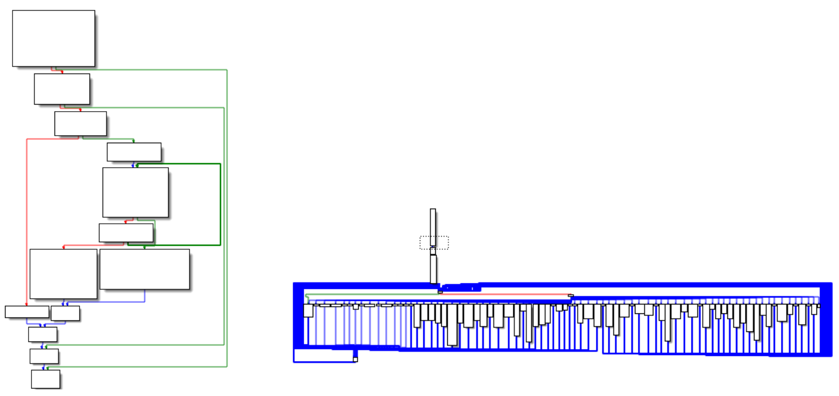


Figure 2.1: A control flow graph before and after code flattening

Source: N. Eyrolles et al. (Quarkslab)

Control-flow flattening Control-flow flattening (or code flattening) is an advanced control-flow obfuscation technique that is usually applied at function-level. The function is modified such that, basically, every branching construct is replaced with a big *switch* statement (different implementations use *if-else* constructs, calling of sub-functions, etc. but the underlying principle remains unaltered). All edges between basic blocks are redirected to a *dispatcher* node and before every branch an artificial variable (i.e. the dispatcher context) needs to be set. This variable is used by the dispatcher to decide which is the next block where to jump.

Clearly, by applying this technique any relationship between basic blocks is hidden in the dispatcher context. The control flow graph doesn't help much in understanding the logic behind the program as all basic blocks have the same set of ancestors and children. To harden even more the program other techniques can be included: complex operations or opaque predicates to generate the context, junk states or dependencies between the different basic blocks.

This technique was first introduced by C. Wang [14] and later improved by other researchers and especially by the industry. Figure 2.1 shows an example of the control flow graphs of a program before and after the code flattening obfuscation. This transformation is used in many commercial products, some examples include Apple FairPlay or Adobe Flash.

Virtual machines An even more advanced transformation consists in the implementation of a custom virtual machine. In practice, an ad-hoc instruction set is defined and selected parts of the program are converted to opcodes for this VM. At runtime the newly created bytecode will be interpreted by the virtual machine, achieving a semantically equivalent program.

Even though this technique implies a significant overhead it is effective

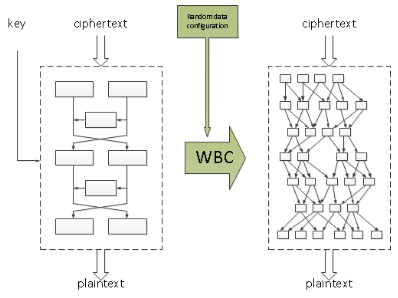


Figure 2.2: An overview of white-box cryptography

Source: Wyseur et al.

in obfuscating the program. In fact, an adversary needs to first reverse engineer the virtual machine implementation and understand the behavior of each opcode. Only after these operations it will be possible to decompile the bytecode to actual machine code.

White-Box Cryptography Cryptography is constantly deployed in many products where there is no secure element or other trusted hardware, a typical example are software DRM. In these contexts the adversaries control the environment where the program runs, therefore, if no protection is in place, it is trivial to extract the secret key used by the algorithm. A possible approach is for instance setting a breakpoint just before the invocation of the cryptographic function and intercept its parameters. Implementing cryptographic algorithms in a *white-box attack context*, namely a context where the software implementation is visible and alterable and even the execution platform is controlled by an adversary, is definitely a challenge. There the implementation itself is the only line of defense and needs to well protect the confidentiality of the secret key.

White-box cryptography (WBC) tries to propose a solution to this problem. In a nutshell, B. Wyseur describes it as following: “The challenge that white-box cryptography aims to address is to implement a cryptographic algorithm in software in such a way that cryptographic assets remain secure even when subject to white-box attacks” [15]. In practice, the main idea is to perform cryptographic operations without revealing any secret by merging the algorithm with the key and random data, in such a way that the random data cannot be distinguished from the confidential data (see Figure 2.2).

As demonstrated by Barak et al. [4] a general implementation of an obfuscator that is resilient to a white-box attack does not exist. However it remains of interest for researchers to investigate on possible white-box implementations of specific algorithms, such as DES or AES [16, 17]. Chow et al. proposed as first a white-box DES implementation in 2002. Even though it was broken in 2007 by Wyseur et al. [18] and Goubin et al. [19], it laid the foundation for research in this field.

In the real world WBC is implemented in different commercial products by many companies such as Microsoft, Apple, Sony or NAGRA. They deployed state-of-the-art obfuscation techniques by creating software implementations that embody the cryptographic key.

2.2 Obfuscators in the real world

Even though, for economic reasons, the most research in the area of obfuscation is carried out by companies and is often kept private, we can find in literature different examples of obfuscators. Those are mainly used as proof of concepts for validating research hypothesis and rarely used in practice, also because the fact that the obfuscator is public poses a threat in the security-by-obscurity of this protection mechanism.

Some of the most interesting approaches to this problem that can be found in literature are based on LLVM. It is one of the most popular compilation frameworks thanks to the plethora of supported languages and architectures. Additionally, its Intermediate Representation (IR) allows to have a common language that is independent from the starting code and the target architecture. This enables researchers to develop obfuscators that just manipulate the IR code and consequently obtain support for all languages and platforms that are supported by LLVM, without any additional effort. *Confuse* [20] is one simple attempt to build an obfuscator based on LLVM implementing different widespread techniques. This tool offers basic functionalities like data obfuscation, insertion of irrelevant code, opaque predicates and control flow indirection. An interesting description about how LLVM works and how it is possible to exploit its features for software protection are explained in detail in the white paper by A. Souchet [21]. He developed *Kryptonite*, a proof-of-concept obfuscator for showing the potentiality of LLVM IR.

One of the most interesting advances in open source obfuscation tools is given by *Obfuscator-LLVM* (OLLVM) [22], an open implementation based on the LLVM compilation suite developed by the information security group of the University of Applied Sciences and Arts Western Switzerland of Yverdon-les-Bains (HEIG-VD). The goal of this project is to provide software security through code obfuscation and experiment with tamper-proof binaries. It currently implements instructions substitution, bogus control, control flow flattening and functions annotations. Additional features are under development while others are planned for the future.

Recently, University of Arizona released Tigress [23], a free diversifying source-to-source obfuscator that implements different kind of protections against both static and dynamic analysis. The authors claim that their technology is similar to the one employed in commercial obfuscators, such as Cloakware/IRDETO's Transcoder. Features offered by Tigress include virtualization with a randomly-generated instruction set, control flow flattening with different dispatching techniques, function splitting and merging, data encoding and countermeasures against data tainting and alias analysis.

On the market there are many commercial obfuscation solutions. The most famous include Morpher [24], Arxan [25] and Whitecryption [26]. Purely considering technical aspects, the availability of open source solutions is of great significance not only for academics but also for companies. Firstly, the fact of having access to the code makes it much easier to spot the injection of backdoors or security vulnerabilities in the final binary. Secondly, such a tool allows to experiment with new techniques, benchmark them against reverse engineering and develop more sophisticated protection mechanisms. Lastly, obfuscation tools can be used as a mitigation for exploitation: if each obfuscation is randomized it will be possible to easily and cheaply produce customized binaries, one for each customer, making the development of mass exploits very difficult. Clearly, as stated earlier closed source implementations might provide better protection as the obfuscation process is unknown. Nevertheless there are many advantages in open source solutions as well and probably a combination of these two different approaches can lead to higher quality results.

2.3 Advances in De-obfuscation

In the previous chapter we presented some widely deployed as well as effective techniques for software obfuscation. Now we can start asking ourselves different questions, in particular Udupa et al. [7] in their work addressed the following: "What sorts of techniques are useful for understanding obfuscated code?" and "What are the weaknesses of current code obfuscation techniques, and how can we address them?". The answers to those questions are important for different reasons. Firstly it is useful to know more about what the code we run on our machines is actually doing (e.g.: it could be a malware), secondly obfuscation techniques that are not really effective are not only useless but actually worse than useless: they increase the size of the program, decrease performance and also offer a false sense of security.

We need therefore to elaborate models and criteria to develop and evaluate de-obfuscation techniques. For this we can base our research on previous studies in the field of formal methods, compilers and optimizations. A first possible classification is given by Smaragdakis and Csallner [27], dividing static and dynamic techniques. With static analysis we mean the discipline of identifying specific behavior or, more generally, inferring information about a program without actually running it but by only analyzing the code. On the other hand dynamic analysis consists in all the techniques that require running a program (often in a debugger, sandbox or other controlled environment) for the purpose of extracting information about it. In practice, dynamic and static techniques are combined together, their synergy enhances the precision of static approaches and the coverage of dynamic ones.

The following paragraphs will briefly present various approaches to the de-obfuscation problem, introducing state-of-the-art general-purpose techniques that can help the reverse engineering process. Many attempts were made to develop automatic de-obfuscators [28, 29], however there is no "silver bullet" for solving this problem and currently most of the work needs to be carried out manually by the analyst. Nevertheless, the following techniques propose a defined methodology and basic tools to tackle an obfuscated binary.

Constants identification and pattern matching A simple static analysis technique consists in finding known patterns in the code. If the target binary implements some cryptographic primitive like SHA-1, MD5 or AES we can try to identify strings, numbers or structures that are peculiar of those algorithms. For a block cipher based on substitution-permutation networks it could be easy to recognize S-Boxes while for instance for public key cryptography it might be possible to find unique headers (e.g.: *"BEGIN PUBLIC KEY"*).

Also in the case of function inlining it is possible to use pattern matching techniques in order to identify similar blocks and therefore unveil the replication of the same subroutine. Replacing each occurrence if the pattern with the call of a function will hopefully lead to a more understandable code. The same can be applied against opaque predicates and constants unfolding: once a pattern is found and its final value is known we can substitute it with the obfuscated code.

Another similar technique that we can leverage is *slicing*. Introduced by Weiser [30], it consists in finding parts of the program that correspond to the mental abstraction that people make when they are debugging it.

Data tainting and slicing Dynamic analysis allows us to monitor code as it executes and thus perform analysis on information available only at run-time. As defined by Schwartz et al., "dynamic taint analysis runs a program and observes which computations are affected by predefined taint sources such as user input" [31]. In other words the purpose of taint analysis is to track the flow of specific data, from its source to its sink. We can decide to *taint* some parts of the memory, then any computation performed on that data will be also considered *tainted*, all the rest of the data is considered *untainted*. This operation allows us to track every flow of the data we want to target and all its derivations computed at run-time. It is particularly interesting in the case of malware analysis as we can for instance taint personal data present on our system and see if it is processed by the program and maybe exfiltrated to a “Command & Control” server.

To give an example, an implementation of this technique is present in *Anubis*, a popular malware analysis platform developed by the “International Secure Systems Lab” [32]. In the case of Android applications the system taints sensitive information such as the IMEI, phone number, Google account and so on, and runs the program in a sandbox, checking if tainted data is processed.

Data slicing is a similar technique. While tainting attempts to find all derivations of a selected piece of information and their flow, slicing works backwards: starting from an output we try to find all elements that influenced it [33].

Symbolic and concolic execution A simple approach for dynamic analysis is the generation of test-cases, execute the program with those inputs and check its output. This naive technique is not very effective and the coverage of all possible execution paths is usually not very high. A better approach is given by *symbolic execution*, a means of analyzing which inputs of a program lead to each possible execution path [34]. The binary is instrumented and, instead of actual input, symbolic values are assigned to each data that depends on external input. From constraints posed by conditional branches in the program an expression in terms of those symbols is derived. At each step of the execution is then possible to use a constraint solver to determine which concrete input satisfies all the constraints and thus allows to reach that specific program instruction.

Unfortunately symbolic execution is not always an option: there are many cases in which there are too many possible paths and we will reach a *state explosion* or the constraints are too complex to be solved, that makes the computation infeasible. For avoiding this problem we can apply *concolic execution* [35]. The idea is to combine symbolic and concrete execution of a program to solve a constraint path, maximizing the code coverage. Basically, concrete information is used to simplify the constraint, replacing symbolic values with real values.

Dynamic tracing Following the idea of symbolic and concolic execution it is also interesting, from a reverse engineering point of view, to obtain a concrete trace of the execution of a program. This allows us to have a recording of the execution and perform further offline analysis, visualize the instructions and the memory, show an overview of the invoked system calls or API calls and so on. This approach has also the advantage that we have to deal with only one execution of the program, so we only have one sequence of instructions. The analyst does not have to deal with branches, control-flow graphs or dead code, thus the reverse engineering process can be easier. Of course, we need to take into account that the trace might not include all the needed information.

Qira by George Hotz offers an implementation of this technique. It is introduced by the author as a "timeless debugger" [36] as it allows to go navigate the execution trace and see the computation performed by each instruction and how it modifies the memory. A different approach is offered by PANDA [37] which among other features allows to record an execution of a full system and replay it. The advantage of it is that it is possible to first record a trace with minor overhead, later we can run computationally intensive analysis on the recording without incurring in network timeouts or anti-debugging checks caused by a very slow execution.

Statistical analysis of I/O An alternative and innovative approach for automatically bypassing DRM protection in streaming services is introduced by Wang et al. [38]. They analyzed input and outputs from memory during the execution of a cryptographic process and determined the following assumptions:

- An encoded media file (e.g.: an MP3 music file) has high entropy but low randomness
- An encrypted stream has high entropy and high randomness
- Other data has low entropy and low randomness

Using these guidelines it is possible to identify cryptographic functions and intercepting its plaintext output by just analyzing I/O and treating the program as a black-box. There is no need of reversing the cryptographic algorithm nor knowing which is the decryption key, the only requirement is being able to instrument the binary and intercept the data read and written at each instruction in RAM. Their approach was shown to automatically break the DRM protection and get the high quality decrypted stream of different commercial applications such as Amazon Instant Video, Hulu, Spotify, and Netflix.

This work was later improved by Dolan-Gavitt et al. by showing how PANDA (Platform for Architecture-Neutral Dynamic Analysis) can be used to automatically and efficiently determine interesting memory location to monitor (i.e.: tap-points) [39, 40].

It is interesting to notice that this approach allows the completely automatic extraction of decrypted content from a binary employing different obfuscation techniques, only by leveraging statistical properties of I/O.

Advanced fuzzers Another approach that was recently developed is based on instrumentation-guided genetic fuzzers. Fuzzers are usually used for finding vulnerabilities by crafting peculiar inputs. These could have been unexpected by the developer of the program and could lead to unintended behavior. More advanced fuzzers leverage symbolic execution and advances in artificial intelligence to automatically understand which inputs trigger different conditions and follow different execution paths. M. Zalewsky developed *american fuzzy lop (afl)*, “a security-oriented fuzzer that employs a novel type of compile-time instrumentation and genetic algorithms to automatically discover clean, interesting test cases that trigger new internal states in the targeted binary”. He showed how it is possible to use *afl* against *djpeg*, an utility processing a JPEG image as input. His tool was able to create a valid image without knowing anything about the JPEG format but by only fuzzing the program and analyzing its internal states [41].

Decompilers Instead of dealing with assembly it is sometimes preferable to have a higher abstraction and handle pseudo-code. In the last years new tools were released to allow to obtain readable code from a binary: some examples are *Hopper*, *IDA Pro HexRays* which supports Intel x86 32bit and 64bit and ARM or *JD-GUI* for Java decompilation.

Unfortunately these tools rely on common translations of high-level constructs, thus some simple obfuscation techniques or the usage of *packers* could easily neutralize them. Even though they are not really resilient, it is worth employing them when there is the need to reverse engineer secondary parts of the code that are not heavily obfuscated or after some initial deobfuscation preprocessing.

CHAPTER 3

Behavior analysis of memory and execution traces

Reverse engineering obfuscated binaries is a very difficult and time consuming operation. Analysts need to be highly skilled and the learning curve is very steep. Moreover, in the common case of reversing of large binaries, it is unpractical to analyze the whole program. There is the need to identify interesting parts in order to narrow down the analysis. On top of this, obfuscation can heavily complicate the situation by adding spurious code and additional complexity.

As the amount of information collected using static and dynamic analysis can be overwhelming, we need effective techniques to gather high-level information on the program. Especially in the case of DRM implementations, it is important to understand which cryptographic algorithms are used and which parts of the code deal with the encryption process. This is needed, for instance, to collect information about the intermediate values to infer information on the secret key or to successfully perform fault injection attacks on the cryptographic implementation.

We argue that there are characteristics of the behavior of a program that heavily depend on the structure of the source code and can be revealed by an analysis of the execution. Furthermore, we show that these properties are invariant after transformations performed by obfuscators. This is intrinsic in the concept of obfuscator: as semantic equivalency needs to be guaranteed, most of the original structure needs to be preserved. Moreover, obfuscators are usually conservative while applying transformations to reduce failures to a minimum. We can exploit these properties for the purpose of reverse engineering, exploring side effects of the execution to gather insightful information.

A program is formed by a sequence of instructions that are executed by the processor, these instructions operate on the memory. Following from this, we derive the observation that the behavior of a program is well described by recording executed instructions and memory operations over time. We can collect this data through dynamic analysis, the extraction of useful information from these traces will be the focus of this report.

In summary, the underlying hypothesis of this project is that distinctive patterns in the logic of the program are reflected in the output of dynamic analysis, regardless of the complexity of the implementation or possible obfuscation transformations.

Continuing on these lines, from the side-channel analysis world we know that interesting information can be extracted from the analysis of different phenomenons, such as power consumption, electromagnetic emissions or even the sound produced during a computation. These methods are mostly not dependent on a specific implementation of the target algorithm and are not bounded to strong assumptions on the underlying logic, thus are applicable in a black-box context. We inspired our work to these techniques and we adapted them to reverse engineering of software. Compared to physical side channels, we can collect perfect traces of memory accesses and executed instructions. As we can completely control the execution environment, we do not have to to deal with imprecise data or issues due to the recording setups, like noise. On the other hand, the targets are usually much more complex and possibly obfuscated.

The main advantage of the proposed approach is that we can infer information about the target program without manually looking at the code. This fact highly simplifies the reverse engineering and allows the extraction of the semantics of almost arbitrary complex binaries. Also, the process is not bounded to a specific architecture, the same methods can be applied to any target. The main problem remains how to effectively process and show the collected data, in such a way that patterns are identifiable and are beneficial for the purpose of reverse engineering.

As already shown by related studies, data visualization can be a valuable and effective tool for tackling this kind of issues, especially when dealing with information buried together with other less meaningful data. In literature we can find different applications of visualization to the purpose of reverse engineering. Conti et al [42] showed different techniques and examples for the analysis of unknown binary file formats containing images, audio or other data. They claim that *"carefully crafted visualizations provide big picture context and facilitate rapid analysis of both medium (on the order of hundreds of kilobytes) and large (on the order of tens of megabytes and larger) binary files"*. It is possible to find similar research results in the field of software reversing, especially regarding malware analysis. Quist et al. used visualization of execution traces for better understanding the behavior of packed malware samples [43]. Trinius et al. instead focused on the visualization of library calls performed by the target program in order to infer information about the semantics of the code [44]. Also in the forensics world we can find attempts to use visual techniques, for example to identify rootkits [45] or to collect digital forensics evidence [46].

As these results show, visualization is a powerful companion for the analyst. Compared to other possible solutions, such as pattern recognition based on machine learning or other automatic approaches, it is generally applicable, it does not require fine tuning or ad-hoc training and the result of the analysis can be quickly interpreted by the analyst and enhanced with other findings.

Following from these premises, in our work we want to address the following research questions:

- Which information is inferable from memory and execution traces that is attributable to the behavior of the program and reveals information on its semantics, regardless of obfuscation?
- Which techniques are effective in highlighting this information and give useful insights in the business logic of the target program?

For this research project we developed different methods to extract information about the semantics of a program by analyzing its behavior. This section will introduce these techniques, divided in two categories: data-flow analysis and control-flow analysis. The former is focused on visualization of memory accesses, the discovery of repeating or distinctive patterns in the data-flow and the analysis of statistical properties of the data. The latter aims at giving information about the logic of the program by visualizing an execution graph, loops or repetitions of basic blocks and by using graph analysis to counter obfuscations of the control-flow.

In our work we recorded every memory access and every execution of basic blocks produced by target binary during one concrete execution. For the instruction trace we only record basic blocks addresses in order to keep the trace smaller and more manageable, it is implicit that every instruction in the basic block was executed. Table 3.1 shows the data that is recorded for every entry in the traces.

3.1 Data-flow analysis methods

The main rationale behind this category of analysis techniques is that sequences of memory accesses are tightly coupled with the semantics of the program. Most obfuscation methods are concerned of concealing the program logic by substituting instructions with equivalent (but more complex)

Memory Trace Entry	Execution Trace Entry
Type (Read/Write)	Basic block address
Memory address	Instruction count
Data	
Program Counter (PC)	
Instruction count	

Table 3.1: Description of the data recorded for each entry of the memory and execution traces.

ones or by tweaking the control-flow. However, distinctive patterns in the memory accesses remain unvaried and part of the data that flows to and from the memory is also unchanged. Moreover, when dealing with programs that process confidential data (e.g. cryptographic algorithms), we can use memory traces to extract secret information.

For all these reasons, we explored different possibilities in the analysis of the memory trace. The most simple technique is the visualization of memory accesses on an interactive chart. As the information showed by this method can be overwhelming, we present possible solutions to this problem. Different techniques will be discussed to reduce the scope of the analysis by focusing on parts of the execution that depend on user input.

Later, we move deeper in the analysis of the actual data that flows to and from the memory. We exploit statistical properties of the content of memory accesses, in terms of entropy and randomness, to unveil information from the execution. Next, we analyze the trace in terms of location of memory accesses, instead of their content. By applying auto-correlation analysis we aim at identifying repeated patterns in the accesses. These two techniques allow to take into account two diametrically opposed types of data, content and location of memory accesses, and thus gather a more complete picture of the behavior of the target program.

3.1.1 Visualizing the memory trace

As a first step, the memory trace is displayed in an interactive chart, where the x-axis represents the instruction count while the y-axis the address space. Every memory access performed by the target program is represented as a point in this 2D space.

This allows the analyst to visually identify memory segments (data, heap, libraries and stack) and explore the trace for finding interesting patterns or accesses that leak confidential information. Even though this technique is very simple, it can provide an insightful overview of parts of the execution, as well as allowing analysis similar to the ones performed with Simple Power

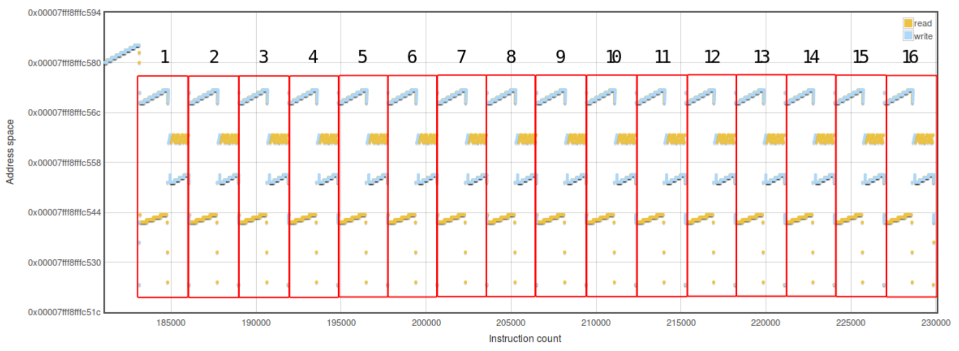


Figure 3.1: Memory reads and writes on the stack during a DES encryption. The 16 repeated patterns that represent the encryption rounds are highlighted.

Analysis (SPA).

A straightforward example is given by Figure 3.1, the plot of memory accesses during a DES encryption [1]. By interactively navigating the trace is possible to easily identify the part of the execution that performs the encryption operation. From the chart we can notice 16 similar patterns, composed by read and writes in different buffers. Only by using this information we can elaborate accurate hypotheses on the semantics of the code: each one of the 16 patterns probably represents one encryption round, buffers that are read and written are for the left and right halves of the Feistel Network or temporary arrays for the F function. Later, an analysis of the code can confirm these hypotheses.

Recovering an RSA key from OpenSSL A more complex practical application of this technique is given by the following example. We analyzed the memory accesses of OpenSSL while encrypting data using RSA. As we will show, the RSA implementation offered by OpenSSL (version 1.0.2a - latest at the moment of writing) reads from an array where the index is key-dependent. By simply visualizing these accesses we can recover the key.

OpenSSL uses by default a constant-time sliding-window exponentiation algorithm [2], an optimization of the square-and-multiply algorithm. Briefly, the exponent is divided in chunks of k bits, where k is the size of the window. At each iteration one chunk is processed, so, instead of considering one bit at a time as in the square-and-multiply, several bits are processed at once.

This algorithm requires the pre-computation of a table, that is later used for calculating the result. Indexes to access this table are chunks of the exponent. The pseudocode in Listing 3.1 describes a simplified version

[1]The target program used for this test is available at `https://github.com/tarequeh/DES`

[2]For additional details refer to the implementation of the `BN_mod_exp_mont_consttime` function in `openssl/crypto/bn/bn_exp.c` in the OpenSSL source code

of the sliding-window algorithm that we analyzed. Furthermore, OpenSSL uses as default the Chinese Remainder Theorem (CRT) to compute the result modulo p and q separately, to later combine them for obtaining the final result. For this reason we aim at finding two exponentiation operations during one encryption.

The result of the attack is shown in Figure 3.2. As a countermeasure against cache timing attacks discovered by C. Percival [47] is implemented, the precomputed values are not placed sequentially in the table. Basically, the table contains the first byte of every value one after each other, then the second byte and so on. Thus, for reading the i^{th} byte of the j^{th} precomputed value we need to access $table[i * window\_size + j]$. As we are interested in getting the index of the value that is being accessed we can just consider the offset of the first byte of the value, as highlighted in the picture. For ease of demonstration we used a very short RSA key (128 bits). In this case the window size is 3, so we leak 3 bits of the key at every access of the array. If we convert these indexes in binary and concatenate them, we obtain the private exponents d_p and d_q which in our example are $0x7c549e013545278b$ and $0x4af98ac085990e5$.

```
def exponentiate(a, p, n):  # compute a^p mod n
    winsize = get_winsize()  # in our test it is 3

    # Precomputation
    val = [1, a, a * a]
    for i = 3 .. 2^winsize - 1:
        val[i] = a * val[i-1]

    # divide p in chunks of winsize bits
    window_values = get_chunks(p, winsize)

    # length of p in bytes, divided by winsize and
    # rounded up to the next integer
    l = ceiling(byte_len(p) / winsize)

    # Square and multiply
    tmp = val[l-1]
    for i = l-2 .. 0:
        for j = 1 .. winsize:
            tmp = tmp * tmp % n
        tmp = tmp * val[window_values[i]] % n
    return tmp
```

Listing 3.1: OpenSSL's implementation of the sliding-window exponentiation.

This example demonstrated how visualization of memory accesses can reveal information about the execution and can be used in a similar way as

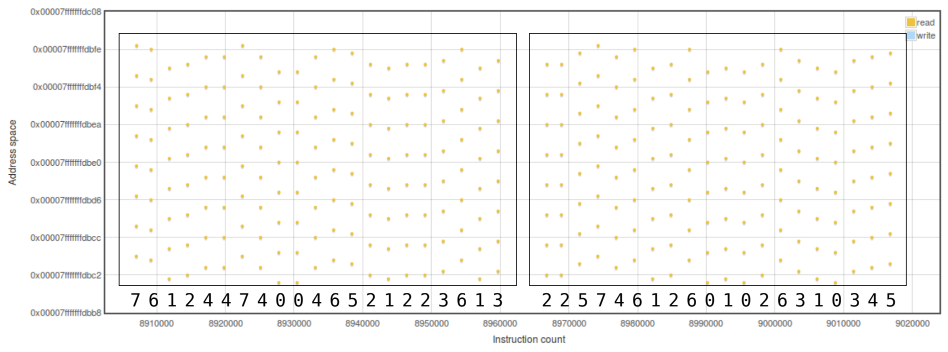


Figure 3.2: Memory accesses in the pre-computed tables used by OpenSSL during one RSA encryption. Locations of reads from this memory area leak the secret key. For demonstration purposes a very short key (128 bits) was used.

it is done with SPA in order to extract secret keys.

3.1.2 Data-flow tainting and diff of memory traces

Identifying which parts of the execution depend on our input can be helpful in order to isolate smaller parts of the code that will later be analyzed in detail. For achieving this goal we used two different techniques: data-flow tainting and *diff* of memory traces.

We based our work on tools offered by PANDA. It implements a tainting engine [48] that can be applied during replays of executions. It is architecture independent, thanks to the fact that it relies first on QEMU for binary translation and later on LLVM as an intermediate representation upon which the actual analysis is performed. Information-flow tainting offered by PANDA works at a byte level, it can be applied to different ISAs and does not require source code. As the literature regarding taint analysis is ample we will not present details here, we refer the reader to consult the work of Schwartz at al. [31].

In some cases, tainting is computationally expensive and due to state explosion it might not always be applicable. Moreover, in some tests the implementation offered by PANDA is requiring too much memory and thus the analysis can be unfeasible. As an alternative we propose the computation of the difference between memory traces, recorded with different inputs. Even though there are multiple implementation issues, it is a possible lightweight solution to the problem. However, there are some restrictions that we need to consider. First, we need to assume that the control flow of the program does not depend on the data, this is a valid assumption for many algorithms, cryptographic functions in particular. Second, we need the traces to be aligned: for achieving this goal the recorded trace needs to be filtered in order not to consider context switches, interferences with

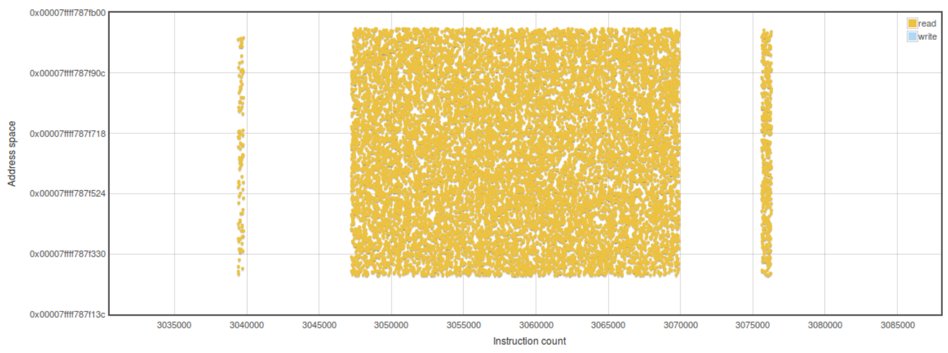


Figure 3.3: Identification of OpenSSL AES T-Tables by using *diff* of memory traces during encryption with different plaintexts.

other processes, operations in kernel space and I/O operations with variable time. We use a shadow instruction counter to normalize the trace and have it aligned. Also, when recording traces, the address space layout randomization (ASLR) features of the kernel need to be switched off, on the contrary the accessed memory locations would not match. For more details on the implementation refer to section 3.3. We experimented with different *diff* algorithms: visualizing accesses where the data differs, where the memory location differs or both.

An application of this technique is shown in Figure 3.3, obtained from the difference of two traces recorded during an AES encryption with OpenSSL with different plaintexts of the same length. In this case, by plotting memory accesses that differ in location, we clearly identify the *T-Tables* used in this AES implementation [49]. These tables are used for efficiency, they allow to perform and AES encryption by only leveraging XOR, shift and lookup operations. As indexes of these lookups are data-dependent and the rest of the computation does not differ in memory location the result is accurate.

By focusing on differences in the data content, it is possible to calculate the Hamming distance between the data-flow of two memory traces. This can be helpful, for instance, in detecting cryptographic operations and buffers containing ciphertext-related data. Two ciphertexts with different plaintexts and their intermediate values during the computation should be unrelated, thus their Hamming distance should be, on average, half of the bit-length of the data.

3.1.3 Entropy and randomness of the data-flow

Extending the work of Wang et al. [38] presented in section 2.3, we propose the use of statistical properties of the data-flow also to identify parts of the binary that deal with data with distinctive characteristics, not only to extract decrypted media streams. This is particularly useful for programs that involve cryptographic operations, such as DRM implementations. However, there are other possible use-cases for this approach, for example compression algorithms.

Entropy expresses the average amount of information that is contained in a specific data stream. We can conclude that encrypted or compressed data has very high entropy. On the contrary, a BMP image, a text or pointers to memory have lower entropy. We can then use this property to effectively locate parts of the code that deal with high-entropy data. In our experiments, we group the memory accesses in chunks of selectable length. For each chunk the probability distribution of each possible byte value, from $0x00$ to $0xFF$, is computed. Later the entropy level H is calculated with the following formula, where $P(x_i)$ is the frequency of each byte in the observed data.

$$H(X) = -\sum_i P(x_i) \log P(x_i).$$

One important property of encrypted data is that it is indistinguishable from random data, on the other hand compressed streams or other kind of data have bad randomness [50]. The Chi-Square test (χ^2) is one example of test that gives us an indication of how much the byte distribution in our data-stream is similar to a another distribution, in our case the uniform distribution. It is computed as follows, where O_i is the observed frequency and E_i the expected frequency of each byte:

$$\chi^2 = \sum_{i=1}^{n} \frac{(O_i - E_i)^2}{E_i}$$

It is worth noticing that other randomness tests can be used, the Chi-Square is just one possible solution. For our work we chose this algorithm as it was previously used for similar purposes, among others also by Wang et al. Moreover, it is a common and validated choice for randomness testing, its effectiveness was presented by L'Ecuyer in his research [51].

According to our observations, values of entropy of the data-flow during cryptographic algorithms have usually values close to 4.0 while the Chi-Square test returns values that are close to 1.0. On the other hand, while performing general purpose computations the values of entropy are usually around 2.0 while the Chi-Square test returns values in the range of thousands.

An application of this technique is shown in Figure 3.4. The target program is a reversing challenge from the security competition *Nuit du Hack 2015*. The binary is calling 7 times a function that decrypts the code of a second function using AES and later executes it. From the graph it is easily possible to identify the parts of the execution where the cryptographic operation takes place.

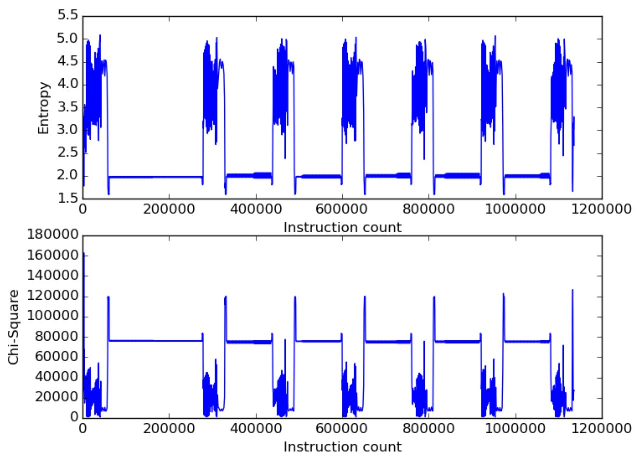


Figure 3.4: Data-flow entropy and randomness of the memory trace of a *crackme* from *Nuit du Hack Quals CTF 2015*. From the graph we can see that a cryptographic operation is performed 7 times.

In many cases, the visualization of entropy and randomness of the data flow can reveal patterns that enable the identification of distinctive operations performed by the code. We thus extend the observations of Wang et al. by using statistical properties of I/O, not only to identify peaks that could indicate the presence of cryptographic operations, but also to infer semantics of the code by using an SPA-like approach. An example is provided by Figure 3.5, which shows the entropy and randomness plots of the execution of the K-Means++ algorithm [3]. K-Means++ is a probabilistic clustering algorithm that is executed multiple times until a good solution is found. We run the test with 100 randomly generated points, in this case the function was executed 5 times, as it is possible to infer from the chart.

3.1.4 Auto-correlation of memory accesses

Auto-correlation, i.e. the cross-correlation of a sequence with itself at different points in time, is a common technique used in side-channel analysis of power traces. It is used to identify repeating patterns in time series of power consumption. In our project we adapted the same technique to work in the context of reverse engineering, in particular we applied auto-correlation to locations of memory accesses.

First of all, the memory trace needs to be transformed in a time series, on which we can apply the analysis. As we are interested in finding repeating patterns in the memory accesses, we consider locations that were

[3]The source code used in this test is available on *RosettaCode* at `http://rosettacode.org/wiki/K-means++_clustering`

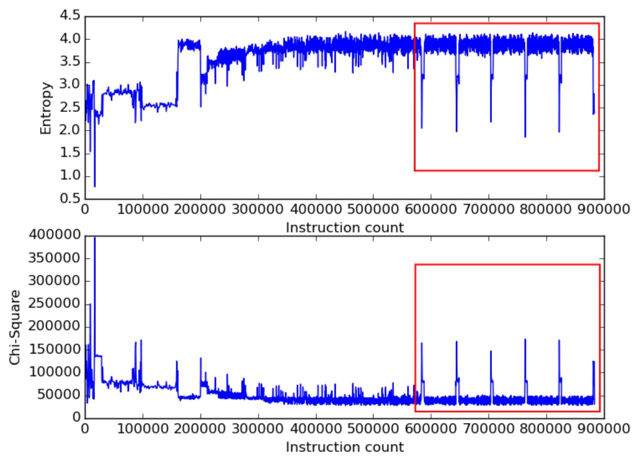


Figure 3.5: Data-flow entropy and randomness of memory accesses during an execution of the K-Means++ algorithm. The 5 iterations of the algorithm are highlighted in the graph.

accessed over time. This can reveal if computations with distinctive memory operations are performed multiple times. For distinctive operations we intend specific sequences of read or writes: an example could be a part of the program that sequentially accesses a buffer on the stack, then reads a word from the heap and eventually writes on the buffer on the stack. If this operation is repeated multiple times we would be able to identify patterns in the auto-correlation matrix computed from this sequence of memory accesses.

We compute the auto-correlation matrix P as follows:

$$P_{ij} = \frac{C_{ij}}{\sqrt{C_{ii} * C_{jj}}}$$

where C is the covariance matrix. Every C_{ij} indicates the level to which two variables x_i and x_j vary together. In our case every variable x_i is a chunk of the time series of adjustable length. Covariance $\sigma(X,Y)$ is defined as follows, where $\mathrm{E}[X]$ is the expected value of X.

$$\sigma(X,Y) = \mathrm{E}\left[(X - \mathrm{E}[X])(Y - \mathrm{E}[Y])\right]$$

We later display the auto-correlation matrix in a chart, where each value is represented by a dot with a color that varies from white (1.0, positive correlation) to black (-1.0, negative correlation).

An example of the application of this technique is given by Figure 3.6, which shows the auto-correlation matrix computed on the memory accesses in the whole address space during one AES128 encryption. It is possible to easily notice 9 repeating patterns that represent 9 rounds of the algorithm (the 10^{th} round is different from the others).

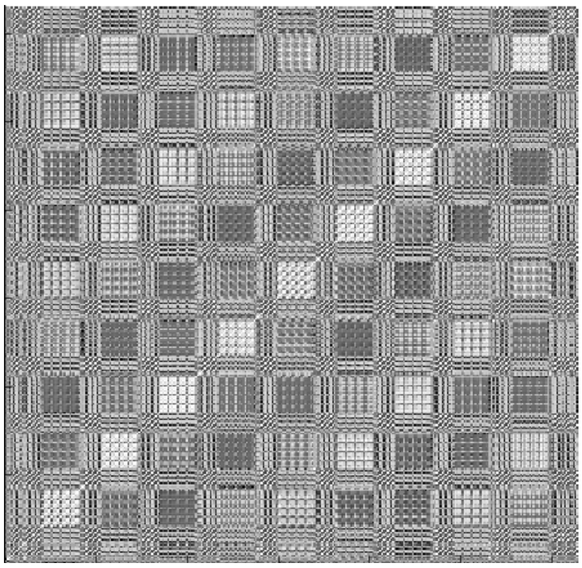


Figure 3.6: Auto-correlation matrix of memory accesses during one AES128 encryption. White corresponds to a correlation of 1.0 while black to -1.0.

3.2 Control-flow analysis methods

After obfuscation transformations, the control flow is often heavily modified in order to make static analysis more difficult. By recording concrete traces of the execution we intrinsically filter out all the dead/junk code and can only focus on the parts of the program that were actually executed, at the expense of not reaching complete coverage of the possible execution paths. We also don't have to deal with deductions of values of opaque predicates as they are computed during the execution. Even though for the general case we should perform multiple recordings in order to achieve a reasonable degree of coverage, when analyzing cryptographic implementations one or very few traces are often enough as the control-flow of a cryptographic function should not depend on the input data (i.e.: there should not be conditional branches that depend on confidential data), as this would leak information.

The rationale behind this kind of analysis is the assumption that even though the original control-flow of the program is transformed, there are still some patterns in the execution trace that remain. Some examples are multiple executions of parts of the code (caused by loops) or distinctive sequences of blocks that are run one after each other. We will first introduces methods to visualize these patterns while later techniques to counter control-flow obfuscation will be discussed.

```
A
B
A
B
[...]
A
B
A
C
```

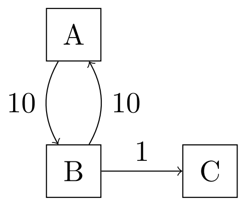


Figure 3.7: Example of visualization of the execution trace. Labels on the edges represent the number of edges between the two blocks.

3.2.1 Visualizing the execution trace

In order to visualize the collected data for the analyst, a graph is built from the execution trace. This graph is a subset of the control-flow graph (CFG) of the original binary, considering only executed parts of it. This directed graph $G(V, E)$ is such that the nodes are the basic blocks in the execution trace while edges represent a transition from one basic block to another (i.e.: if there is an edge from $block_a$ to $block_b$ it means that $block_b$ was executed after $block_a$). More formally, the graph is composed as follows:

$$V = \{basic\_block \mid basic\_block \ \in \ execution\_trace\}$$

$$E = \{\{block_a, block_b\} \mid block_b \ follows \ block_a \ in \ the \ execution\_trace\}$$

This allows us to visualize sequentiality of basic blocks. Moreover the number of edges between two blocks highlights which parts of the binary are executed multiple times and thus allows to identify loops.

Figure 3.7 shows an example of visualizing a sequence of blocks that were executed one after the other. It is possible to notice that blocks A and B are part of a loop that was executed 10 times while C is the block that is executed after the loop.

3.2.2 Analysis of the execution graph for countering control-flow flattening

A common obfuscation technique for camouflaging the CFG of a program is control-flow flattening. This technique is very effective as it radically changes the shape of the control-flow graph. Other techniques like control indirection or function splitting/merging do not significantly change the execution flow: in fact if we only consider basic blocks and we build a graph as described in the previous section, we would obtain very similar results with an obfuscated and non-obfuscated binary. These methods make static analysis harder as the concept of function in the binary becomes less related to the one of

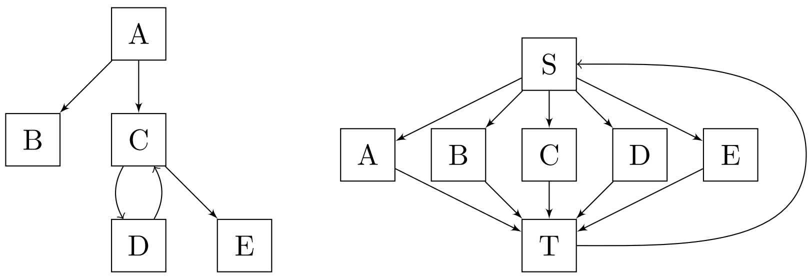


Figure 3.8: Original and flattened control-flow graph.

functions in the original code, however this can be defeated with dynamic analysis. On the other hand, control-flow flattening makes the execution graph completely different from the one of the original code: sequentiality of basic blocks is obfuscated by the state variable and it is really difficult to statically get information about the original semantics.

The rationales behind our decision to counter control-flow flattening are diverse: firstly, it is a widespread technique in commercial tools, secondly, it is effective in obfuscating the CFG and, thirdly, it is computationally lighter than other other transformations, such as virtualization, and thus it is applicable in different contexts. Furthermore, in literature it is possible to find various efforts in de-obfuscation of control-flow flattening [7, 52]. Our work differs from other proposed techniques as it is only based on graph analysis and does not require processing of the program code.

An example of control-flow flattening is given by Figure 3.8: as it is possible to notice, from the obfuscated graph only it is not possible to obtain any information of the original program flow. Instead, the logic and sequentiality of execution is hidden in the "artificial" blocks S and T and in the last instructions of blocks A to E.

Through the rest of the document we will refer to block S as the *dispatcher block*, block T as the *pre-dispatcher block* while A to E as *relevant blocks*. For our goals we focus, first of all, in categorizing each block using graph analysis. Later we reconstruct the original flow graph with the support of the execution trace.

Types of control-flow flattening

Control-flow flattening can be implemented in different ways and we need to adjust our techniques in order to better support different cases. In this section we will present common solutions implemented in the obfuscators we analyzed.

The solution proposed by C. Wang [14] in the paper where control-flow flattening was first introduced is based on *switch* statements. The resulting graph will be similar to the one in Figure 3.8, where the switch statement is

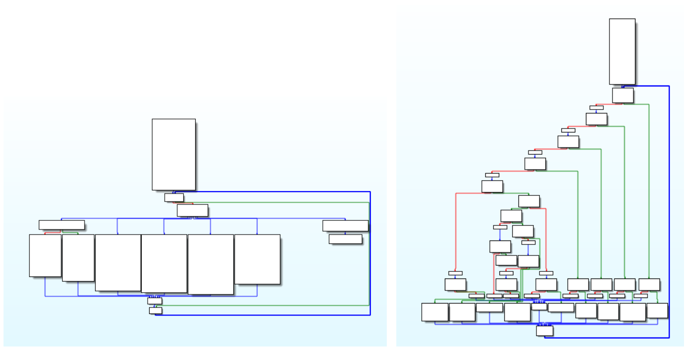


Figure 3.9: The same program obfuscated with two different implementations of control-flow flattening. The graph on the left shows a switch-based flattening while the one on the right is if-else-based.

compiled in the dispatcher block. A register is used as index in a jump table to reach the correct relevant block. All relevant blocks would then set this register to point to the location in the table with the address of the following relevant block to execute. Relevant blocks point to the pre-dispatcher that would redirect the execution flow according to the jump table. A variation of this technique consists in the creation of a function for each relevant block and an array of pointers to these functions, a register is used as index for this array. However, after compilation the switch statement gets converted in code similar to the one based on indirect calls. Also, instead of using a register, a variable in main memory can be used.

This technique (switch-based or indirect call-based) is implemented in Tigress as well as in other commercial obfuscators.

Another possibility is to use *if-else* constructs and a local variable or a register as state. A cascade of if-else performs multiple checks on the state variable and leads to the correct relevant block. This technique is available in the OLLVM obfuscator.

Optimizations or different implementations of this technique can lead to binaries with a slightly different structure. For example, as the pre-dispatcher always directly jumps to the dispatcher sometimes these two blocks are merged. Also, if multiple relevant blocks set the state variable to the same value, these logic can be isolated in a different block and make the multiple relevant blocks to point there before reaching the pre-dispatcher.

In Figure 3.9 it is possible to notice the difference between an obfuscation based on switch statements and one based on if-else constructs. As we can see the resulting CFGs are really different, nevertheless similar techniques can be applied for de-obfuscation.

Un-flattening the CFG

The first step in the un-flattening technique that we developed is to identify the dispatchers and the relevant blocks. For achieving this goal techniques of graph analysis are used. Different algorithms were implemented according to the type of obfuscation that was applied to the binary. In case of variations or custom implementations of control-flow flattening the analysis needs to be adjusted, for this reason these algorithms are included as plugins to the project. For handling special corner cases, new ones can be developed and applied to the analysis.

The key observations for identifying interesting blocks are the following:

- The dispatcher block has a high number of outgoing edges (high out-degree)
- The pre-dispatcher block has a high number of incoming edges (high in-degree)
- The pre-dispatcher has only one outgoing edge to the dispatcher (unless they are merged in the same block)
- In the case of a switch-based flattening, relevant blocks are all blocks in between the dispatcher and the pre-dispatcher. In the case of an if-else-based flattening, relevant blocks are only the parents of the pre-dispatcher (the depth in the parent search can depend on the obfuscation, we might need to consider more than the direct parent of the pre-dispatcher)

Using this remarks we developed different algorithms for supporting various kinds of obfuscations. Example pseudocode is presented in Listing 3.2.

Once the basic blocks are categorized, our approach consists in filtering the execution trace by removing the entries related to the dispatching introduced by the obfuscation. In this way only relevant blocks remain and the original execution flow is reconstructed. In summary, the de-obfuscation algorithm works as follows:

- Generate a graph from the execution trace as presented in 3.2.1
- Apply graph analysis to find dispatcher and relevant blocks (the analysis depends on the flattening technique)
- Filter the trace by considering only relevant blocks when executing blocks between the dispatcher and the pre-dispatcher
- Generate a de-obfuscated graph from the filtered execution trace

```
def categorize_blocks_ifelse(graph):
    predispatcher = node with highest indegree
    dispatcher = node with highest outdegree
    relevant_blocks = set()

    # consider as relevant only parents of the predispatcher
    for block in predispatcher.parents():
        relevant_blocks.add(block)

    return dispatcher, predispatcher, relevant_blocks


def categorize_blocks_switch(graph):
    predispatcher = node with highest indegree
    dispatcher = node with highest outdegree

    relevant_blocks = set()

    # consider as relevant all the blocks between the
    # dispatcher and the predispatcher
    for path in paths(from=dispatcher, to=predispatcher):
        for block in path:
            relevant_blocks.add(block)

    return dispatcher, predispatcher, relevant_blocks
```

Listing 3.2: Pseudocode for identifying dispatcher, pre-dispatcher and relevant blocks with different control-flow flattening implementations.

It is worth remarking that only techniques of graph analysis were employed, without the need to actually understand the semantics of the code. This is a great advantage as the code for manipulating the control flow can be arbitrarily complex, however once again the side-channel information leaked by the sequence of blocks executed one after the other reveals information of the original semantics of the code.

Clearly, we cannot obtain the same basic blocks of the original CFG as other transformations in the binary code or computations of the state variable might remain. Nevertheless, with this technique it is possible to reconstruct sequentiality of basic blocks, loops and branches. Moreover, our implementation also offers integration with static analysis tools such as *IDA Pro* to help analysts in better understanding the semantics of the code.

An example is shown in Figure 3.10: the target is a Base64 encoding function, whose CFG was flattened using Tigress. The figure shows the original, the obfuscated and the recovered CFGs using the proposed technique.

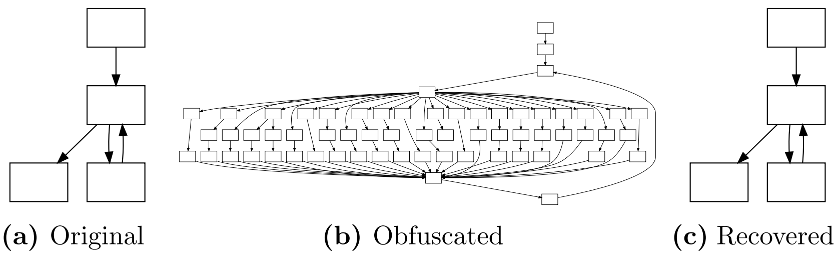


Figure 3.10: Example of control-flow flattening and recovery of the original CFG of a Base64 encoding implementation.

3.3 Implementation

The software developed for this research project can be decoupled in different parts: firstly, of all traces of memory accesses and executed instructions need to be recorded. Secondly, those traces need to be stored in such a way that it is possible to query them efficiently and apply filtering or aggregation. Thirdly, we need an interface to enable the analyst to interactively browse the collected data, filter it and use it in combination with other tools. Lastly, an analysis framework accessible from the interface is essential to apply transformations on the data, process it and extract insightful information.

Acquisition of memory and execution traces For our research we based the acquisition of memory and execution traces on PANDA (Platform for Architecture-Neutral Dynamic Analysis) [37], a project resulted from the collaboration of MIT Lincoln Laboratory, Georgia Tech and Northeastern University. PANDA is built on top of the QEMU system emulator and it consists of a framework that allows to instrument it with custom plugins. It supports different architectures: i386, x86_64 and ARM (also with Android support).

When analyzing protected software we need to often face anti-debugging techniques. Those can help the binary detect if it is running in a sandbox or a debugging environment and act differently, or even refuse to execute. If there are just few simple checks we can patch the code to skip them, however if integrity checks or thousands of anti-debugging conditions are in place this can be particularly tedious. For these reasons it would be very helpful to create controlled execution environments that are indistinguishable from a normal execution for the program itself. PANDA, by offering instrumentation at the emulator level, allows us to not interfere with the userland and evade common anti-debugging tricks like checks for *ptrace* on Unix systems or *IsDebuggerPresent* on Windows.

Clearly, this is not an infallible solution: some binaries can use different checks to detect the fact of being run in an emulator. One example for ARM programs is to check the behavior of the data and instruction caches, which on ARM architectures are separated. If the host is running on an Intel architecture they are unified and thus the emulated caches behave differently compared to the real ones. Moreover, system-level instrumentation is more challenging as we need to isolate the process that we want to analyze from the others and detect and handle context switches from kernel and user spaces. However, we consider this approach as a reasonable trade-off between resilience to anti-debugging and complexity.

Even though PANDA supports record and replay of executions, only non deterministic input/output is recorded in order to optimize traces for size. Furthermore, it records at a system level, without having the notion of processes. As we are interested in all memory accesses and instructions that are executed by one single process, we developed ad-hoc plugins for our goals: one for recording memory traces and one for execution traces. In order to isolate the target process from others running on the same system we use the ASID (Address-Space Identifier) register, as it determines the location of the page table associated to a process and it is univocally corresponding to it. To handle jumps to kernel space correctly, recording is paused whenever the user-space of our target program is left and resumed afterwards. For performance and interoperability reasons we chose to store the output of the plugins in the Google Protobuf format.

It is worth remarking that the recording tools are not coupled with other parts of the project. As long as the traces are in the correct format they can be recorded with other platforms as well, if desired also with tools that operate at a user-level.

Datastore In order to efficiently query the traces, they are moved in a Postgresql database. Each trace is saved in a different table. Common data that is present in every trace and needs to be indexed (e.g.: instruction count, current pc, etc.) is saved in a normal column. Other attributes, whose schema could change from trace to trace, the JSON datatype offered by Postgresql is used. As data does not need to be modified after import in the database, every column is indexed to make querying more efficient.

Interface and analysis tools The interface is a web application developed using Python and the Django framework. The GUI was built using HTML5 and Javascript. A REST API is used to communicate with the datastore and query and retrieve the traces. Analysis scripts can be interactively invoked from the interface on the selected part of the trace. The Python libraries *Scipy* and *Numpy* are used as a support for running the analysis.

CHAPTER 4

Evaluation

4.1 Introduction of the benchmarks

The evaluation of the techniques proposed in this report is based on tests carried out on sample programs, obfuscated with publicly available obfuscators. We decided to rely on publicly available obfuscators for reproducibility of our work and for the unrestricted access to these tools.

For our experiments we employed OLLVM by HEIG-VD and Tigress by the University of Arizona. This choice was guided by the fact that these two obfuscators were recently released and thus implement state-of-the-art and modern techniques, they are highly configurable and flexible. Furthermore, in recent literature we can find other research outcomes based on these two tools, such as [53, 54].

Regarding the target programs, our tests were carried out on implementations of common cryptographic algorithms. This is due to the fact that the main application of our research is the analysis of heavily obfuscated DRM solutions, which are often based on widely tested algorithms such as AES or DES. Moreover, we target programs of which we also have the original source code, in order to have the possibility to compare the original behavior of the program with the obfuscated one. This is essential to evaluate how much information is leaked from the side-channels that we are considering and how well the techniques we propose reveal this information.

The programs used in our experiments are the following:

- An implementation of AES128 that encrypts 4 blocks of data.
- A program encrypting 2 blocks of data using DES.

- An HMAC-SHA1 implementation that computes one message authentication code.

4.1.1 Obfuscators configuration

OLLVM

Obfuscator-LLVM offers three main features for the obfuscation of a target binary:

- Instruction Substitution: replaces standard binary operations (e.g.: addition, division, bitwise operations, etc.) with semantically equivalent but more complicated sequences of instructions. This feature adds diversity in the resulting binary thanks to random constants added to the code.
- Bogus Control Flow: when applied to a basic block, it modifies the control flow by adding a spurious block that conditionally jumps either to the legitimate basic block or to junk code. The condition is an opaque predicate that always results in a jump to the original block, but its result is difficult to determine statically. It applies by default to 30% of basic blocks. It also adds diversity to the binary as the dead code is randomly generated.
- Control-flow flattening: it fully flattens the control flow graph by using a sequence of conditional statements.

OLLVM integrates with *clang* and works at the LLVM intermediate representation level. For this reason the output of the obfuscator is either LLVM bitcode or a binary. For our experiments, OLLVM version 3.5 was used. The command used to compile the test programs, that enables all the available obfuscation techniques, is the following:

```
$OLLVM/bin/clang -mllvm -sub -mllvm -fla -mllvm -bcf
                  source.c -o obfuscated_bin
```

Tigress

Tigress offers a wide variety of obfuscation techniques, for our experiments we apply common general-purpose features. In detail, the following transformations are employed:

- EncodeLiterals: substitutes integer and string literals with opaque expressions that evaluate to the original value at run-time.
- EncodeArithmetic: obfuscates arithmetic operations with more complex but functionally equivalent ones. For each operation there are

multiple encodings available, one of them is selected randomly creating diversity in the output.

- AddOpaque: splits the control-flow by adding conditional jumps based on opaque predicates. At run-time the evaluation of the predicate will always result in a jump to the original code, however dead code is added to make static analysis more complex. Tigress offers different possibilities for generating junk code: generation of a buggified version of the original code, calls to random or non-existing functions or insertion of randomly generated assembly.

- AntiTaintAnalysis: disrupts taint analysis tools that rely on dynamic analysis. It uses two different methods to copy a value, such that it is more complex for the engine to detect the taint propagation. The first method is to use a for-loop and increment the destination variable according to the original value, while the second consists in a bit-by-bit copy of the original value using if-constructs.

- Flatten: control-flow flattening using different techniques. Switch-based and indirect call-based flattening methods are used in our tests. Tigress also offers a goto-based flattening transformation, however in that case the code is flattened only at the source code level. When compiled, gotos are converted to unconditional jumps so the shape of the CFG stays unchanged. This obfuscation is only effective in thwarting decompilers, so it was not considered in our tests.

Tigress is a code-to-code obfuscator. For this reason, the output is still C source code that needs to be compiled. For our tests we use *gcc 4.8.2* on *Ubuntu Linux* with Tigress version 2 (Purple Nuple). The command used to obfuscate the target source code is the following:

```
tigress --Transform=InitEntropy --Functions=main
        --Transform=InitOpaque --Functions=main
        --Transform=EncodeLiterals --Functions=*
        --Transform=EncodeArithmetic --Functions=*
        --Transform=AddOpaque --Functions=*
        --Transform=AntiTaintAnalysis --Functions=*
        --Transform=Flatten --Functions=*
        --Transform=CleanUp
        --out=obfuscated_source.c source.c
```

4.1.2 Data-flow analysis evaluation benchmark

The effectiveness and the resilience of the proposed data-flow de-obfuscation techniques are evaluated by comparing the results of the analysis on the original and the obfuscated targets. In this way we can better understand how much information is revealed by the behavior of the program, in terms of memory accesses. Moreover, we can validate the hypothesis that some patterns in the memory reads and writes are preserved after obfuscation transformations and allow us to infer the original semantics of the code.

For every target binary different tests are presented: statistical analysis of the I/O in terms of entropy and randomness and results of auto-correlation analysis. These two methods are both based on memory accesses, however they take into account very different aspects of the same data. The former is based on the analysis of the content of the data-flow. The latter instead only considers the locations of reads and writes in memory.

4.1.3 Control-flow unflattening evaluation benchmark

To evaluate the proposed de-obfuscation techniques against control-flow flattening, different experiments were carried out. Sample programs are obfuscated using OLLVM and Tigress obfuscators. For Tigress, the switch-based and indirect call-based flattening methods is used. Later, a similarity score between the execution flow graph of the original code and the one of the obfuscated code is computed. For completeness also a similarity score between the original and the obfuscated graph is shown.

In literature we can find different approaches to quantify similarity between two CFGs. In particular, Udupa et al. [7], in their work regarding control-flow de-obfuscation, measure the number of spurious edges in the CFG that were added by the obfuscator and later deleted by the de-obfuscator. They also consider false positives and false negatives as a separate measure of the error. On the other hand, Yadegari et al. [55] propose a different approach, based on the computation of an approximation of the edit distance between the two graphs. This benchmark takes into account spurious nodes that are added by the obfuscation transformation, as well as edges. In detail, the similarity algorithm computes the number of vertexes and edges that need to be added or removed to transform one graph in the other. The computation of the graph edit distance is an NP-hard problem, for this reason an approximation proposed by Hu et al. [56] is used.

For our experiments we opted for the graph edit distance as a measure of similarity between the original and the obfuscated execution graphs. This decision is supported by the work of Chan et al. [57], that compared different algorithms for computing the similarity of control flow graphs. The algorithm proposed by Hu et al. was shown to be the most accurate.

To compare CFGs of different sizes the distance value is normalized according to the total size of the graphs being compared, with the following formula, where $\delta(G_1, G_2)$ is the edit distance and $|G|$ is the size (i.e.: number of nodes and edges) of graph G.

$$sim(G_1, G_2) = 1 - \frac{\delta(G_1, G_2)}{|G_1| + |G_2|}$$

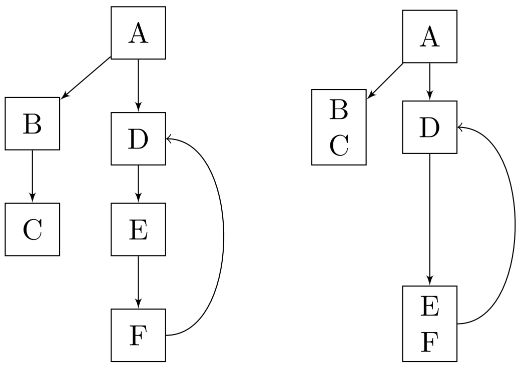


Figure 4.1: Example of graph normalization. The graph on the left is the original one while the one on the right is normalized.

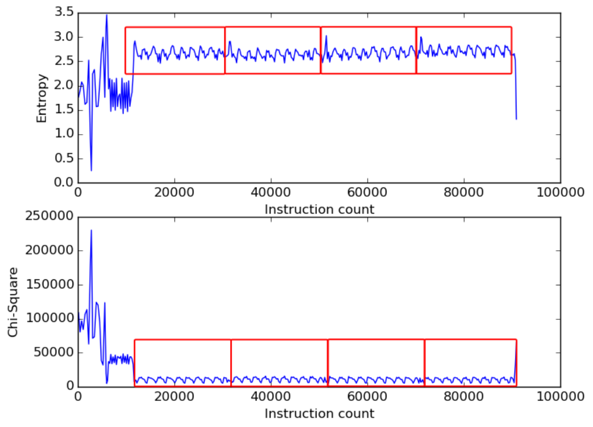
As obfuscation tools add spurious code that make the binary more complex, it can happen that the same logic that is contained in one basic block in the original code is spread in multiple subsequent basic blocks in the obfuscated code. However, this is not relevant for the matter of comparing the shape of the CFG. For this reason, before computing the similarity score, the graphs are normalized in such a way that basic blocks that are always unconditionally executed one after the other are merged in the same block. In detail, every node that has only one parent and no siblings is merged with the parent. An example is given by Figure 4.1.

4.2 Data-flow recovery results

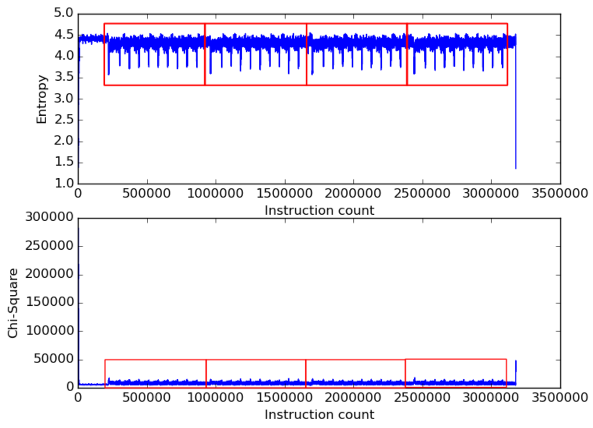
From the graphs regarding statistical analysis of I/O (Figures 4.2, 4.3 and 4.4) we can gather interesting information on the underlying business logic of the targets. For AES128 it is possible to clearly notice that the program is executing the encryption algorithm 4 times. For each run, the 10 rounds of AES128 are identifiable. This is possible for the data gathered for the non-obfuscated target, as well as for obfuscated ones. Regarding DES, we can easily see from the graph that the encryption function was executed two times, each composed by 16 rounds. Additionally, for both AES128 and DES, we can infer that the part of the chart before the encryption functions includes the key scheduling algorithm.

For the HMAC-SHA1 target, we can see 4 repeated patterns with a particular shape. This is due to the fact that the compression function of the algorithm is called 4 times, for one HMAC. The following formula summarizes the computation of an HMAC, given an hash function H, where *opad* and *ipad* are XOR masks, K is the secret key and m the message.

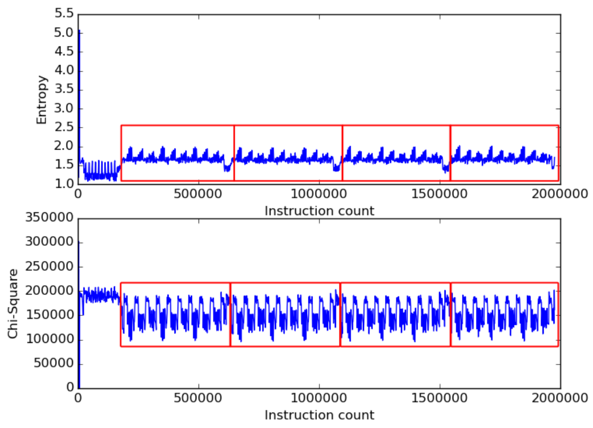
$$HMAC(K, m) = H\left((K \oplus opad) | H((K \oplus ipad)|m)\right)$$



(a) Original



(b) OLLVM



(c) Tigress

Figure 4.2: AES128: Entropy and randomness of the data flow

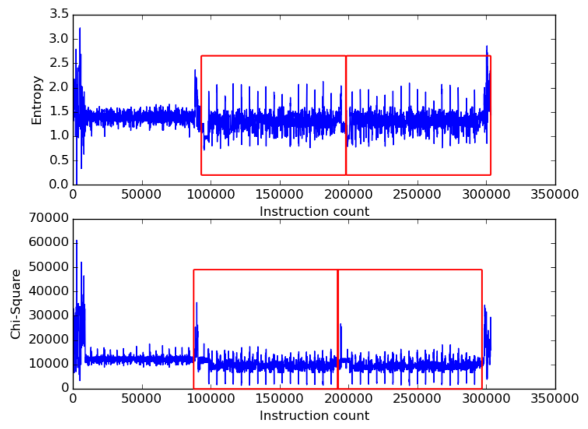
Thus, for the first hash $K \oplus ipad$ is first added to the internal state, then m is concatenated to it. Later a second hash is computed with $K \oplus opad$, then the result of the previous operation is added. This makes 4 calls to the SHA1 update function for one HMAC computation, which is what we can see from the charts.

It is peculiar that OLLVM makes the identification of the repeated patterns even easier, compared to the original binary. This can be caused by spurious instructions are added to the program, for obfuscating the underlying logic. The increasing number of memory accesses helps in making distinctive patterns standing out.

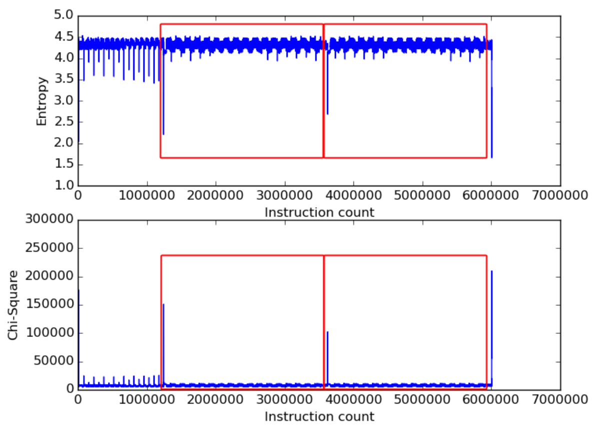
The absolute values of entropy and randomness change very much from the original program to the obfuscated ones. This can be caused for instance by the computation of opaque predicates, control-flow flattening dispatching instructions or decoding of literals. Even though spurious operations add significant noise, patterns still remain and can be easily identified. From this results we can notice that OLLVM highly contributes in increasing the levels of entropy and randomness, because of the usage of values with this characteristics in the computation of opaque predicates and for instruction substitution. On the other hand, in all our tests, the obfuscation layers added by Tigress contribute in lowering entropy and randomness levels, possibly because of different values used for encoding of literals or opaque predicates. For future research, these obfuscator-dependent characteristics could be exploited for initial reconnaissance in a black-box context, for advancing hypothesis on the obfuscator that was employed.

The results of the auto-correlation analysis (Figures 4.5, 4.6 and 4.7) match very well with the outcomes of the study of entropy and randomness of the data-flow. This is a very interesting result as auto-correlation only takes into account the location of memory accesses without looking at the content of the data that was read and written. On the contrary, statistical analysis of I/O only considers the data that was exchanged, without taking into account locations of buffers.

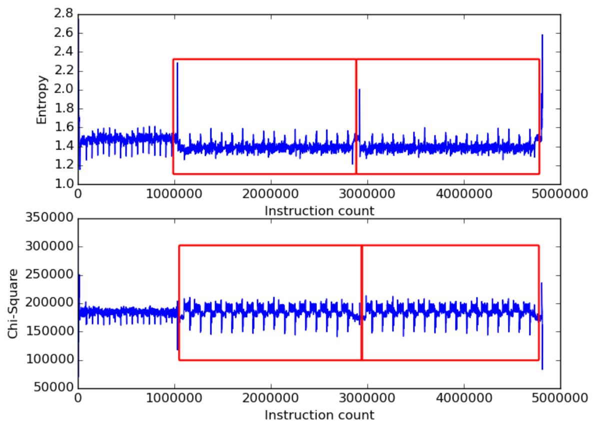
Regarding AES128 we obtained very good results, as the 4 executions of the encryption function are very clear and the 10 rounds of each run are distinguishable. Curiously, the auto-correlation matrix resulted from the execution of the target obfuscated with OLLVM is even more clear than the one obtained from the non-obfuscated binary. Again, this can be due to the additional computation introduced by the obfuscation, that make the memory accesses dependent on the actual logic of the program prevalent on memory operations made by library functions or other less relevant accesses (e.g.: zeroing of buffers, saving of register values on the stack or management of heap structures). Moreover, obfuscation often forces the usage of memory for saving intermediate results, on the other hand, for the original binary, an heavier use of registers can be preferred by the compiler for performance reasons.



(a) Original



(b) OLLVM

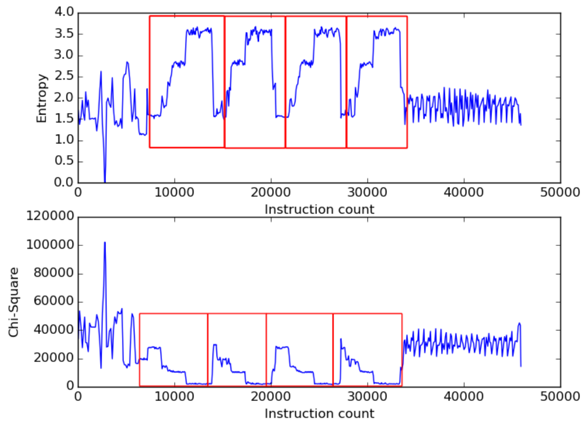


(c) Tigress

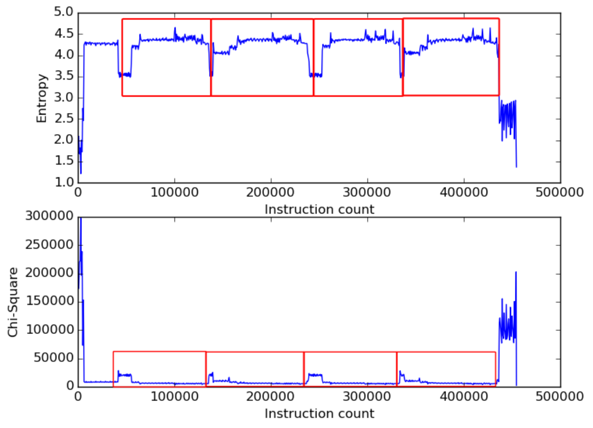
Figure 4.3: DES: Entropy and randomness of the data flow

In the case of DES, using auto-correlation the key scheduling algorithm part becomes evident. From the figures, it is possible to distinguish a square in the bottom-left corner that represents it. Again, the two encryption functions, each one with 16 rounds, are distinguishable. In this test, the results obtained from obfuscated targets are noisier, compared to the non-obfuscated target. However, it is still possible to recognize patterns.

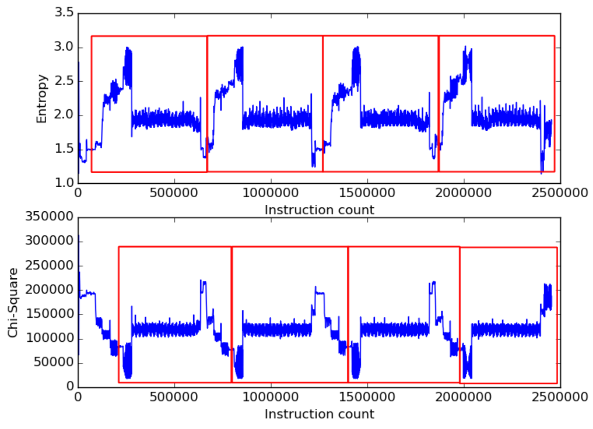
In the third experiment, we can see that part of the information revealed from the auto-correlation matrix of the original target is not observable in the obfuscated ones. In the former, it is possible to notice that there are 4 repeated operations, nonetheless they differ. In the obfuscated programs we can still recognize 4 repeated operations, however they look all very similar. Tigress in this case managed to make the results of auto-correlation significantly noisy. The squares representing the SHA1 compression functions are considerably smaller and the rest of the image shows little correlation. Despite this, the patterns are still identifiable.



(a) Original

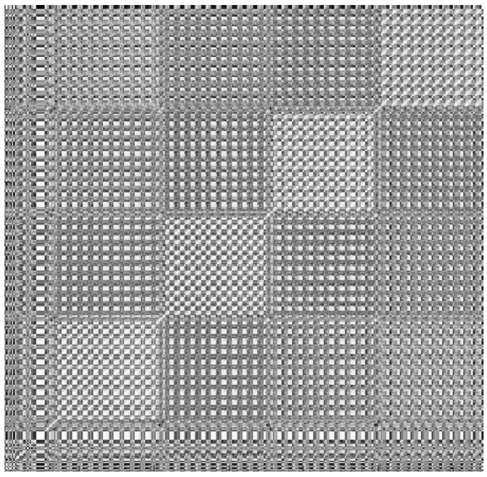


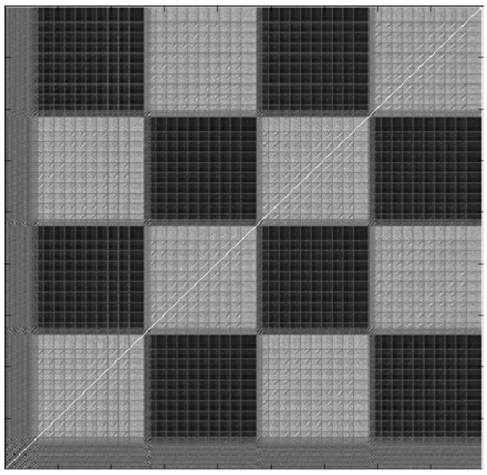
(b) OLLVM



(c) Tigress

Figure 4.4: HMAC-SHA1: Entropy and randomness of the data flow


(a) Original


(b) OLLVM

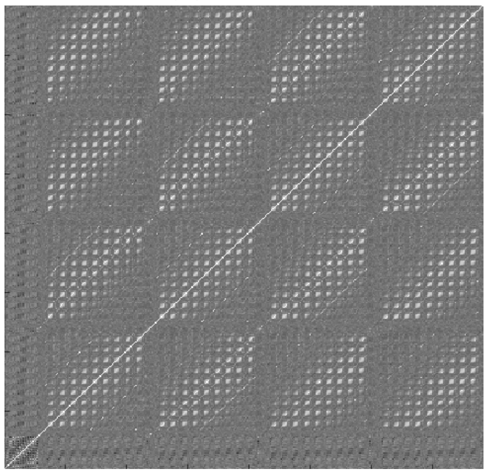
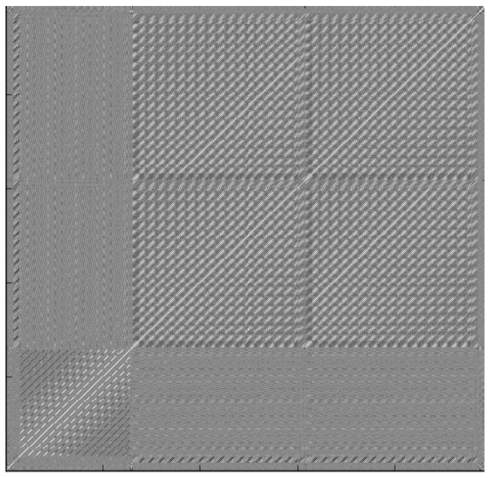
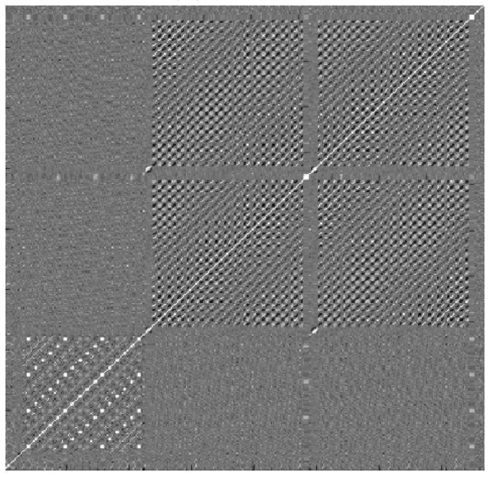

(c) Tigress

Figure 4.5: AES128: Autocorrelation of memory accesses


(a) Original


(b) OLLVM

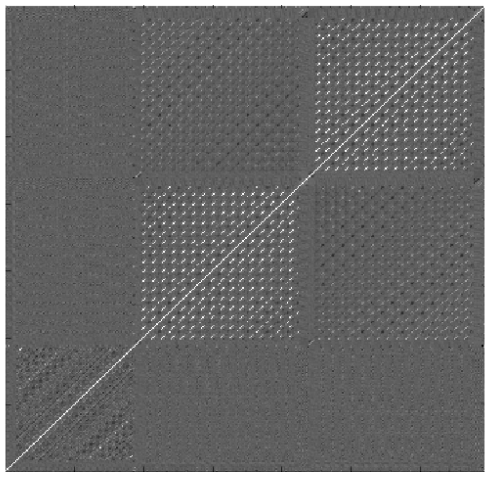
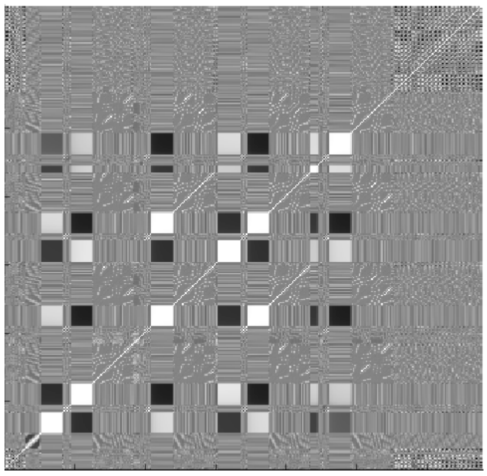
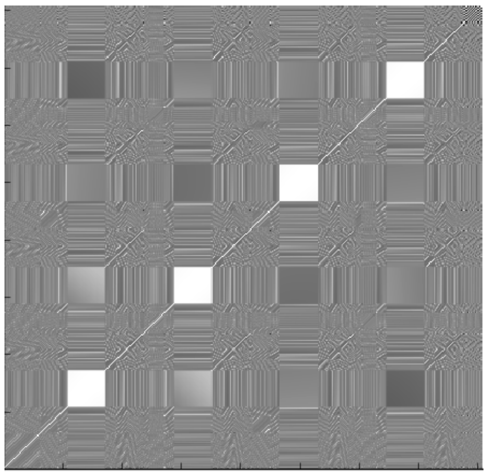

(c) Tigress

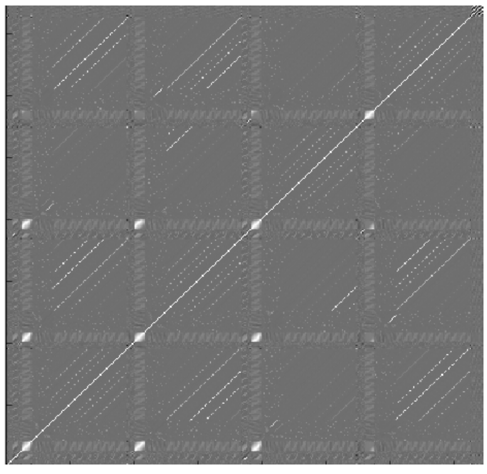
Figure 4.6: DES: Autocorrelation of memory accesses



(a) Original



(b) OLLVM



(c) Tigress

Figure 4.7: HMAC-SHA1: Autocorrelation of memory accesses

4.3 Control-flow recovery results

In order to evaluate the resilience of our approach against changes to the control-flow, we also consider compiler optimized versions of the target programs. These optimizations can cause the merging of basic blocks and thus modify the structure of the code outputted by the obfuscator. To enable them we added the compilation flag "-O3" to the commands of *clang* in the case of OLLVM or *gcc* in the case of Tigress. We use this additional test to show that our graph-based analysis is resilient to these changes. However, it is worth remarking that in a real-world scenario it is not desirable to apply compiler optimizations to obfuscated code: some transformations such as constant unfolding or instruction substitutions could be simplified and the protection offered by the obfuscation layers could be compromised.

The results of our experiments are shown in Table 4.1. Every target program was obfuscated using different features of Tigress and OLLVM, as presented in 4.1.1. For every obfuscated binary, the table shows the similarity scores between the obfuscated CFG and the original one, as well as the similarity score between the recovered CFG and the original one.

Our graph analysis-based approach has shown to produce high-quality results in recovering the original control flow from obfuscated target programs, even with very diverse control-flow flattening techniques. In few cases we obtained a lower similarity score, this is due to the fact that compilation optimizations or other obfuscation methods tweaked the control flow of the program, for example by unrolling loops or by transforming a for-loop into a do-while loop. Nevertheless, even in those cases an analysis of the CFG revealed structures of the original code. In some tests with optimized indirect-based control flow flattening we could not apply our algorithm, as the obfuscation was removed by the compiler and the resulting binary was not actually flattened. In fact, the similarity score between the obfuscated and the original CFG is considerably high. In the HMAC-SHA1, even though the compiler removed part of the obfuscation, the CFG was still partially flattened so we could still apply our techniques.

These tests show that valuable information can be extracted only by using a trace of execution, that records the sequence in which basic blocks are executed. Compared to other studies, the proposed method only relies on graph analysis and is generally applicable to different control flow flattening obfuscations, without the need to reverse engineer the code or parse it in any way. This is a considerable advantage as our analysis is more lightweight, resilient against any obfuscation of the basic blocks and not dependent on a specific architecture or ISA.

AES128		
	Obfuscated	Recovered
OLLVM	0.011	1.000
OLLVM -O3	0.054	0.995
Tigress switch	0.249	1.000
Tigress switch -O3	0.342	1.000
Tigress indirect	0.314	1.000
Tigress indirect -O3	0.670	N/A

DES		
	Obfuscated	Recovered
OLLVM	0.020	1.000
OLLVM -O3	0.046	1.000
Tigress switch	0.174	1.000
Tigress switch -O3	0.245	1.000
Tigress indirect	0.228	0.789
Tigress indirect -O3	0.670	N/A

HMAC-SHA1		
	Obfuscated	Recovered
OLLVM	0.046	0.817
OLLVM -O3	0.111	0.963
Tigress switch	0.143	0.956
Tigress switch -O3	0.255	0.968
Tigress indirect	0.237	0.934
Tigress indirect -O3	0.604	0.933

Table 4.1: Similarity scores of obfuscated and de-obfuscated CFGs, compared to the original one.

4.4 Analysis of shortcomings

We presented different techniques that exploit side-channel information produced by the execution of the target program, for the purpose of reverse engineering. For recording this information there is the basic assumption that we can execute the binary in a controlled environment, such as a full-system emulator like PANDA or a userspace DBI (e.g.: Intel Pin, Miasm, etc.). In case the target employs some countermeasures to obstruct this operation, we need to first address those protections and find a way to overcome them before running our analysis. This is a common problem for every approach based on dynamic analysis, therefore we will not discuss further this issue. Different advances in this area can be found in literature.

Another limitation is given by the code coverage of concrete traces. As we mostly rely on dynamic analysis of one execution, only one execution path is considered. Even though this could be enough in many cases, especially when considering cryptographic functions, it can often be a problem. As a further enhancement of our work, symbolic execution could be applied to obtain complete code coverage and explore every possible branch. Also, fuzzers (e.g. *afl-fuzz*) could be used in order to produce inputs that trigger interesting code branches. Later, recording the execution with these inputs can lead to the desired results.

In the case of large binaries or programs that perform a huge amount of memory accesses or instructions it might not be desirable to record a full trace of the execution, as it would be impractical to process because of its size. It is possible to overcome this problem by narrowing down the focus of the trace to a reduced part of the execution or to specific memory areas. Regarding memory accesses, for instance, it might be possible to restrict the recording to accesses on the heap as the analyst suspects that the cryptographic operations use mostly that memory area. In other cases it might be possible to reduce the trace by filtering out operations performed by dynamically loaded libraries and only consider code from the binary itself, or even parts of it. Another possible solution would be to aggregate the data on-the-fly during the execution, if possible. This would allow for instance to compute partial results of the analysis during execution, instead of recording a full trace that would be later processed.

To further confuse the analysis and complicate the reverse engineering, binaries can be compiled to extensively use registers and don't hit the memory very often. This would cause our techniques based on memory accesses to fail, however the issue is easily solvable as we can add register changes to the trace by enhancing the recoding engine. In our tests we did not conducted experiments in this direction, as the use of registers is not a widespread obfuscation method and it can be easily overcome for our purposes.

Müller et al. proposed TRESOR [58], a patch for the Linux kernel for running disk encryption without storing the keys in RAM and using Intel

AES extensions of the CPU. This approach is resilient against cold-boot attacks for countering disk encryption. However, in the context of DRM solutions, it is not effective as it can be easily defeated by running the target in an emulator, hooking the implementation of the encryption instructions and thus extracting the secret keys.

CHAPTER 5

Conclusions

Software protections can be a valuable and effective mean of securing intellectual property or create a robust DRM solution. On the other hand, they can also be used for hide malicious activities in malware. For the purpose of evaluating the level of security offered by these protections or unveiling harmful code, it is worthwhile to analyze them and test their resilience.

Reverse engineering of obfuscated software is a very challenging task that requires outstanding skills and can be extremely time consuming. In this report we presented a novel methodology to collect high-level information from the execution of a program, in order to provide the analyst with an insightful overview of the logic and to narrow down the study to small portions of the code. Instead of basing our analysis on the understanding of the disassembled binary, we study the behavior of the target program. This behavior is well described by the sequence of instructions that are executed after each other and by accesses to memory. This information is collected using dynamic analysis techniques and later these traces are processed using different methods.

For the study of memory accesses, we propose interactive visualizations to assist the analyst in finding patterns in the execution. Furthermore, the use of data tainting or the computation of the difference between traces can be used to find which parts of the program are dependent on user input and thus restrict the scope of the analysis. To collect information from the content exchanged to and from the memory, we propose the analysis of statistical properties of the data-flow. Similarly, the information about location of memory accesses alone can give valuable information, as shown with results of auto-correlation analysis.

Regarding instruction traces, CFG reconstruction methods and visualization of the executed basic blocks are introduced. We also show how to process this data in order to counter a common obfuscation technique: control flow flattening. The proposed method is only based on graph analysis techniques and allows to recover the original control-flow graph from different implementations of flattening obfuscation, without the need to analyze the code of the target program.

An evaluation and discussion of the proposed techniques was presented. We show how the analysis of behavior of executions is effective for the purpose of unveiling the original semantics of the program. We also showed the resilience of these methods by testing them with different settings of state-of-the-art obfuscators.

From these results we can affirm that current obfuscation techniques are not effective in hiding side-channel information that is revealed with the methods proposed in this report. From our evaluation we managed in all tests to extract insightful information from the samples and, even though some noise was present, patterns were clearly identifiable. It is also remarkable that in our tests we did not perform any pre-processing or filtering of the trace, analysis were run on the full execution. With additional knowledge about the target logic and a more accurate selection of specific memory and time regions it would be possible to extract even more accurate results.

We also argue that concealing the behavior of a program is a hard and computationally expensive task. Regarding statistical analysis of the data flow, it is really hard to mask these properties as pointers have intrinsically less entropy and randomness, compared to other data. Even if all literals and intermediate values are encoded to there would still be a considerable difference between those randomized values and pointers. A similar observation applies to the analysis of location of memory accesses. In some cases it might be possible to reorder operations that read or write into memory in order to disguise repeated operations. This is however not applicable in the general case.

Concealing the original semantics of the control-flow graph is an easier task. It can be hardened with different techniques: for example by duplicating pieces of code, to hide operations that repeat, or by adding useless operations. This is unfortunately not always possible: it can be unpractical in terms of performance and size of the binary.

5.1 Future work

The analysis of behavior of obfuscated code proposed in this report has its limitations, as discussed in section 4.4. As a first step in this area we obtained promising results, however further work can be done. In this section we propose different directions for future research.

Regarding execution traces and CFG recovery, further developments are needed in order to provide de-obfuscation methods for other techniques than control-flow flattening, such as virtualization. It would also be interesting to apply graph isomorphism techniques in order to automatically understand which algorithm is being executed. Moreover, possibly detect if it is derived from common implementations, such as the OpenSSL codebase. In 2015, Lestringant et al. worked in this direction by applying graph isomorphism techniques to data-flow graphs [59].

For the analysis of memory accesses we suggest to expand the visualization methods proposed in this report, as well as the experimentation with different techniques for identifying patterns. In particular, it would be interesting to test the effectiveness of 3D interactive visualization, possibly supported by virtual reality appliance as proposed by Stefan et al. [60].

In addition, more tests should be carried out in order to improve the proposed techniques. We suggest to analyze binaries produced by different obfuscators in order to further test the robustness and resilience of the proposed methods.

Finally, in this work we highlighted weaknesses of state-of-the-art obfuscators. In different commercial applications, software obfuscation is crucial for the economical sustainability of the firm. We believe there is the need to further investigate in developing obfuscation methods that are robust against behavior analysis. Considerable work in this direction was done for the hardware world in order to prevent side-channel analysis, similar efforts are needed also for software.

Bibliography

[1] Julie E Cohen and Mark A Lemley. Patent scope and innovation in the software industry. *California Law Review*, pages 1–57, 2001.

[2] T. Grimm. Reverse engineering is criminal. *Time-Compression Technologies*, 2004.

[3] Alorie Gilbert. Attack targets Sony ’rootkit’ fix. *CNET News*, 2005.

[4] Boaz Barak, Oded Goldreich, Rusell Impagliazzo, Steven Rudich, Amit Sahai, Salil Vadhan, and Ke Yang. On the (im) possibility of obfuscating programs. In *Advances in Cryptology—CRYPTO 2001*, pages 1–18. Springer, 2001.

[5] Mila Dalla Preda. *Code obfuscation and malware detection by abstract interpretation*. PhD thesis, PhD thesis, Dipartimento di Informatica, University of Verona (February 2007), 2007.

[6] Christian Collberg, Clark Thomborson, and Douglas Low. A taxonomy of obfuscating transformations. Technical report, Department of Computer Science, The University of Auckland, New Zealand, 1997.

[7] Sharath K Udupa, Saumya K Debray, and Matias Madou. Deobfuscation: Reverse engineering obfuscated code. In *Reverse Engineering, 12th Working Conference on*, pages 10–pp. IEEE, 2005.

[8] Jean-Marie Borello and Ludovic Mé. Code obfuscation techniques for metamorphic viruses. *Journal in Computer Virology*, 4(3):211–220, 2008.

[9] Ilsun You and Kangbin Yim. Malware obfuscation techniques: A brief survey. In *BWCCA*, pages 297–300, 2010.

[10] Cullen Linn and Saumya Debray. Obfuscation of executable code to improve resistance to static disassembly. In *Proceedings of the 10th ACM conference on Computer and communications security*, pages 290–299. ACM, 2003.

[11] Anirban Majumdar, Clark Thomborson, and Stephen Drape. A survey of control-flow obfuscations. In *Information Systems Security*, pages 353–356. Springer, 2006.

[12] Julian Bangert, Sergey Bratus, Rebecca Shapiro, and Sean W Smith. The page-fault weird machine: Lessons in instruction-less computation. In *WOOT*, 2013.

[13] William Zhu, Clark Thomborson, and Fei-Yue Wang. Applications of homomorphic functions to software obfuscation. In *Intelligence and Security Informatics*, pages 152–153. Springer, 2006.

[14] Chenxi Wang. *A security architecture for survivability mechanisms.* PhD thesis, University of Virginia, 2001.

[15] W Brecht. White-box cryptography: hiding keys in software. *NAGRA Kudelski Group*, 2012.

[16] Stanley Chow, Phil Eisen, Harold Johnson, and Paul C Van Oorschot. A white-box DES implementation for DRM applications. In *Digital Rights Management*, pages 1–15. Springer, 2003.

[17] Stanley Chow, Philip Eisen, Harold Johnson, and Paul C Van Oorschot. White-box cryptography and an AES implementation. In *Selected Areas in Cryptography*, pages 250–270. Springer, 2003.

[18] Brecht Wyseur, Wil Michiels, Paul Gorissen, and Bart Preneel. Cryptanalysis of white-box DES implementations with arbitrary external encodings. In *Selected Areas in Cryptography*, pages 264–277. Springer, 2007.

[19] Louis Goubin, Jean-Michel Masereel, and Michaël Quisquater. Cryptanalysis of white box DES implementations. In *Selected Areas in Cryptography*, pages 278–295. Springer, 2007.

[20] Chih-Fan Chen, Theofilos Petsios, Marios Pomonis, and Adrian Tang. Confuse: LLVM-based Code Obfuscation.

[21] Axel Souchet. Obfuscation of steel: meet my Kryptonite.

[22] University of Applied Sciences and Arts Western Switzerland of Yverdon-les Bains (HEIG-VD). Obfuscator-LLVM. `http://o-llvm.org`.

[23] C Collberg. The Tigress Diversifying C Virtualizer. `http://tigress.cs.arizona.edu/`.

[24] Morpher. Crypt the script. `http://morpher.com/`.

[25] Arxan Technologies Inc. Application protection. `https://www.arxan.com/products/application-protection/`.

[26] whiteCryption. Code protection. `http://www.whitecryption.com/code-protection/`.

[27] Yannis Smaragdakis and Christoph Csallner. Combining static and dynamic reasoning for bug detection. In *Tests and Proofs*, pages 1–16. Springer, 2007.

[28] Yoann Guillot and Alexandre Gazet. Automatic binary deobfuscation. *Journal in computer virology*, 6(3):261–276, 2010.

[29] Monirul Sharif, Andrea Lanzi, Jonathon Giffin, and Wenke Lee. Automatic reverse engineering of malware emulators. In *Security and Privacy, 2009 30th IEEE Symposium on*, pages 94–109. IEEE, 2009.

[30] Mark Weiser. Program slicing. In *Proceedings of the 5th international conference on Software engineering*, pages 439–449. IEEE Press, 1981.

[31] Edward J Schwartz, Thanassis Avgerinos, and David Brumley. All you ever wanted to know about dynamic taint analysis and forward symbolic execution (but might have been afraid to ask). In *Security and Privacy (SP), 2010 IEEE Symposium on*, pages 317–331. IEEE, 2010.

[32] International Secure Systems Lab. Anubis - Malware Analysis for Unknown Binaries. `http://anubis.iseclab.org/`.

[33] QuarksLab. DRM obfuscation versus auxiliary attacks. `http://recon.cx/2014/schedule/events/44.html`.

[34] Robert S Boyer, Bernard Elspas, and Karl N Levitt. SELECT—a formal system for testing and debugging programs by symbolic execution. *ACM SigPlan Notices*, 10(6):234–245, 1975.

[35] Koushik Sen. Concolic testing. In *Proceedings of the twenty-second IEEE/ACM international conference on Automated software engineering*, pages 571–572. ACM, 2007.

[36] George Hotz. qira. `http://qira.me/`.

[37] Brendan F Dolan-Gavitt, Josh Hodosh, Patrick Hulin, Tim Leek, and Ryan Whelan. Repeatable Reverse Engineering for the Greater Good with PANDA. 2014.

[38] Ruoyu Wang, Yan Shoshitaishvili, Christopher Kruegel, and Giovanni Vigna. Steal This Movie: Automatically Bypassing DRM Protection in Streaming Media Services. In *USENIX Security*, pages 687–702, 2013.

[39] Brendan Dolan-Gavitt. Breaking Spotify DRM with PANDA. `http://moyix.blogspot.it/2014/07/breaking-spotify-drm-with-panda.html`.

[40] Brendan Dolan-Gavitt, Tim Leek, Josh Hodosh, and Wenke Lee. Tappan zee (north) bridge: mining memory accesses for introspection. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 839–850. ACM, 2013.

[41] M. Zalewsky "lcamtuf". Pulling JPEGs out of thin air. `http://lcamtuf.blogspot.it/2014/11/pulling-jpegs-out-of-thin-air.html`.

[42] Gregory Conti, Erik Dean, Matthew Sinda, and Benjamin Sangster. Visual reverse engineering of binary and data files. In *Visualization for Computer Security*, pages 1–17. Springer, 2008.

[43] Daniel A Quist and Lorie M Liebrock. Visualizing compiled executables for malware analysis. In *Visualization for Cyber Security, 2009. VizSec 2009. 6th International Workshop on*, pages 27–32. IEEE, 2009.

[44] Philipp Trinius, Thorsten Holz, Jan Gobel, and Felix C Freiling. Visual analysis of malware behavior using treemaps and thread graphs. In *Visualization for Cyber Security, 2009. VizSec 2009. 6th International Workshop on*, pages 33–38. IEEE, 2009.

[45] Stefan Vomel and Hermann Lenz. Visualizing indicators of rootkit infections in memory forensics. In *IT Security Incident Management and IT Forensics (IMF), 2013 Seventh International Conference on*, pages 122–139. IEEE, 2013.

[46] James B Baum. Windows memory forensic data visualization. Technical report, DTIC Document, 2014.

[47] Colin Percival. Cache missing for fun and profit, 2005.

[48] Ryan Whelan, Tim Leek, and David Kaeli. Architecture-independent dynamic information flow tracking. In *Compiler Construction*, pages 144–163. Springer, 2013.

[49] PUB FIPS. 197, Advanced Encryption Standard (AES), National Institute of Standards and Technology, US Department of Commerce (November 2001). *Link in: http://csrc. nist. gov/publications/fips/fips197/fips-197. pdf.*

[50] Weiling Chang, Binxing Fang, Xiaochun Yun, Shupeng Wang, and Xiangzhan Yu. Randomness testing of compressed data. *arXiv preprint arXiv:1001.3485*, 2010.

[51] Pierre L'Ecuyer. Testing random number generators. In *Winter Simulation Conference*, pages 305–313, 1992.

[52] Matias Madou, Bertrand Anckaert, and Koen De Bosschere. Code (de) obfuscation. *Advanced Computer Architecture and Compilation for Embedded Systems (ACACES 2005)*, pages 291–294, 2005.

[53] Sebastian Banescu, Martın Ochoa, and Alexander Pretschner. A framework for measuring software obfuscation resilience against automated attacks.

[54] Yuichiro Kanzaki, Akito Monden, and Christian Collberg. Code Artificiality: A Metric for the Code Stealth Based on an N-gram Model.

[55] Babak Yadegari, Brian Johannesmeyer, Benjamin Whitely, and Saumya Debray. A generic approach to automatic deobfuscation of executable code. Technical report, Technical report, Department of Computer Science, The University of Arizona, 2014.

[56] Xin Hu, Tzi-cker Chiueh, and Kang G Shin. Large-scale malware indexing using function-call graphs. In *Proceedings of the 16th ACM conference on Computer and communications security*, pages 611–620. ACM, 2009.

[57] Patrick PF Chan and Christian Collberg. A Method to Evaluate CFG Comparison Algorithms. In *Quality Software (QSIC), 2014 14th International Conference on*, pages 95–104. IEEE, 2014.

[58] Tilo Müller, Felix C Freiling, and Andreas Dewald. Tresor runs encryption securely outside ram. In *USENIX Security Symposium*, pages 17–17, 2011.

[59] 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*, pages 203–214. ACM, 2015.

[60] Stefan Marks, Javier E Estevez, and Andy M Connor. Towards the holodeck: fully immersive virtual reality visualisation of scientific and engineering data. In *Proceedings of the 29th International Conference on Image and Vision Computing New Zealand*, page 42. ACM, 2014.