UNIVERSITAT DE BARCELONA

**Treball final de grau**

**GRAU DE MATEMÀTIQUES**
**GRAU D'ENGINYERIA INFORMÀTICA**

**Facultat de Matemàtiques i Informàtica**
**Universitat de Barcelona**

---

# Code deobfuscation by program synthesis-aided simplification of Mixed Boolean-Arithmetic expressions

---

**Autor: Arnau Gàmez i Montolio**

| | |
|---|---|
| **Directors:** | **Prof. Raúl Roca Cánovas** |
| | **Dr. Antoni Benseny Ardiaca** |
| | **Dr. Mario Reyes De Los Mozos** |
| **Realitzat a:** | **Departament de Matemàtiques i Informàtica** |
| | **Fundació Eurecat** |

**Barcelona,** **21 de juny de 2020**

# Abstract

This project studies the theoretical background of Mixed Boolean-Arithmetic (MBA) expressions as well as its practical applicability within the field of code obfuscation, which is a technique used both by malware threats and software protection in order to complicate the process of reverse engineering (parts of) a program.

An MBA expression is composed of integer arithmetic operators, e.g. $(+, -, *)$ and bitwise operators, e.g. $(\wedge, \vee, \oplus, \neg)$. MBA expressions can be leveraged to obfuscate the data-flow of code by iteratively applying rewrite rules and function identities that complicate (obfuscate) the initial expression while preserving its semantic behavior. This possibility is motivated by the fact that the combination of operators from these different fields *do not interact well together*: we have no rules (distributivity, factorization...) or general theory to deal with this mixing of operators.

Current deobfuscation techniques to address simplification of this type of data-flow obfuscation are limited by being strongly tied to syntactic complexity. We explore novel program synthesis approaches for addressing simplification of MBA expressions by reasoning on the semantics of the obfuscated expressions instead of syntax, discussing their applicability as well as their limits.

We present our own tool *r2syntia* that integrates *Syntia*, an open source program synthesis tool, into the reverse engineering framework *radare2* in order to retrieve the semantics of obfuscated code from its Input/Output behavior. Finally, we provide some improvement ideas and potential areas for future work to be done.

2010 Mathematics Subject Classification. 68R01, 03B70, 08A70, 68Q42, 68Q55

# Resum

Aquest projecte estudia el rerefons teòric de les expressions Mixtes Booleanes-Aritmètiques (MBA) així com la seva aplicació pràctica en el camp de l'ofuscació de codi, una tècnica usada tant per les amenaces de programari maliciós (malware) com pels sistemes de protecció de programari, per tal de complicar el procés d'enginyeria inversa sobre la totalitat (o parts) d'un programari.

Una expressió MBA està formada per operadors aritmètics sobre enters, per exemple $(+, -, *)$ i operadors bit a bit, per exemple $(\wedge, \vee, \oplus, \neg)$. Les expressions MBA es poden aprofitar per ofuscar el flux de dades del codi aplicant iterativament regles de reescriptura i identitats de funcions que compliquen (ofusquen) l'expressió inicial, al mateix temps que es preserva el seu comportament semàntic. Aquesta possibilitat està motivada pel fet que la combinació d'operadors d'aquests dos camps diferents *no interactuen gaire bé*: no tenim regles (distributivitat, factorització...) o una teoria general per tractar amb aquests operadors barrejats.

Les tècniques actuals de desofuscació de codi per tractar la simplificació d'aquest tipus d'ofuscació del flux de dades estan limitades pel fet d'estar fortament lligades a la complexitat sintàctica. Explorem nous enfocaments basats en síntesi de programes per tal d'adreçar la qüestió de simplificar expressions MBA a partir d'un raonament al voltant del comportament semàntic de les expressions ofuscades, en lloc de centrar-nos en la seva representació sintàctica, tot discutint l'aplicabilitat i limitacions d'aquests.

Presentem la nostra pròpia eina *r2syntia* que integra *Syntia*, una eina de codi obert per síntesi de programes, en el programari d'enginyeria inversa *radare2* per tal d'obtenir la semàntica de codi ofuscat a partir del seu comportament d'Entrada/Sortida. Finalment, proposem algunes possibles millores i diferents àrees d'interès per a un treball futur.

# Resumen

Este proyecto estudia el trasfondo teórico de las expresiones Mixtas Booleanas-Aritméticas (MBA) así como su aplicación práctica en el campo de la ofuscación de código, una técnica usada tanto por las amenazas de programas maliciosos (malware) como por los sistemas de protección de programas, con el objetivo de complicar el proceso de ingeniería inversa sobre la totalidad (o partes) de un programa.

Una expresión MBA está formada por operadores aritméticos sobre enteros, por ejemplo $(+, -, *)$ y operadores bit a bit, por ejemplo $(\wedge, \vee, \oplus, \neg)$. Las expresiones MBA se puede aprovechar para ofuscar el flujo de datos del código aplicando iterativamente reglas de reescritura e identidades de funciones que complique (ofusquen) la expresión inicial, al mismo tiempo que se preserva su comportamiento semántico. Esta posibilidad está motivada por el hecho que la combinación de operadores de estos dos campos diferentes *no interactúan demasiado bien*: no tenemos reglas (distributividad, factorización...) o una teoría general para tratar con estos operadores mezclados.

Las técnicas actuales de desofuscación de código para tratar la simplificación de este tipo de ofuscación de flujo de datos están limitadas por el hecho de estar fuertemente ligadas a la complejidad sintáctica. Exploramos nuevos enfoques basados en síntesis de programas para poder abordar la cuestión de simplificar expresiones MBA a partir de un razonamiento alrededor del comportamiento semántico de las expresiones ofuscadas, en lugar de centrarnos en su representación sintáctica, así discutiendo su aplicabilidad y limitaciones propias.

Presentamos nuestra propia herramienta *r2syntia* que integra *Syntia*, una herramienta de código abierto para síntesis de programas, con el programa de ingeniería inversa *radare2* con el objetivo de obtener la semántica de código ofuscado a partir de su comportamiento de Entrada/Salida. Finalmente, proponemos algunas posibles mejores y diferentes áreas de interés para un trabajo futuro.

# Contents

# Introduction

Code obfuscation is the process of transforming an input program $P$ into a functionally equivalent program $P'$ which is harder to analyze and to extract information than from the initial program $P$.

We find two clearly differentiated areas in which code obfuscation is commonly and widely used: malware threats and (commercial) software protection. The desired technical outcome is the same for both cases: complicate the process of reverse engineering the final product (software) and therefore difficult the understanding of the workings and intention of initial code. However, the motivation can deeply vary. On the one hand, malware threats leverage obfuscation in order to hide malicious payloads and increase the total time being undetected. On the other hand, commercial software protection is usually intended to protect intellectual property and prevent illegal distribution of non-registered or non-licensed copies. Thus, code obfuscation is an important part of any modern Digital Rights Management technology solution.

Many different code obfuscation techniques exist with their own particularities. Nevertheless, the general idea is as follows: mess with program's control-flow and/or data-flow at different abstraction levels (source code, compiled binary, intermediate representation) on different target units (whole program, function, basic block). It is important to note that different techniques can be mixed together to increase the complexity of the resulting obfuscated code in an even more unpredictable way.

Code deobfuscation is the process of transforming an obfuscated (piece of) program $P'$ into a (piece of) program $P''$ that is easier to analyze than $P'$. Ideally, we would like to have $P'' \approx P$, where $P$ represents the original non-obfuscated program code. This is rarely possible to guarantee, mainly because the analyst doing the deobfuscation process almost never has access to original code to check against. Moreover, usually the analyst is just interested in some specific parts of the program rather than the whole program. The analyst might also be interested in understanding the code rather than reconstructing a functional binary.

State-of-the-art code deobfuscation techniques rely on symbolic execution, taint analysis and a combination of them. Symbolic execution is a technique that lets the analyst transform the control-flow and data-flow of the program into symbolic expressions. Taint

analysis is a technique that lets the analyst know at each program point what part of memory or registers are controllable by the user input.

These techniques have been shown to be promising to address control-flow based obfuscation, in which we need to check the satisfiability of the obtained symbolic boolean condition. However, when analyzing data-flow based obfuscation (like Mixed Boolean-Arithmetic or virtualized handlers behavior), we are interested in finding a simpler semantically equivalent expression rather than checking for its satisfiability. We find that these techniques are heavily dependent on the syntactic complexity of the code being analyzed. Thus, an adversary might thwart the analysis capabilities by arbitrarily increasing the syntactic complexity of the obfuscated code.

In order to overcome the scalability issues of increased syntactic complexity and, specifically, to be able to address data-flow based code obfuscation techniques, we would like to be able to reason about the semantics of the code instead of syntax. Some work has been recently done in that direction. Mainly, trying to incorporate program synthesis techniques to the deobfuscation process in order to synthesize the semantics of a particular snippet of code, presumably obfuscated.

Although there has been some progress recently, there is still a huge lack of both theoretical foundations and practical tools to address the question of simplifying/deobfuscating generic Mixed Boolean-Arithmetic expressions.

## Report organization

In Chapter 1 we provide foundational context and a general overview of basic code obfuscation techniques, introducing different metrics and models to measure the *quality* of a code obfuscation transformation. We also present some important aspects to consider when dealing with the task of code deobfuscation, both from an academic and practical standpoint, while showcasing the differences between them.

In Chapter 2 we delve into the study of Mixed Boolean-Arithmetic expressions (MBA) and their application to data-flow code obfuscation, presenting some of the most important academic results in the field so far. We also introduce several metrics to assess the *complexity* of MBA expressions, as well as discussing different theoretical simplification approaches.

Chapter 3 is devoted to introducing the field of program synthesis. In particular, we dig into different oracle-guided inductive program synthesis approaches and their application to code deobfuscation. We present current state-of-the-art research addressing code deobfuscation by simplification of MBA expressions using program synthesis and discuss the limitations of this kind of approaches.

Finally, in Chapter 4 we present *r2syntia*, a tool we have developed that integrates *Syntia*, an open source prototype implementing one of the discussed program synthesis approaches, within the reverse engineering framework *radare2*. We test our implementation and obtain satisfactory results, which are explained and compared against original implementation. Then, we provide some insights on potential improvements that our integration could implement in a near future.

A guided example recreating the process of obfuscating a snippet of *C* code, analyzing the obtained binary with radare2 and extracting its semantics (i.e. deobfuscating it) with r2syntia, can be found in Appendix A. The planning and methodology followed during the realization of this project are presented in Appendix B and Appendix C, respectively.

# Chapter 1

# Code obfuscation

## 1.1 Context

*Reverse engineering* is the process of analyzing a product based on its finished and distributed form. In this work, we will specifically talk about software (or code) reverse engineering. We will sometimes refer to it simply as *reversing*. The purpose of reverse engineering can deeply vary. Among others, we find:

- Describe a (proprietary) format or protocol to provide (open) implementations that guarantee interoperability.
- Find errors in software that can lead to a non-desired behavior of the analyzed system. This is usually known as vulnerability research. Such misbehavior can be leveraged to the favor of the attacker/analyst by *exploiting* it.
- Analyze malware samples in order to create signatures and defense mechanisms to prevent propagation.
- Bypass software protection and/or licensing mechanisms.

The motivation behind it can differ greatly as well. In this work, we will not discuss moral considerations, but only cover technical aspects of processes that relate to reverse engineering.

*Code obfuscation* is the process of transforming an input program $P$ into a functionally equivalent program $P'$ which is harder to analyze and to extract information than from $P$. We define obfuscation in the context of the technical protections against Man-At-The-End (MATE) attacks. The concept of MATE attacks represents the scenario where the attacker/analyst has an instance of the program and completely controls the environment where it is executed.

Thus, code obfuscation aims to complicate the process of reverse engineering, and is mainly used in the following domains:

- Malware: allows malicious software to avoid automatic signature detection by antivirus engines and slows down the analysis work of the reverser for further classification and detection.

- Protection of intellectual property: in commercial software, it allows to protect an algorithm or a protocol. We can take Skype [BD06] or Dropbox [KW13] as examples.

- Digital Rights Management: protect access to software with a license check, or to digital content.

We find different software solutions implementing obfuscation. On the one hand, we have academic obfuscators like Tigress [Col20], or O-LLVM [Jun+15]. On the other hand, we find many commercial software protection products (VMProtect[1], Epona[2], Themida[3]...) that leverage obfuscation within the protection mechanisms that they provide.

The general idea of code obfuscation is to apply a transformation to *mess* (complicate) the program's control-flow and/or data-flow at different *abstraction levels* (source code, compiled binary or an intermediate representation[4]) and affecting different *target units* (whole program, function, basic block[5] or instruction).

In the following section we present a few obfuscation techniques to illustrate different approaches that could be used for achieving a certain degree of protection from reverse engineering. It is by no means a complete or exhaustive listing. The interested reader can refer to [BP17] and [CN09] for further details, more obfuscation techniques and discussion about classification of obfuscation transformations. Also take into account that many *weak* techniques that are presented alone can be combined to create a *hard* obfuscation transformation.

## 1.2 Survey of obfuscation techniques

### 1.2.1 Data-flow based

#### Constant unfolding

We can think of constant unfolding as opposed to the compilation optimization technique known as constant folding, which replaces computations whose results are known at compile time with such results.

Think of the following *C*-like statement:

$$x = 2 * 3 + 23;$$

[1] `https://vmpsoft.com/`
[2] `https://quarkslab.com/epona/`
[3] `https://www.oreans.com/Themida.php`
[4] `https://en.wikipedia.org/wiki/Intermediate_representation`
[5] `https://en.wikipedia.org/wiki/Basic_block`

As the result of this assignment can be known at compile-time, it would be a waste to generate assembly code that computes each operation at run-time. Thus, the compiler can replace the assignment to

$$x = 29;$$

and generate assembly code for this *folded* expression instead.

Constant unfolding can be defined as the inverse process. That is, the obfuscator would *unfold* a certain constant into a computation process that produces it.

A common way to deal with this kind of obfuscation is to apply the same optimization techniques that compilers use. In this case, one could perform Reachable Definition Analysis[6] (forward data-flow analysis), which calculates the set of definitions that may potentially reach each program point.

**Dead code insertion**

Dead code is code that does not have any effect on the program's operation. Indeed, another common compiler optimization technique is known as dead code elimination.

To illustrate it, consider the following *C*-like function:

```
int f() {
    int x, y, z;
    x = 1;      // This assignment to x is dead
    y = 2;
    z = 3;      // z is dead, as it is not used again
    x = y + 4;  // Any x above is not live
    return x;
}
```

We can also discuss about code in the assembly level. Consider the following x86 assembly snippet of code:

```
mov eax, 1      ; This assignment to eax is dead
mov ebx, 2
mov ecx, 3      ; ecx is dead, as it is not used again
add ebx, 4
mov eax, ebx    ; Any occurrence of eax above is not live
```

Thus, we can define dead code insertion obfuscation technique as the process of deliberately inserting instructions that will not have any effect in the computations' outcome.

In order to deal with this kind of obfuscation one could perform Liveness Analysis[7] (backward data-flow analysis), which calculates the variables (in assembly, registers and memory locations) that are live at each point in the program.

[6] https://en.wikipedia.org/wiki/Reaching_definition

[7] https://en.wikipedia.org/wiki/Live_variable_analysis

**Encodings**

Encodings aim at preventing a specific value to appear in clear at any point of the program execution. They are composed of an encoding function $f(x)$ and its corresponding decoding function $f_{-1}(x)$.

Consider the encoding function $f(x) = x - 0x1234$, whose result $f(x)$ is pushed on the stack. Then, whenever the actual value $x$ is needed, the decoding function $f_{-1}(x) = x + 0x1234$ has to be applied beforehand. This case is exemplified in the following x86 assembly snippet of code:

```
...
sub eax, 0x1234             ; Apply encoding function to eax
push eax                    ; Push eax on the stack
...
add dword [esp], 0x1234     ; Apply decoding function to value on top of stack
...
pop ebx                     ; Retrieve decoded value into ebx register
```

Affine functions are very common in obfuscation encodings, as they are easily invertible and produce low performance overhead.

The main drawback on this technique is that constants have to be dynamically decoded at run-time before being processed. Even if it is possible to use encodings homomorphic to a certain operator (i.e., computations could be done on the encoded values), that would decrease the diversity of possible encodings.

**Pattern-based obfuscation**

The basic idea is to transform one or more adjacent instructions into a new sequence of instructions preserving the semantic behavior but being more complicated to analyze.

Consider the following x86 assembly instruction:

```
push eax
```

It could be converted to:

```
lea esp, [esp - 4]
mov dword [esp], eax
```

Now consider the first instruction from previous snippet:

```
lea esp, [esp - 4]
```

It could be converted to:

```
push ebx
mov ebx, esp
xchg [esp], ebx
pop esp
```

These kinds of pattern substitutions can be as complicated as desired, and can be applied iteratively an arbitrary number of times to complicate the initial code. Also note that we have illustrated the examples with the concrete registers `eax` and `ebx`, but these transformations would apply to any 32-bit register.

To deal with this kind of obfuscation, one can construct inverse pattern substitutions to map the target sequences into the original ones. However, manually extracting the necessary rewriting transformations can be very time-consuming.

### 1.2.2 Control-flow based

**Functions Inlining and Outlining**

- On the one hand, the technique of function inlining consists of replacing a call to a function by the body of the function itself.
- On the other hand, the technique of function outlining extracts a snippet of code into a new function that is called whenever the previous code was placed.

Both function inlining and outlining modify the Control Flow Graph[8] (CFG) of particular functions, as well as the Call Graph[9] (CG) of the whole program. Combining both techniques degenerates the logic that could be extracted or inferred from the original CG and the CFG of potentially interesting functions.

**Opaque predicates**

An opaque predicate is a specially crafted boolean expression $P$ that always evaluates to either true or false. That is, its value is known during obfuscation, but (should be) unknown for the analyst.

The typical use of an opaque predicate is to create fake branches in the CFG of a function. The branch that will always be followed contains the intended (original) code, while the other branch could contain junk code that will never execute.

Another variation is to consider a predicate $P$ that randomly evaluates to true or false. Then, each conditional branch should contain a different obfuscation transformation of the original code preserving its semantics, creating the illusion of a differentiated behavior between the branches.

A standard way to deal with opaque predicates is by using symbolic execution to obtain the predicates and solve them with the aid of a satisfiability modulo theories[10] (SMT) solver.

[8] `https://en.wikipedia.org/wiki/Control-flow_graph`
[9] `https://en.wikipedia.org/wiki/Call_graph`
[10] `https://en.wikipedia.org/wiki/Satisfiability_modulo_theories#Solvers`

#### Control flow flattening

The basic idea of control flow flattening is to change the structure of a function's CFG by replacing all control structures with a central and unique *dispatcher*, typically implemented as a switch-like statement. Then, each basic block is responsible of updating the dispatcher's context so it can link to the next basic block. The strength (and weakness) of this technique rests on the ability to conceal the context's manipulations and transitions.

### 1.2.3 Mixed data-flow and control-flow based

#### VM-based obfuscation

Virtual Machine (VM) based obfuscation is a rather advanced technique, and one of the most prominently used by state-of-the-art protection mechanisms, both in malware threats and in commercial software protectors.

A VM consists of some bytecode in a custom Instruction Set Architecture (ISA) and an interpreter for such architecture. At compile-time, the parts of the code that are to be protected (obfuscated) are compiled into the VM-ISA and are inserted into the protected program alongside its respective interpreter, whilst removing original (native) code. Each invocation of the protected call is replaced by a call to the VM-ISA interpreter, passing the bytecode as parameter. The interpreter will generally follow a Fetch-Decode-Execute loop over the bytecode calling each VM instruction handler and finally update the VM context and returning control back to native code. More advanced VM implementations can be derived, with added features that harden the implementation and make them more resilient to analysis. A description of different advanced VM implementations can be found in Chapter 5 of [Dan+14].

The design of the VM-ISA is entirely at the discretion of the protection designer, and can be generated uniquely upon protection time. Thus, it is necessary for the analyst to understand the interpreter in order to analyze the bytecode and, eventually translate it back into native instructions.

Note that we can think of VM-based obfuscation as a generalization of Control flow flattening, where both control-flow and data-flow are virtualized, while in control flow flattening only the control-flow was virtualized.

The interested reader can refer to Rolf Rolles' work on unpacking virtualization obfuscators [Rol09]. For a guided example deobfuscating the VM-based protection included in the FinSpy Malware please refer to [Rol18a; Rol18b; Rol18c].

## 1.3 Assessing the quality of code obfuscation

We are interested in evaluating the quality of an obfuscation technique. This turns out to be a complicated issue, mainly due to the fact that attacks on obfuscation are composed

of both human and automatic analysis, meaning that we have to take into account the following items, which are hard to quantify:

- The contribution of reverse engineers in terms of intuition and experience.
- The analysis tools, very often proprietary and custom-made, at least to some extent.

Thus, we find several approaches to characterize the quality of obfuscation. On the one hand, we could use classical software complexity metrics (presented in Section 1.3.1) and derive obfuscation metrics from them (presented in Section 1.3.2). On the other hand, we could take an approach using abstract interpretation[11] to give an attack model to obfuscation (presented in 1.3.3).

### 1.3.1 Complexity metrics of a program

Common software complexity metrics [CTL97] can be classified in three types:

- Number of instructions.
- Control-flow: cyclomatic complexity[12] (number of linearly independent paths), nesting level, knots...
- Data-flow: fan-in/fan-out of instructions or basic blocks, data-flow complexity (number of inter-basic block variable references)...

### 1.3.2 Metrics for obfuscation

We could use the software complexity metrics described above and state that a *good* obfuscation should increase some of them (chosen depending on the particular obfuscation technique). Indeed, this will be the case for the *potency* metric described below.

Collberg et al. define in [CTL97] three metrics often used as a basis to characterize the quality of an obfuscation technique:

- *potency*: Using one of the software complexity metrics described in Section 1.3.1, a *potent* obfuscation transformation is a transformation that increases the chosen complexity.
- *resilience*: determines the resistance of the transformation to human analysis (local, global, inter-procedural o inter-process effort) and to automatic analysis (polynomial or exponential time).
- *cost*: determines the extra resources (execution time and space) of the obfuscated program compared to the original. The cost can be free ($\mathcal{O}(1)$), cheap ($\mathcal{O}(n)$), costly ($\mathcal{O}(n^p)$) or dear (exponential).

[11] `https://www.di.ens.fr/~cousot/AI/IntroAbsInt.html`
[12] `https://en.wikipedia.org/wiki/Cyclomatic_complexity`

Then the authors define the quality of an obfuscation transformation as the combination of these three metrics.

We could also consider *stealth* as a metric: that is, the difficulty to detect the obfuscation. Collberg defines in [CN09] two types of stealth:

- Steganographic[13] stealth: the analyst is unable to determine whether the obfuscation transformation has been applied or not.
- Local stealth: the analyst is unable to determine where the obfuscation transformation has been applied.

### 1.3.3 Attack model from Abstract Interpretation

The metrics described before are clearly limited by the fact that do not take into account the process of reverse engineering. For that reason, Dalla Preda et al. propose in [DG05] a model for attacks and a definition of potency relying on the properties that a reverser might want to infer from the analysis of the program.

This attack model is based on abstract interpretation, which is a way of approximating the concrete semantics of a program. The basic idea is that properties that an analyst might be interested in are encoded as elements $\varphi$ of an abstract domain that models the static and dynamic analyzers. Then, a transformation is said to be potent to a property $\varphi$ if a reverser cannot deduce this property from the obfuscated program, effectively identifying the class of attacks against which the obfuscation is potent.

This does not guarantee the resilience of the obfuscation (i.e. that it cannot be undone). Moreover, although the authors assess the resilience of opaque predicate insertions as a case study, they do not provide a general model for resilience estimation.

This abstract model and its definition of potency are closer to real settings, as they take the reverser interest and methods into account. However, the abstraction of the static and dynamic analyzers raises the question of how are we supposed to know the methods and tools used by the analyst, as reversers often possess their own private and custom toolkit, and it is not really possible to model and assess the resistance against unknown analysis.

## 1.4 Fundamentals of code deobfuscation

Code deobfuscation is the process of transforming an obfuscated (piece of) program $P'$ into a (piece of) program $P''$ that is easier to analyze than $P'$.

Ideally, we would like to have $P'' \approx P$, where $P$ represents the original non-obfuscated program code. This is rarely possible to guarantee, due to the following reasons:

[13] `https://en.wikipedia.org/wiki/Steganography`

- The analyst performing the deobfuscation process almost never has access to original code to check against.
- The analyst is usually not interested in the whole program, but rather some specific parts of it.
- The analyst might be interested in understanding the program operation, or some particular part of its code, rather than reconstructing a functional non-obfuscated binary.

The analyst has several tools available in order to deal with obfuscated code. Besides general reverse engineering frameworks with interactive disassemblers like radare2[14] (or its GUI Cutter[15]), Ghidra[16] or IDA Pro[17], more specific tools that provide dynamic binary instrumentation capabilities (Frida[18]), advanced binary analysis and (dynamic) symbolic execution (Triton[19], Miasm[20], Metasm[21]) are usually very helpful and even necessary for some kinds of analysis.

We quote below brief literal excerpts from [Dan+14] that accurately describe some crucial aspects to reflect on when we discuss about code deobfuscation tools and techniques.

**Static analysis vs Dynamic analysis**

"*Static analysis* is the discipline of automatically inferring information about computer programs without running them. Static analysis tries to derive properties (invariants) that hold for all executions of the program, through a conservative over-approximation of its concrete semantics.

...*Dynamic analysis* is the discipline of automatically inferring information about a running computer program. Dynamic analysis derives properties that hold for one or more executions of a program, through a precise under-approximation."

**Symbolic and concolic execution**

"A common method of dynamic analysis is *dynamic testing*, which executes a program with several inputs and checks the program's response. Generally, test cases explore only a subset of the possible executions of the program.

In order to enlarge the coverage of dynamic testing, the principle of *symbolic execution* uses symbolic values rather than concrete inputs. At any point during symbolic execution, a *symbolic state* of the program is updated. This symbolic state consists of a *symbolic*

[14] `https://rada.re`
[15] `https://cutter.re`
[16] `https://ghidra-sre.org/`
[17] `https://www.hex-rays.com/products/ida/`
[18] `https:/frida.re`
[19] `https://triton.quarkslab.com/`
[20] `https://miasm.re/`
[21] `https://github.com/jjyg/metasm`

*store* and a *path constraint*. The symbolic store contains the symbolic values, and the path constraint is a formula that records the history of all conditional branches taken until the current instruction.

At a given instruction of the program, you can use a *constraint solver* (SMT or SAT solver) to determine the corresponding path constraint. A satisfying assignment provides concrete inputs with which the program reaches the program instruction. By generating new tests and exploring new paths, you can increase the coverage of dynamic testing.

Unfortunately, constraints generated during symbolic execution may be too complex for the constraint solver. If the constraint solver is unable to compute a satisfying assignment, you cannot determine whether a path is feasible or not.

*Concolic execution* provides a solution to this problem in many situations. The idea is to perform both symbolic execution and concrete execution of a program. When the path constraint is too complex for the constraint solver, you can use the concrete information to simplify the constraint (typically by replacing some of the symbolic values with concrete values). You can then expect to find a satisfying assignment of this simplified constraint."

**Soundness and Completeness**

"You can formulate any program analysis problem as a verification that the program satisfies a property. Two fundamental concepts can be used to characterize an analysis algorithm: its *soundness* and its *completeness*.

...Given a property, a *sound* program analysis identifies all violations of the property. However, because it over-approximates the behaviors of the program, it may also report violations of the property that cannot occur.

...A sound symbolic execution guarantees that because a symbolic constraint path is satisfiable, there must be a concrete execution path that reaches the corresponding concrete state (even if some reachable concrete state does not have a corresponding symbolic state).

...The soundness of an abstract interpreter is relative to which questions it can answer correctly, despite the loss of information.

...Given a property, a *complete* analysis algorithm reports a violation of the property only if there is a concrete violation of the property. However, because it under-approximates the behaviors of the program, some concrete violations of the property may not be reported.

...A complete symbolic execution covers all concrete transitions. It guarantees that if a concrete execution state is reachable, then there must be a corresponding symbolic state.

... Both soundness and completeness can be defined for static and dynamic analyses, which are good candidates to represent the actions conducted by reversers when they try to simplify the representation of an obfuscated program."

**Remark 1.1.** We will not go deep into any particular deobfuscation procedure, as different techniques are usually very specific to the type of obfuscation we deal with. The interested reader can refer to Chapter 5 of [Dan+14] for more information, a detailed treatment of the aspects addressed above and a discussion of different tools with guided examples. Part III (chapters 9 to 13) of [And18] also provides a good treatment of some advanced binary analysis techniques that can be used for deobfuscation, with guided examples as well. Another good resource with hands-on exercises can be found in [Lab20].

## 1.5 Discussion: academic vs practical code deobfuscation

The nature of code deobfuscation is different from formal verification[22] or program analysis[23], even if we prefer to use techniques developed in those contexts.

As stated before, whereas an obfuscator transforms a program $P$ into an obfuscated $P'$, in code deobfuscation we do not necessarily need to transform back the obfuscated $P'$ into a deobfuscated (or at least simpler) form, but instead we usually only seek enough information of $P$ to answer certain questions related to our current reverse engineering effort. Thus, even if we might "dislike" using unsound methods, we still prefer to obtain tangible results by the end of the day, so we may employ such methods.

For example, consider a pattern-based obfuscator that contains erroneous pattern substitutions, i.e. not preserving semantic equivalence (e.g. think of a substitution where the side effects on CPU flags are not preserved). Suppose that an analyst dealing with such an obfuscation is aware of the errors and is able to correct them at deobfuscation time. This means that the deobfuscator would be "incorrect", as it will not preserve the semantic equivalence of the transformation. However, it will actually produce "correct" results with respect to the pre-obfuscated code. Then, should the deobfuscating substitution be applied?

From a strictly academic/formal standpoint, the answer would probably be no. However, from a reverse engineering/practical standpoint we would say yes.

[22] `https://en.wikipedia.org/wiki/Formal_verification`
[23] `https://en.wikipedia.org/wiki/Program_analysis`

# Chapter 2

# Mixed Boolean-Arithmetic expressions

## 2.1 Fundamentals

The concept of Mixed Boolean-Arithmetic (MBA) expressions initially appeared in the context of code obfuscation, in the work by Zhou et al. [Zho+07]. Expressions mixing arithmetic and bitwise operators are already in use in different areas, such as in cryptography, even if they are not given a name (see Section 2.2.4).

Research literature in the specific field of MBA expressions is very scarce. Since the work of [Zho+07], the few publications explicitly related to MBA expressions have been mainly focused on assessing the *strength* or *resilience* of the obfuscation techniques introduced by Zhou et al. To our knowledge, there is no other published work addressing the formalization of MBA expressions than Eyrolles' PhD thesis [Eyr17]. Hence, it will be extensively cited in this chapter.

### 2.1.1 Polynomial MBA expressions

In order to capture the different computations that occur in any modern microprocessor, Zhou et al. propose the following algebraic system.

**Definition 2.1** ([Zho+07])**.** *With $n$ a positive integer and $B = \{0, 1\}$, the algebraic system $(B^n, \wedge, \vee, \oplus, \neg, \leq, \geq, >, <, \leq^s, \geq^s, >^s, <^s, \neq, =, \gg, \gg^s, \ll, +, -, \cdot)$, where $\ll, \gg$ denote left and right shifts, $\cdot$ denotes multiply, and signed compares and arithmetic right shift are indicated by $^s$, is a* Boolean-arithmetic algebra (BA-algebra)*, $BA[n]$. $n$ is the dimension of the algebra.*

Nevertheless, we chose to follow the same approach as in [Eyr17], by deliberately excluding from our study the subject of MBA inequalities present in the former definition. This is because we are interested in deobfuscating (simplifying) MBA expressions, while an inequality is an assertion that returns a true or false value, which changes the issue to handle. In terms of notation, this essentially means that we will be somewhat more flexible

in our definition of MBA expressions. As in [Eyr17], we chose to define MBA expressions by explicitly describing the different building blocks that compose them and how they are bundled together, but we will not strictly define them as a function $f : (B^n)^t \to B^n$ linked to the definition of a *BA-algebra*, as is the case of [Zho+07].

In spite of that, it is important to remark that we will work on $n$-bit words considered at the same time as elements of different mathematical structures. For example, standard arithmetic operations are considered in $(\mathbb{Z}/2^n\mathbb{Z}, +, \times)$ while bitwise operations belong to $(\{0,1\}^n, \wedge, \vee, \neg)$ or $(\{0,1\}^n, \wedge, \oplus)$.

**Remark 2.2.** We will use interchangeably the terms of *boolean* and *bitwise* operators.

**Definition 2.3** (Polynomial MBA expression)**.** *An expression $E$ of the form*

$$E = \sum_{i \in I} a_i \left( \prod_{j \in J_i} e_{i,j}(x_1, \ldots, x_t) \right) \tag{2.1}$$

*where the arithmetic sum and product are modulo $2^n$, $a_i$ are constants in $\mathbb{Z}/2^n\mathbb{Z}$, $e_{i,j}$ are bitwise expressions of variables $x_1, \ldots, x_t$ in $\{0,1\}^n$, $I \subset \mathbb{Z}$ and for all $i \in I$, $J_i \subset \mathbb{Z}$ are finite index sets, is a polynomial Mixed Boolean-Arithmetic (MBA) expression.*

**Definition 2.4** (Linear MBA expression)**.** *A polynomial MBA expression of the form*

$$E = \sum_{i \in I} a_i e_i(x_1, \ldots, x_t)$$

*is called a linear MBA expression.*

**Example 2.5.** [Zho+07] The expression $E$ written as

$$E = 8458(x \vee y \wedge z)^3((xy) \wedge x \vee t) + x + 9(x \vee y)yz^3$$

is a non-linear polynomial MBA expression.

**Example 2.6.** [Eyr17] The expression $E$ written as

$$E = (x \oplus y) + 2 \times (x \wedge y) \tag{2.2}$$

is a linear MBA expression, which *simplifies* to $E = x + y$.

The MBA-obfuscated expressions that are of our interest rely on composing layers of MBA rewritings. Therefore, the following statement from [Zho+07] is essential to our study.

**Proposition 2.7.** *Consider the composition of polynomial MBA expressions by treating each of $x_1, \ldots, x_t$ as a polynomial MBA expression of other variables itself. Then, this composition is still a polynomial MBA expression.*

This fact guarantees that we will always be working with polynomial MBA expressions. Unless stated otherwise, when we refer to an MBA expression $E$ we are denoting a polynomial MBA expression.

## 2.2 Obfuscation with MBA expressions

### 2.2.1 Obfuscation of expressions

This technique was presented in [Zho+07] and in several patents with intersecting authors, like [JXY08].

Given an MBA expression $E_1$, we are interested in generating an equivalent expression $E_2$ which is more *complex* (see Section 2.3) than the initial expression $E_1$. The main idea is to obfuscate one or several operators from $E_1$. The process relies on two differentiated components:

#### MBA rewriting

A chosen operator is rewritten with an equivalent MBA expression. A list of rewriting rules is given in Appendix A of [Eyr17]. Other MBA equalities can be found in [War12] referred as *bit hacks*.

**Example 2.8.**

$$x + y \to (x \oplus y) + 2 \times (x \wedge y) \tag{2.3}$$

#### Insertion of identities

Let $e$ be any subexpression of the target expression being obfuscated. Then, we can write $e$ as $f^{-1}(f(e))$ with $f$ being any invertible function on $\mathbb{Z}/2^n\mathbb{Z}$. The function $f$ is often an affine function, as presented in [Zho+07].

**Example 2.9.** Let $E_1 = x + y$ on $\mathbb{Z}/2^8\mathbb{Z}$. Consider the following functions $f$ and $f^{-1}$ on $\mathbb{Z}/2^8\mathbb{Z}$:

$$f : x \mapsto 39x + 23$$
$$f^{-1} : x \mapsto 151x + 111$$

Consider now the expression $e_1$ obtained by applying the rewriting rule (2.3) to $E_1$:

$$e_1 = (x \oplus y) + 2 \times (x \wedge y)$$

Then apply the insertion of identities produced by $f$ and $f^{-1}$:

$$e_2 = f(e_1) = 39 \times e_1 + 23$$
$$E_2 = f^{-1}(e_2) = 151 \times e_2 + 111$$

Finally, expand $E_2$ to observe the final obfuscated expression derived from $E_1 = x + y$:

$$E_2 = 151 \times (39 \times ((x \oplus y) + 2 \times (x \wedge y)) + 23) + 111$$

### 2.2.2 Opaque constant

This technique allows to hide a target constant $K$ (e.g. a secret key used in a decryption routine). It uses *permutation polynomials*[1], which were characterized by Rivest in [Riv01]. However, no inversion algorithm was provided then. Later, Zhou et al. provide a subset of such polynomials of degree $m$ noted $P_m(\mathbb{Z}/2^n\mathbb{Z})$, as well as a formula to find the inverse of such polynomials. A study of the formal characterization of such polynomials is out of scope of this work (see [Zho+07]).

The method to construct the opaque constant (i.e. the constant obfuscation) is given by the following proposition.

**Proposition 2.10.** *Let*

- $K \in \mathbb{Z}/2^n\mathbb{Z}$ *be the target constant to hide,*
- $P \in P_m(\mathbb{Z}/2^n\mathbb{Z})$ *and* $Q$ *its inverse:* $P(Q(X)) = X, \forall X \in \mathbb{Z}/2^n\mathbb{Z}$*,*
- $E$ *be an MBA expression of variables* $(x_1, \ldots, x_t) \in (\mathbb{Z}/2^n\mathbb{Z})^t$ *non-trivially equal to zero.*

*Then, the constant* $K$ *can be replaced by* $P(E + Q(K))$ *for any values taken by* $(x_1, \ldots, x_t)$*.*

*Proof.* By construction, we have:

$$P(E + Q(K)) = P(Q(K)) = K$$

regardless of the input variables $(x_1, \ldots, x_t)$, as the expression $E$ vanishes. □

**Remark 2.11.** Such an expression $E$ can be easily obtained from a given MBA rewriting rule by subtracting the one side from the other of the equivalence rule. For example, from rule (2.3) we can generate the non-trivially zero expression:

$$E = x + y - (x \oplus y) - 2 \times (x \wedge y)$$

With the opaque constant technique we obtain a function that computes $K$ for all its input variables. Those variables can be chosen randomly every time the program requires the key $K$ to perform any computation.

#### Algebraic weakness of the opaque constant technique

**Lemma 2.12** ([Eyr17])**.** *Let*

- $P(X) = a_0 + a_1X + a_2X^2 + \cdots + a_dX^d$ *be a polynomial of degree* $d$*,*
- $Q$ *be a polynomial of degree* $d$*, such that* $(P(Q(X)) = X, \forall X \in \mathbb{Z}/2^n\mathbb{Z}$*,*

[1] `https://en.wikipedia.org/wiki/Permutation_polynomial`

- $E = \sum_{i \in I} a_i \big( \prod_{j \in J_i} e_{i,j}(x_1, \ldots, x_t) \big)$ *be an MBA expression of variables* $(x_1, \ldots, x_t) \in (\mathbb{Z}/2^n\mathbb{Z})^t$ *non-trivially equal to zero, with no* $e_{i,j}$ *such that* $e_{i,j}(x_1, \ldots, x_t) = 1$ *whatever the values of* $(x_1, \ldots, x_t)$*. This condition enforces that the sum composing the MBA expression does not contain any constant.*

*Then, the constant monomial of* $P(E + Q(K))$ *is equal to* $K$*.*

*Proof.*

$$\begin{aligned} P(E + Q(K)) &= a_0 + a_1(E + Q(K)) + \cdots + a_d(E + Q(K))^d \\ &= a_0 + a_1 Q(K) + a_2 Q(K)^2 + \cdots + a_d Q(K)^d + \varphi(E) \end{aligned}$$

with $\varphi(E) = \sum_{k=1}^{d} a_k \big( \sum_{i=0}^{k-1} E^{k-i} Q(K)^i \big)$, a polynomial in variables $x_1, \ldots, x_t$ with no constant monomial, as every monomial of $\varphi(E)$ is multiplied by a positive power of $E$. This means that the constant part of $P(E + Q(K))$ is:

$$a_0 + a_1 Q(K) + a_2 Q(K)^2 + \cdots + a_d Q(K)^d = P(Q(K)) = K$$

□

This lemma reveals that depending on the form of the null MBA expression used for obfuscation, the expanded form of the opaque constant might display clearly the constant intended to be concealed. In particular, this will happen when the chosen null MBA expression does not contain a constant part.

A detailed assessment of the resilience of the opaque constant technique was presented by Biondi et al. in [Bio+17].

### 2.2.3 Generating new linear MBA equalities

**Theorem 2.13** ([Zho+07; Eyr17])**.** *With* $n$ *the number of bits,* $s$ *the number of bitwise expressions and* $t$ *the number of variables, all positive integers, let:*

- $(X_1, \ldots, X_k, \ldots, X_t) \in \{\{0,1\}^n\}^t$ *be vectors of variables on* $n$ *bits.,*
- $e_0, \ldots, e_j, \ldots, e_{s-1}$ *be bitwise expressions,*
- $e = \sum_{j=0}^{s-1} a_j e_j$ *be a linear MBA expression, with* $a_j$ *integers,*
- $e_j(X_1, \ldots, X_t) = \begin{pmatrix} f_j(X_{1,0}, \ldots, X_{t,0}) \\ \vdots \\ f_j(X_{1,n-1}, \ldots, X_{t,n-1}) \end{pmatrix}$ *with* $X_{k,i}$ *the* $i$*-th bit of* $X_k$ *and*

$$\begin{aligned} f_j : \{0,1\}^t &\to \{0,1\} \qquad 0 \leq j \leq s-1 \\ u &\mapsto f_j(u) \end{aligned}$$

- $F = \begin{pmatrix} f_0(0) & \dots & f_{s-1}(0) \\ \vdots & & \vdots \\ f_0(2^t-1) & \dots & f_{s-1}(2^t-1) \end{pmatrix}$ *the* $2^t \times s$ *matrix of all possible values of* $f_j$ *for any i-th bit.*

*If* $F \cdot V = 0$ *has a non-trivial solution, with* $V = (a_0, \dots, a_{s-1})^T$*, then* $e = 0$*.*

*Proof.* Let $F \cdot V = 0$, with $V = (a_0, \dots, a_{s-1})^T$. If we explicit $F \cdot V$, we get:

$$F \cdot V = \begin{pmatrix} f_0(0) & \dots & f_{s-1}(0) \\ \vdots & & \vdots \\ f_0(2^t-1) & \dots & f_{s-1}(2^t-1) \end{pmatrix} \cdot \begin{pmatrix} a_0 \\ \vdots \\ a_{s-1} \end{pmatrix} = \begin{pmatrix} \sum_{j=0}^{s-1} a_j \cdot f_j(0) \\ \vdots \\ \sum_{j=0}^{s-1} a_j \cdot f_j(2^t-1) \end{pmatrix},$$

meaning that $F \cdot V = 0 \Leftrightarrow \sum_{j=0}^{s-1} a_j \cdot f_j(l) = 0$ for every $l \in \{0, \dots, 2^t - 1\}$.
This is equivalent to having $\sum_{j=0}^{s-1} a_j \cdot f_j(X_{1,i}, \dots, X_{t,i}) = 0$ for every i, whatever the values of the $X_{k,i}$.
On the other hand, we can write $e$ as:

$$\begin{aligned} \sum_{j=0}^{s-1} a_j \cdot e_j(X_1, \dots, X_t) &= \sum_{j=0}^{s-1} a_j \cdot \begin{pmatrix} f_j(X_{1,0}, \dots, X_{t,0}) \\ \vdots \\ f_j(X_{1,n-1}, \dots, X_{t,n-1}) \end{pmatrix} \\ &= \sum_{j=0}^{s-1} a_j \cdot \sum_{i=0}^{n-1} f_j(X_{1,i}, \dots, X_{t,i}) \cdot 2^i \\ &= \sum_{j=0}^{s-1} \Big( \sum_{i=0}^{n-1} a_j \cdot f_j(X_{1,i}, \dots, X_{t,i}) \cdot 2^i \Big) \\ &= \sum_{i=0}^{n-1} 2^i \Big( \sum_{j=0}^{s-1} a_j \cdot f_j(X_{1,i}, \dots, X_{t,i}) \Big). \end{aligned}$$

If $F \cdot V = 0$, then $\sum_{j=0}^{s-1} a_j \cdot f_j(X_{1,i}, \dots, X_{t,i}) = 0$ for every i, thus

$$\sum_{i=0}^{n-1} 2^i \Big( \sum_{j=0}^{s-1} a_j \cdot f_j(X_{1,i}, \dots, X_{t,i}) \Big) = 0 \quad \forall i,\ 0 \le i \le n-1,$$

meaning that $e = 0$. □

This theorem provides a method to create new linear MBA equalities. The method is based on the following corollary.

**Corollary 2.14.** *Given a $\{0,1\}$-matrix of size $2^t \times s$ with linearly dependent column vectors, one can generate a non-trivially equal to zero linear MBA expression of $t$ variables as a linear combination of $s$ bitwise expressions.*

**Example 2.15.** Let

$$F = \begin{pmatrix} 0 & 0 & 0 & 1 & 1 \\ 0 & 1 & 1 & 0 & 1 \\ 1 & 0 & 1 & 1 & 1 \\ 1 & 1 & 0 & 1 & 1 \end{pmatrix}$$

with column-vectors truth-tables for:

$$\begin{aligned} f_0(x,y) &= x \\ f_1(x,y) &= y \\ f_2(x,y) &= (x \oplus y) \\ f_3(x,y) &= (x \vee (\neg y)) \\ f_4(x,y) &= -1 \end{aligned}$$

Now, the vector $V$, solution to $F \cdot V = 0$ is:

$$V = (1, -1, -1, -2, 2)^T$$

This yields the following linear MBA equation:

$$x - y - (x \oplus y) - 2(x \vee (\neg y)) - 2 = 0$$

Finally, we can derive many equalities from this equation and use them to form MBA rewriting rules:

$$\begin{aligned} x - y &\rightarrow (x \oplus y) + 2(x \vee (\neg y)) + 2 \\ (x \oplus y) &\rightarrow x - y - 2(x \vee (\neg y)) - 2 \end{aligned}$$

### 2.2.4 Obfuscation vs Cryptography

A mixing of bitwise and arithmetic operators was already used in the context of cryptography (before being given a name) to design symmetric primitives, with the stated goal of getting efficient, non-linear and complex interactions between operations. However, there is a key difference between what is looked for in cryptography and in obfuscation, as stated in [Eyr17]:

- In cryptography, the MBA expression is the direct result of the algorithm description, and the resulting cryptosystem has to verify a set of properties (e.g. non-linearity, high algebraic degree) from a *black-box* point of view. The complex form of writing is directly related to some kind of hopefully intrinsic computational complexity for the resulting function: one wants the inverse computation to be difficult to deduce without knowing the key.

- In obfuscation, an MBA is the result of rewriting iterations from a simpler expression which can have very simple *black-box* characteristics. There is no direct relation between the complex form of the expression and any intrinsic computational complexity of the resulting function: on the contrary, when obfuscating simple functions, one knows that the complex form of writing is related to a simpler computational function. Nevertheless, getting the result of the computation for the obfuscated expression requires indeed to get through all the operators in the considered expression which implies somehow a computational complexity.

To sum up, we could say that in cryptography, the intention is to gain intrinsic computational complexity, while in obfuscation, one seeks a complexity linked to the form of writing that would prevent simplification.

## 2.3 Complexity

### 2.3.1 Incompatibility of operators

The mixing of arithmetic and bitwise operators do not interact very well, as there are no general rules (e.g. distributivity, associativity...) to cope with them. Although we find some particular cases where rules analogous to distribution apply, they cannot be generalized.

**Example 2.16.** We have that the following equality holds for all input values:

$$\forall x, y \in \mathbb{Z}/2^n\mathbb{Z} : 2 \times (x \wedge y) = (2x \wedge 2y) \tag{2.4}$$

but this distribution rule does not generalize:

$$\exists x, y \in \mathbb{Z}/2^n\mathbb{Z} : 3 \times (x \wedge y) \neq (3x \wedge 3y) \tag{2.5}$$

The equality from expression (2.4) is due to the fact that multiplication by $2^n$ can be seen as a left shift on $n$ bits, which is a bitwise operator.

As motivated by 2.2.4, cryptography can provide us with an example study of what means incompatibility between operators and how it can prevent an easy study of MBA expressions in a unified domain.

A classic example is the IDEA block cipher [LM91]. Its construction relied in three key components, carefully interleaved to prevent easy manipulation of the resulting expressions:

- multiplication $\odot$ in $(\mathbb{Z}/(2^{16}+1)/\mathbb{Z})^*$
- addition $\boxplus$ in $\mathbb{Z}/2^{16}\mathbb{Z}$
- bitwise XOR $\oplus$ in $\mathbb{F}_2^{16}$ (where $\mathbb{F}_2$ is the finite field of two elements as described in Section 2.4.2).

Among the reasons why those three operations are incompatible, we find that:

- No pair of three operations satisfies a distributive law.
- No pair of three operations satisfies a generalized associative law.

The incompatibility study of former operators was at the basis of the argument on the *confusion*[2] property of the block cipher.

Further details and more reasons for this incompatibility of operators in IDEA block cipher can be found in [LM91].

### 2.3.2 DAG representation

We want to obtain a tree-like representation for MBA expressions. A common representation would be in the form of an Abstract Syntax Tree (AST)[3]. However, we are interested in avoiding multiple occurrences of the same subexpression. Because of that, a Directed Acyclic Graph (DAG)[4] representation is chosen, as it identifies common subexpressions of a given expression.

**Definition 2.17** (DAG representation [Eyr17])**.** *The DAG representation of an MBA expression is an acyclic graph G where:*

- *there is only one root node,*
- *leaves represent constant numbers or variables,*
- *intermediate nodes represent arithmetic or bitwise operators,*
- *an edge from a node v to a node v' means that v' is an operand of v,*
- *common expressions are shared, which means that they only appear once in the graph.*

An example of a DAG representation of an MBA expression can be seen in Figure 2.1.

### 2.3.3 Metrics

Based on the DAG representation, Eyrolles proposes three different metrics in [Eyr17] intended to characterize the complexity of MBA expressions.

**Remark 2.18.** The metrics presented are not particularly meant to characterize the resilience of the MBA obfuscation techniques described before (MBA rewriting and opaque constant), but the complexity of an MBA expression in general. Nevertheless, the complexity of the MBA expressions generated by an obfuscation technique can indeed be a factor used to evaluate the resilience of said technique. Details in this topic can be found in Chapter 5 of [Eyr17].

[2] `https://en.wikipedia.org/wiki/Confusion_and_diffusion`
[3] `https://en.wikipedia.org/wiki/Abstract_syntax_tree`
[4] `https://en.wikipedia.org/wiki/Directed_acyclic_graph`

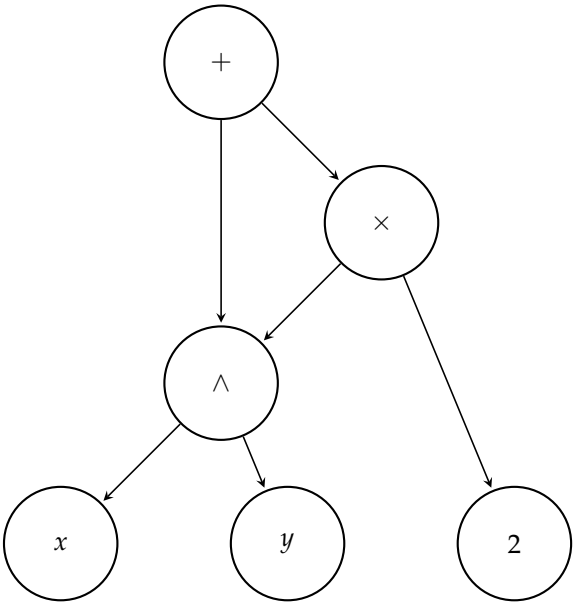



Figure 2.1: DAG representation of MBA expression: $2 \times (x \wedge y) + (x \wedge y)$

### Number of Nodes

**Definition 2.19.** *Let $E$ be an MBA expression. We define the* size *of $E$ as its number of nodes in DAG representation, i.e. operators, variables and constants.*

As we consider that every occurrence of any subexpression can be simplified in the same way, the sharing property of DAG representation presents a great advantage, as each occurrence of such a subexpression is only taken into account once. This means that simplifying an occurrence of a subexpression is equivalent to simplifying all its occurrences.

Decreasing the number of nodes makes the expression easier to apprehend and manipulate. It might be useful for a potential brute-force simplification approach, as it decreases the size of the input set.

### MBA Alternation

We are interested in quantifying the "mixed aspect" of an MBA expression with. Thus, a purely arithmetic or purely boolean expression should have a null *MBA alternation*. To define the MBA alternation of an expression, we first need to define the *type* of an operator *op*. Eyrolles proposes that the type is arithmetic if $op \in \{+, -, \times\}$ and boolean (or bitwise) if $op \in \{\wedge, \vee, \oplus, \neg\}$.

**Definition 2.20** (MBA alternation)**.** *For a graph $G = (V, E)$ (being the DAG representation of an MBA expression), with $V$ the set of vertices and $E$ the set of edges, the* MBA alternation *$alt_{MBA}(G)$ is:*

$$alt_{MBA}(G) = |\{(v_1, v_2) : type(v_1) \neq type(v_2)\}|$$

*where* $(v_1, v_2) \in E$ *represents the edge linking the two vertices* $v_1, v_2 \in V$

These edges represent potentially *difficult* points in the MBA expression to be simplified.

#### Average Bit-vector size

We add to the DAG representation a property that we call the *bit-vector size*.

**Definition 2.21** (Bit-vector size)**.** *For a node* $v$*, the bit-vector size* $\text{bvsize}(v)$ *is:*

- *If* $v$ *is a leaf node, the bit size of the variable or constant it represents. This size can be deduced from the context (e.g. size of the input of a function), or by additional indications (e.g. binary masks). The size of a constant may also be inferred from the actual number of bits of that constant (possibly rounded to the next power of two).*
- *If* $v$ *represents an operator, the bit size of the output of the operation. This depends on the nature of the operator:*
  - *if* $v$ *represents a binary operator in* $\{+, -, \times, \oplus, \vee\}$ *with* $v_1, v_2$ *as operands, then:* $\text{bvsize}(v) = \max(\text{bvsize}(v_1), \text{bvsize}(v_2))$
  - *if* $v$ *represents a boolean AND* $(\wedge)$ *with* $v_1, v_2$ *as operands, then:* $\text{bvsize}(v) = \min(\text{bvsize}(v_1), \text{bvsize}(v_2))$
  - *if* $v$ *represents a unary operator in* $\{\neg, -\}$ *with* $v_1$ *as operand, then:* $\text{bvsize}(v) = \text{bvsize}(v_1)$

To illustrate the definition, observe the DAG example of the expression $(x + 25863) \wedge y$ in Figure 2.2, assuming the variable $x$ is on 32 bits, the variable $y$ is on 8 bits and the constant 25863 is on 16 bits.

This definition of the bit-vector size just accounts for the "default" size inferred from initial analysis of the program. However, it is certainly possible to apply some transformations that reduce the bit-vector size of specific nodes in some cases. The most common reduction occurs when an operator *AND* $(\wedge)$ has operands $op_1, op_2$ of different bit-vector size, say $\text{bvsize}(op_1) < \text{bvsize}(op_2)$. In that case, the bit-vector size of the *AND* is $\text{bvsize}(op_1)$, as the definition claims, but if $op_2 \in \{+, -, \times, \wedge, \vee, \oplus, \neg\}$, then we can reduce the bit-vector size of all terms in $op_2$ to $\text{bvsize}(op_1)$.

The restriction of operators in $op_2$ guarantees that there will be no dependencies on the most significant bits in the computation of the least significant bits. This reduction process is illustrated in Figure 2.3, where the expression $(x + 25863) \wedge y$ as represented in Figure 2.2 has been reduced to $(x + 7) \wedge y$. Observe that the constant 25863 has been reduced to 7, as it satisfies $25863 \equiv 7 \mod 8$.

We can use *average bit-vector size* as a metric accounting for both local and global reductions of the bit-vector size. Note that if we only consider bit-vector size of the root

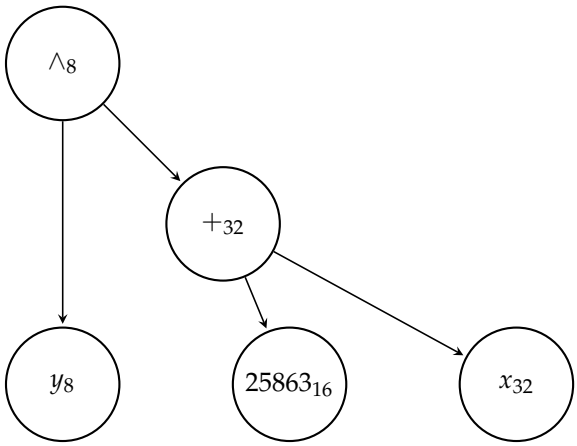



Figure 2.2: DAG with bit-vector size: $(x + 25863) \wedge y$

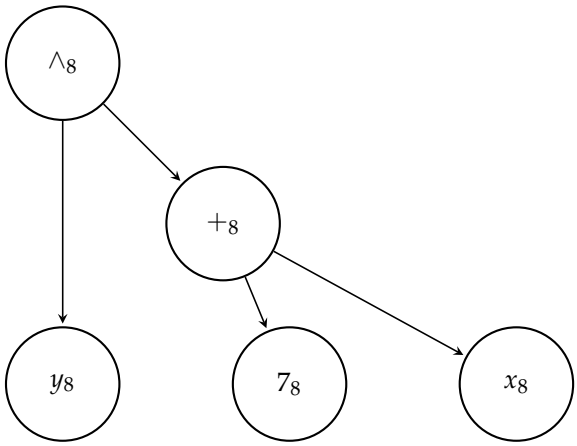



Figure 2.3: DAG with reduced bit-vector size: $(x + 25863) \wedge y \rightarrow (x + 7) \wedge y$

node, we would miss any simplification of a subgraph (subexpression).

Decreasing the bit-vector size of certain nodes might be beneficial in different ways:

- Allow easier recognition of rewrite rules used for obfuscation.
- Reduce complexity of simplification approaches that use bit-blasting (Section 2.4.2).

## 2.4 Simplification

### 2.4.1 Context

The main reason why MBA expressions are used in the field of code obfuscation is because there is a lack of not only theoretical background, as we have seen, but also practical tools to deal with them. As stated in [Eyr15], on the side of computer algebra software (Maple[5], Sage[6]...), if arithmetic expressions (namely, polynomials) are featured, bitwise operators are often not supported with symbolic variables, meaning the simple manipulation of MBA is not even possible. On the other hand, SMT solvers such as Z3[7] or Boolector[8] offer an implementation of the bit-vector logic, but it is not their role to focus on simplification: they rather aim at proving SAT.

Before continuing, the first thing to address is the question of what does it even mean to simplify (or deobfuscate) an MBA expression. Considering the general literature on expression simplification, we usually find two different approaches:

- On the one hand: computing a unique representation for equivalent objects. That is, finding a *canonical representation*. This is the most studied simplification technique in literature, both for pure arithmetic expressions and for pure boolean expressions (normal forms). However, depending on the definition of simplicity (whether it is cheaper to store, easier to read, more efficient to compute...) a canonical form may not always be considered the simplest form. Consider for example the pure arithmetic expression $(1 + x)^{100}$. We have that its canonical form as an expanded polynomial $1 + 100x + 4950x^2 + \cdots + 4950x^{98} + 100x^{99} + x^{100}$ is clearly more confusing for a human than the factorized form. Even in terms of internal representation, a machine would probably prefer storing the factorized form.
- On the other hand: finding an equivalent, but simpler form. Here, the meaning of "simpler" depends on the context. As obfuscation is designed to counter both human and automatic analysis, the definition of what is a simple program and/or expression is thus double: for a human, the readability would probably be of concern, while for a machine, the questions of performance (in terms of memory or computing time) would probably be of importance. Because obfuscation aims mainly at

[5] `https://www.maplesoft.com/`
[6] `https://www.sagemath.org/`
[7] `https://github.com/Z3Prover/z3`
[8] `https://boolector.github.io/`

preventing the analyst to get information, we usually focus on the readability (or the understandability) of the program.

Generally speaking, we are interested in decreasing the value of complexity metrics presented in 2.3.3. Nevertheless, it is important to note that the definition of a simple expression could also be conditioned by the simplification approach chosen. This effectively means that different complexity metrics could be relevant and bring difficulty to the simplification process depending on the strategy followed.

Two different approaches will be discussed in following sections 2.4.2 and 2.4.3. The former treats MBA expressions on a bit level, while the latter treats them on a word level.

**Remark 2.22.** Note that we will be discussing the following simplification techniques assuming a *clean* context, where MBA-obfuscated expressions are generated and simplified directly from the source code. In a reverse engineering context, the analyzed MBA would be in their assembly form, and we would need to translate from assembly to a high-level representation of the expression. This is not a trivial task, and usually requires the aid of a symbolic execution engine running on the assembly level.

### 2.4.2 Bit-blasting approach

This approach was presented in [GEV16]. The idea is that MBA expressions can be expressed as boolean expressions by computing the effect of every operation on each bit of the resulting value. Thus, the goal is to get a symbolic and canonical representation of MBA expressions at the bit-level.

The unicity of the representation is guaranteed by using a canonical representation. In particular, the *algebraic normal form* (ANF) is chosen, which means that expressions obtained will only contain $XOR$ $(\oplus)$ and $AND$ $(\wedge)$ operators. The motivation of this choice is due to the fact that we can take advantage of the underlying algebraic structure, which involves the finite field with 2 elements:

$$\mathbb{F}_2 = (\{0,1\}, \oplus, \wedge)$$

Then, any $n$-bit boolean expression can be expressed in the following form:

$$\bigoplus_{u \in \mathbb{F}_2^n} c_u \bigwedge_{i=0}^{n-1} x_i^{u_i}$$

where $c_u \in \mathbb{F}_2$, $x_i^{u_i} = x_i$ if $u_i = 1$ and $x_i^{u_i} = 1$ if $u_i = 0$.

The ANF of a boolean expression can be naturally extended to a vectorial boolean expression. Thus, any vectorial boolean function from $\mathbb{F}_2^n$ into $\mathbb{F}_2^m$ can be expressed canonically. Indeed, each coordinate of the vectorial expression is a boolean expression.

**Remark 2.23.** At the bit level, a variable $x \in \mathbb{Z}/2^n\mathbb{Z}$ is represented with a symbolic vector in $\mathbb{F}_2^n$, where $x_0$ is the Least Significant Bit (LSB) of $x$ and $x_{n-1}$ its Most Significant Bit (MSB),

**Example 2.24.** The application defined by $F(x) = (x \oplus 14) \wedge 7$ from $\mathbb{F}_2^4$ into $\mathbb{F}_2^4$ is represented by:

$$\begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \end{pmatrix} \mapsto \begin{pmatrix} (x_0 \oplus 0) \wedge 1 \\ (x_1 \oplus 1) \wedge 1 \\ (x_2 \oplus 1) \wedge 1 \\ (x_3 \oplus 1) \wedge 0 \end{pmatrix} = \begin{pmatrix} x_0 \\ x_1 \oplus 1 \\ x_2 \oplus 1 \\ 0 \end{pmatrix}$$

**Example 2.25.** The application defined by $F(x, y) = x \vee y$ from $\mathbb{F}_2^4 \times \mathbb{F}_2^4 \cong \mathbb{F}_2^8$ into $\mathbb{F}_2^4$ is represented by:

$$\begin{pmatrix} x_0 \\ x_1 \\ x_2 \\ x_3 \\ y_0 \\ y_1 \\ y_2 \\ y_3 \end{pmatrix} \mapsto \begin{pmatrix} (x_0 \wedge y_0) \oplus x_0 \oplus y_0 \\ (x_1 \wedge y_1) \oplus x_1 \oplus y_1 \\ (x_2 \wedge y_2) \oplus x_2 \oplus y_2 \\ (x_3 \wedge y_3) \oplus x_3 \oplus y_3 \end{pmatrix}$$

Note that the we are expressing the $OR$ $(\vee)$ operator in ANF [9].

As stated before, the main goal is to represent, with a bit-per-bit symbolic and canonical form, a given MBA expression like:

$$F(X, Y, Z) = (((X \oplus (Y - 58)) \times Z) \vee 31)$$

To do so, we need to be able to get the ANF of word-level (vectorial) boolean and arithmetic operations. Boolean operations turn out to be fairly easy due to the ANF representation we chose, while arithmetic ones are somewhat more involved. Besides, we are also interested in the inverse process of canonicalization, i.e. *identifying* different operations from a given ANF representation of an MBA expression.

The big strength of the bit-blasting approach is to transform the problem of MBA simplification into boolean expression simplification, while the main drawback is that the canonicalization can be very expensive (in memory and time) especially when arithmetic operators are involved. Another issue is that identification from boolean expressions to word-level expressions is not trivial. Moreover, simplification using bit-blasting can be quite efficient on expressions with low number of bits (e.g. 8 bits), but struggles to scale for expressions with larger number of bits.

Further information on this technique, including canonicalization and identification details for different boolean and arithmetic operators, as well as a software implementation[10] can be found in [GEV16] and in Section 4.2 of [Eyr17].

[9] `https://www.wolframalpha.com/input/?i=simplify+%28a+and+b%29+xor+a+xor+b`

[10] `https://github.com/quarkslab/arybo`

### 2.4.3 Symbolic approach

This approach was presented in [EGV16] and is based on the following:

- On the one hand, using existing simplification techniques on parts of the MBA expression that may contain only one type of operator.
- On the other hand, a *term rewriting* approach is used to create the missing link between subexpressions alternating different types of operators.

This approach is motivated by the fact that MBA obfuscation is mainly achieved by using a set of rewriting rules, as introduced and described in [Zho+07]. The idea is that rewrite rules for deobfuscation can be obtained by inverting the direction of rewrite rules used for obfuscation. For example, the rewrite rule 2.3 can be inverted into a potential deobfuscation rewriting rule:

$$x + y \rightarrow (x \oplus y) + 2 \times (x \wedge y)$$
$$(x \oplus y) + 2 \times (x \wedge y) \rightarrow x + y$$

As we can see, rewriting rules used for obfuscating purposes increase both the number of nodes and the MBA alternance of the expression, while the ones devoted to deobfuscation will decrease those complexity metrics. A set of rewriting rules can be found in Appendix A of [Eyr17].

Because the algorithm using symbolic simplification works at the word-level, the simplification is not impeded by an increasing number of bits. The representation of the expressions is also far smaller than the representation in the bit-blasting approach. However, this simplification approach is very sensible to the size of the obfuscated expression, in terms of number of nodes. Another drawback of this approach is that it is highly dependent on the chosen set of rewrite rules. Indeed, if only one obfuscation rule is unknown, the simplification algorithm will not be able to reduce the expression as much as it would with knowledge of that rule.

Further details on this technique, including a software implementation[11] can be found in [EGV16] and in Section 4.3 of [Eyr17].

[11] `https://github.com/quarkslab/sspam`

# Chapter 3

# Program synthesis for code deobfuscation

## 3.1 Context

State-of-the-art code deobfuscation techniques rely on symbolic execution, taint analysis and a combination of them. Symbolic execution is a technique that lets the analyst transform the control-flow and data-flow of the program into symbolic expressions. Taint analysis is a technique that lets the analyst know at each program point what part of memory or registers are controllable by the user input. These techniques have been shown to be promising to address control-flow based obfuscation, in which we need to check the satisfiability of the obtained symbolic boolean condition. However, when addressing the task of data-flow deobfuscation (for example, MBA simplification or VM instruction handlers behavior extraction from VM-based obfuscation) we are interested in finding a simpler expression that is semantically equivalent to the extracted symbolic expression, rather than checking for its satisfiability. We find that *raw* simplification techniques are heavily dependent on the syntactic complexity of the code being analyzed. Thus, an adversary might thwart the analysis capabilities by arbitrarily increasing the syntactic complexity of the obfuscated code introducing either artificial complexity (e.g. junk code) or algebraic complexity (e.g. MBA rewritings).

In order to overcome the scalability issues that arise from increased syntactic complexity and, specifically, to be able to address data-flow based code deobfuscation more effectively, we would like to reason about the semantics of the code instead of syntax. In this sense, there has been some recent work towards introducing *program synthesis* techniques aiming to *synthesize* the semantics of a particular snippet of code, presumably obfuscated. By reasoning about code semantics, we are no longer limited by the syntactic complexity of the underlying code, which can be arbitrarily increased, but only by its semantic complexity.

## 3.2 Fundamentals of program synthesis

### 3.2.1 Introduction

*Program synthesis* is the process of automatically constructing programs that satisfy a given specification. By specification, we mean to find a way of somehow "telling the computer what to do" and let the implementation details to be carried by the *synthesizer*.

**Remark 3.1.** Note that when talking about program synthesis, even if the definition is a little vague, we are not thinking of automatically constructing fully-fledged and complex computer programs like video games or web browsers, but very particular tasks that can be clearly described or that expose a *simple* observable semantic behavior.

A specification can be provided in different ways. Among the most common ones we find the following:

- A formal specification in some logic (e.g. first-order logic[1]). For example, if we would like to have a program $P$ that adds 7 to any 64-bit integer input, we could write the specification as:

$$\forall x \in \mathbb{Z}/2^{64}\mathbb{Z}, P(x) = x + 7$$

- A set of inputs and outputs that describe how the program should behave. For the example program described before, we could provide as the specification a list of input/output values like:

$$(0, 7), (-4, 3), (123, 130), (-368, -361) \ldots$$

- A reference implementation. Although it might seem strange, it will prove useful in several cases, including our treatment of data-flow code deobfuscation, as will be motivated below.

A formal specification leads to a *deductive* program synthesis style. In this case, given the specification and a set of logical axioms, we try to deduce a suitable implementation. For doing so, we would not only need a complete formal specification but also a complete axiomatization of the target language, which can be very difficult to obtain.

If we consider a more relaxed specification, we can leverage an *inductive* program synthesis style, which tries to find an implementation applying an iterative search technique. This approach allows a more flexible specification but can potentially run into scaling problems more easily.

[1] `https://en.wikipedia.org/wiki/First-order_logic`

### 3.2.2 Inductive oracle-guided program synthesis methods

We will focus on inductive techniques that are based on oracle-guided program synthesis. This type of program synthesis assumes that we already have an implementation of the program that we want to synthesize: an Input/Output (I/O) *oracle*. Then, we can treat this implementation as a *black-box* and obtain I/O pairs of values from it.

Different program synthesis methods can fit into an inductive oracle-guided approach. We will briefly describe some of the most common ones, which have also been used in different publications addressing code deobfuscation.

#### Counterexample-guided program synthesis

The main idea is to have two main components running in a loop. On the one hand, we have the *synthesizer* that will generate a candidate program that must satisfy the given specification. On the other hand, we have the *verifier* that will check if the proposed implementation is *correct*.

The basic steps that a counterexample-guided program synthesis algorithm (using an I/O oracle as its specification) perform are as follows:

- Query the I/O oracle with an input and obtain the corresponding output.
- Find a candidate program that satisfies the I/O behavior.
- Check the correctness of the proposed program.
  - If the program is not correct, return to first step.
- Return the synthesized program candidate.

At each iteration, instead of randomly generating another I/O pair from the oracle, the verification step will provide *feedback* that we can use to guide the next oracle query. This feedback essentially means that we would get a "meaningful" input to query the oracle with. This will be further detailed in Section 3.2.3.

#### Enumerative program synthesis

The differential aspect of enumerative program synthesis is that we generate an exhaustive list of potential program candidates. Later, we filter them following some criteria. In our case, we are interested in those that show the same I/O behavior as the I/O oracle.

The basic approach would simply check if some program from the list of possible candidates exposes the same I/O behavior and return it as the synthesized program. If we arrive to only one possible candidate, we might want to verify its semantic equivalence to the obfuscated program serving as oracle. As discussed in Section 3.2.3, this verification is not always possible or can be very time-consuming

Another possibility would be having several program candidates that meet the specification. In this case, we could proceed in different ways:

- Generating more (random) I/O tests to further filter potential candidates.
- Checking if the candidates are equivalent between them. If they are not, we could get some feedback to generate a "meaningful" input to query the I/O oracle with, as will be detailed in Section 3.2.3.
- If the number of candidates is *low enough* we could try to check directly if they are equivalent to the obfuscated oracle. This might run into the same time issues as before.

The code deobfuscation approach described in Section 3.3.2 leverages a flavor of enumerative program synthesis to address the simplification of MBA expressions.

#### Stochastic program synthesis

The idea is to convert the problem of finding a candidate program into a stochastic optimization problem. We initially provide a list of I/O pairs and try to determine the best candidate program that matches them. The biggest difference is that at each iteration we generate *intermediate* results instead of actual candidate programs, which evolve towards a global optima (i.e. the *best* candidate program) guided by a cost function. A common way to implement such approach is by using a Monte Carlo Tree Search[2] (MCTS) algorithm.

Stochastic program synthesis is at the core of the deobfuscation approach described in Section 3.3.1.

### 3.2.3 Practical considerations

It is important to take into account some practical aspects from program synthesis, specifically oracle-guided approaches that we have discussed.

First, the synthesis step must know *how* to construct candidate programs (also partial results, in the case of stochastic method). Essentially, we need to define the set of *primitive* components (e.g. the set of operands and operators in an MBA simplification context) and the ways in which they are allowed to be combined. A common way to formalize this is by defining a context-free grammar[3] that encompasses the primitive components (*terminals*) and the ways to combine them (*derivation/production rules*). In this sense, candidate programs will be derivations that only contain terminal symbols, while intermediate results that appear in the stochastic approach may contain non-terminal symbols as well.

Second, in any practical scenario we will need to define some limits to ensure that the program synthesis algorithm terminates. On the one hand, we should decide the maximum number of iterations that we want our algorithm to run for a counterexample-guided

[2] `https://en.wikipedia.org/wiki/Monte_Carlo_tree_search`
[3] `https://en.wikipedia.org/wiki/Context-free_grammar`

approach as well as for enumerative methods if we decide to provide some feedback to further filter initial outcome. The number of iterations is implicit in algorithms such as MCTS for a stochastic approach and should be limited as well. We should also limit the number of I/O pairs that the synthesis method will use to try to determine a candidate program. In the case of counterexample-guided synthesis this limit might apply to the initial amount of I/O pairs to generate the first candidate program and not the total number of iterations of the method (producing a new I/O pair at each). On the other hand, we should delimit the amount of possible candidate programs. If we generate them with a context-free grammar as described above, the natural way of doing so is by:

- Deciding the types of instructions and values (operands and operators) contained in the context-free grammar's alphabet of terminals.
- Fixing the maximum amount of derivation rules that can be applied to the initial variable to generate a candidate program.

Third, the verification step of exhaustive and counterexample-guided program synthesis usually involves the aid of an SMT solver. It can be used to check directly if the candidate program is equivalent to the oracle specification. If the result of such check is SAT, then we are done. If they are not equivalent, the SMT solver will produce an input for which they differ. This is the kind of "meaningful" input to query the oracle with at next iteration. In the case of counterexample-guided approach, we could also *ask* the SMT solver if there exists a different candidate program that satisfies all the I/O pairs provided, but differs for another input. If the SMT solver can find such alternative candidate program, we would get another "meaningful" input to query the I/O oracle with. This way we iteratively reduce ambiguity from candidate programs (even if we do not reach a semantically equivalent candidate to the obfuscated oracle).

Fourth, deciding whether a candidate program is *valid enough* is also a matter of the particular problem we are dealing with as well as the amount of uncertainty we are willing to take, depending on our needs. For example, we can decide that a candidate that satisfies the same I/O behavior for a fixed amount of I/O test cases is valid enough for us, even if its equivalence (or unambiguity) has not been formally proved.

Finally, as one might have already noticed, the boundary between different program synthesis approaches, specifically the ones discussed above, are somewhat diffuse. For example, in an enumerative algorithm, if we perform any iteration with a new "meaningful" query to the I/O oracle to refine the initial result, we are actually introducing some kind of counterexample guidance to the enumerative approach. Indeed, even the proposed hierarchy, showing three different submethods of a more general oracle-guided approach, has been presented in this way due to its convenience with respect to the problem of code deobfuscation that we are addressing. Just as an example, in [Bor15] they describe oracle-guided synthesis, enumerative synthesis and stochastic synthesis as submethods of a more general counterexample-guided program synthesis approach. The moral is that even if we define different approaches by abstraction of some common and representative criteria, they are not isolated and rigid building blocks, but flexible frameworks. Even

more, we might leverage different aspects of several approaches combined if doing so provides an overall better outcome, specially from a practical standpoint.

For more information regarding the field of program synthesis, the interested reader can refer to [Bla17a] and [Bor15] for a quick overview. Some examples building simple synthesizers can be found in [Bor18] or [Alb17]. For a detailed survey of program synthesis as of 2017, [GPS17] is an invaluable resource.

## 3.3 Existing work

In the context of reverse engineering, Rolf Rolles introduced in 2014 different problems that can be tackled taking advantage of program synthesis techniques. In particular, he addresses the following:

- Semi-automated synthesis of CPU emulators (inspired in [GT11]).
- Automated generation of deobfuscators for peephole-expansion obfuscators (inspired in [BA06]).
- Reconstruction of obfuscated, metamorphic code sequences (inspired in [Gul+11]).

For a detailed assessment, please refer to [Rol14].

In our context addressing data-flow based deobfuscation, and MBA simplification in particular, the choice of an oracle-guided program synthesis approach derives instinctively by the nature of our problem: we have a reference (obfuscated) implementation that we can treat as a *black-box* and use it as an I/O oracle. This is illustrated in the following example.

#### Motivating example

Consider the following function describing an obfuscated MBA expression [Bla17a]:

$$f(x,y,z) = (((x \oplus y) + ((x \wedge y) \times 2)) \vee z) + (((x \oplus y) + ((x \wedge y) \times 2)) \wedge z)$$

We can treat $f$ as a *black-box* and observe its behavior:

$$\begin{aligned} (1,1,1) &\longrightarrow \boxed{f(x,y,z)} \longrightarrow 3 \\ (2,3,1) &\longrightarrow \boxed{f(x,y,z)} \longrightarrow 6 \\ (0,-7,2) &\longrightarrow \boxed{f(x,y,z)} \longrightarrow -5 \\ &\dots \end{aligned}$$

Thus, our objective is to *learn* (or *synthesize*) a simpler function with the same I/O behavior:

$$h(x,y,z) = x + y + z$$

Although very scarce, there has been some work addressing code deobfuscation with program synthesis approaches. Jha et al. propose in [Jha+10] a counterexample-guided program synthesis approach to deobfuscate code. Even though they do not specifically address MBA simplification, their benchmarks show that the hardest programs to synthesize are the ones containing a mixing of operators, indicating that MBA expressions add difficulty to the process, as one could expect. Unfortunately, their synthesizer (named BRAHMA) has not been publicly released.

More recently, two research papers [Bla+17; DCC20] have been published, which specifically address MBA simplification for code deobfuscation leveraging different program synthesis approaches as a part of their process. We will briefly discuss them below.

### 3.3.1 Syntia: MCTS based stochastic program synthesis

In their paper [Bla+17], Blazytko et al. introduce a generic approach for simplifying assembly instruction traces based on an oracle-guided stochastic program synthesis approach on top of an MCTS algorithm. As a result of their research, a tool called *Syntia* was implemented and publicly released[4]. They demonstrate that this approach can be applied in several domains with significant success. In particular, they address:

- Simplification of MBA expressions.
- Learning the semantics of arithmetic VM instruction handlers.
- Synthesizing the semantics of Return Oriented Programming (ROP) gadgets[5].

Their approach consists of three distinct parts, which will be summarized below.

#### Trace dissection

In this initial step, the given assembly instruction trace is separated into unique subtraces that the authors call trace windows. It is important to note that the way in which the initial trace is divided highly impacts both the process and the usefulness of later program synthesis. Consider the case where a trace window ends at an intermediary computation step. Then, the synthesized expression might not be meaningful at all. Thus, the process of defining the trace windows' boundaries is done in a semi-automated fashion, depending on the problem that is being addressed.

#### Random sampling

After getting a trace window, the random sampling step will produce I/O pairs using the trace itself as an I/O oracle, describing its semantic behavior. To do so, the inputs and outputs are defined as follows:

- All registers and memory locations that are read before they are written in the trace window are considered as inputs.

[4] `https://github.com/RUB-SysSec/syntia`
[5] `https://en.wikipedia.org/wiki/Return-oriented_programming`

- All registers and memory locations that are written for the last time are considered as outputs.

**Program synthesis**

Once a set of I/O samples has been obtained, a stochastic program synthesis method guided by an MCTS algorithm is used in order to synthesize each output independently. The basic idea for this kind of program synthesis approach is described above in Section 3.2.2. It is relevant to mention that in order to produce both intermediate results and actual candidate programs for each synthesis task, a context-free grammar is used to formalize the possible programs that can be synthesized, as it is explained above. Also note that the implementation does not provide any verification step to check for the equivalence between the trace serving as the I/O oracle and the synthesized candidate program. Despite that, this approach has been shown to be effective and produce correct results (manually verified for the benchmark tests in the research paper).

The interested reader can refer to the original paper [Bla+17] for a detailed exposition as well as several tests and benchmarks. This work has also been presented as talk sessions in several conferences [Bla17b; BC17; Bla18].

### 3.3.2 QSynth: Offline enumerative program synthesis

A novel approach for code deobfuscation was presented in the very recent paper [DCC20], which is extensively based on previous work from [Con19]. This new approach is based on an enumerative program synthesis method and introduces some improvement ideas with respect to the stochastic approach used in [Bla+17].

Their approach consists of five distinct parts, which will be summarized below.

**Program tracing**

A trace $Tr \triangleq < ins_0, ins_1, \ldots, ins_n >$ is defined as a sequence of instructions producing side effects on registers and memory. Such a trace is obtained via Dynamic Binary Instrumentation[6] (DBI). If $\mathbb{C}$ denotes the set of all concrete states of a CPU (registers and memory) and we have a concrete state $C \in \mathbb{C}$, then $\rightsquigarrow$ is the concrete evaluation operator of an instruction on the concrete state. With this notation, the process of evaluating $ins_0$ on the concrete state $C_0$ to produce the updated concrete state $C_1$ is denoted by $C_0 \overset{ins_0}{\rightsquigarrow} C_1$.

**Dynamic Symbolic Execution (DSE)**

The considered DSE is performed on the obtained trace as a separated step after program execution. The set of free variables (registers or memory locations) is denoted by $\mathbb{V}ar$ and the set of constant values represented as bit-vectors by $\mathbb{V}al$. Then, we have the concrete state mapping $\mathbb{C} : \mathbb{V}ar \mapsto \mathbb{V}al$. Similarly, we can define a symbolic state mapping from

[6] `https://en.wikipedia.org/wiki/Instrumentation_(computer_programming)`

variables to their logical (i.e. symbolic) counterpart as $\mathbb{S} : \mathbb{V}ar \mapsto \Phi$. If $S \in \mathbb{S}$ represents a symbolic state, the DSE can be defined as $S_n \overset{ins_n*}{\rightsquigarrow} S_{n+1}$. That is, the successive application of the corresponding instruction semantics on the current symbolic state, where $\overset{*}{\rightsquigarrow}$ is the symbolic evaluation function and $n$ the instruction index in the trace.

Finally, if $\Pi$ is the set of all possible symbolic execution paths, we define $\pi \in \Pi$ as $\pi \triangleq < S_0, S_1, \ldots, S_n >$, i.e. the sequence of symbolic state updates.

#### Expression abstract syntax tree computation

A slicing criterion indicating to retrieve a logical expression of $v \in \mathbb{V}ar$ at an offset $n \in \mathbb{N}$ is denoted by $\rho \triangleq (v, n)$. Then, a backward slicing function $get\_expr : (\Pi \times \rho) \rightarrow \Phi$ is introduced. This function performs a recursive backward dependency lookup starting from symbolic state $S_n$ up to $S_0$ in order to find all logical variables expressions affecting $v$ at offset $n$. Thus, the output of $get\_expr(\pi, \rho)$ is an expression formula noted $\varphi_v^n$ that represents the symbolic value of $v$ at offset $n$. The expression $\varphi_v^n$ is structured as an Abstract Syntax Tree (AST) whose leaves are constants or variables. These free variables will be the inputs of the expression. The rest of the nodes are operators.

We can define a function that, given a vector of variables, returns an assignment of values as $assignment : \langle \mathbb{V}ar \rangle \rightarrow \langle \mathbb{V}al \rangle$. Then, the I/O oracle associated with $\varphi_v^n$ can be defined as $\mathcal{O}_\varphi : \langle \mathbb{V}al \rangle \rightarrow \mathbb{V}al$. Given a test input, $\mathcal{O}_\varphi$ returns the associated output after evaluation. Note that such an oracle can be defined for any subexpression (subAST) of $\varphi_v^n$.

#### Synthesis oracle

A general synthesis oracle is defined as $\mathcal{SO} : \Phi \rightarrow \{\Phi \cup \emptyset\}$ which is composed of two suboracles:

- $\mathcal{O}_\varphi : \langle \mathbb{V}al \rangle \rightarrow \mathbb{V}al$, the expression I/O oracle as defined above.
- $\mathcal{O}_S : \{O\} \rightarrow \Phi$, a function mapping a vector of outputs to expressions that produce $O$ from a test input $I$.

Essentially, applying a set of test inputs on the I/O oracle $\mathcal{O}_\varphi$ associated with $\varphi_v^n$ produces an output vector $O_\varphi$. If $O_\varphi$ belongs to $\mathcal{O}_S$ then $\mathcal{O}_S$ provides an expression $\varphi'$ exhibiting the same I/O behavior than $\varphi_v^n$ (with respect to the inputs used). If the synthesis failed, then no formula ($\emptyset$) will be returned.

The function $\mathcal{O}_S$ mapping output vectors to expressions is implemented as an exhaustive search on a context-free grammar. The *offline* aspect comes from the fact that the candidate programs are only generated once from the context-free grammar and can be reused afterwards, without needing to compute them again.

**Expression simplification**

The main idea is to iterate the expression (AST) using the $\mathcal{SO}$ to synthesize and replace parts separately. Synthesized subexpressions are replaced with placeholder variables until reaching a fix-point when no more substitutions can be made. The concrete algorithm proposed has been called QSynth.

**Remark 3.2.** In our opinion, even if the DSE component can provide some advantages in several scenarios where a dynamic trace can be obtained, and the offline component can potentially save time (with respect to a stochastic approach, for example), the most interesting contribution that this research provides is the ability to reduce obfuscated expressions into smaller subexpressions that can be synthesized on their own and then reconstructing the synthesis candidate for the whole expression. This allows to reduce significantly the complexity of the initial target expression, thus leading to a generally higher success rate in the synthesis process.

The interested reader can refer to the original paper [DCC20] for a detailed exposition, as well as a pseudocode description of the QSynth algorithm and several tests and benchmarks presented, including a comparison against [Bla+17] performance. Unfortunately, QTrace[7], the framework were this approach has been implemented, is not publicly available.

## 3.4 Limitations

In general, the limits of (oracle-guided) program synthesis itself apply to any method that leverages such an approach to synthesize simplified expressions for code deobfuscation. These limits might come from different sources:

- *Semantic complexity*: expressions that are inherently very complex, non-linear and with deep nesting level. The clearest example would be cryptographic algorithms, which present strong confusion and diffusion properties.
- *Non-determinism*: algorithms that can exhibit different behaviors on different runs, even for the same input, usually involving some kind of (pseudo)random process.
- *Point functions*: functions that always return the same output for all inputs except for a single distinguished input.

Thus, if we target some (obfuscated) code that leverages any of the limiting factors described, it will be practically impossible to be synthesized. In this case, further manual analysis would be required in order to determine if there might exist different parts of the code that could be synthesized separately.

We should also take into account that some degree of preliminary manual analysis is usually required. Among others, such a manual treatment apply to the following aspects that come prior to being able to leverage any program synthesis approach to address code deobfuscation:

[7] `https://blog.quarkslab.com/exploring-execution-trace-analysis.html`

- *Location of obfuscation*: In order to generate a trace for the particular snippet code we want to deobfuscate, we first need to locate it. This often requires to perform some kind of static or dynamic analysis to the program containing the obfuscated code.
- *Choice of trace windows*: Even if some basic hints are provided in [Bla+17] with respect to the particular problems they address, there is no general rule to (automatically) divide a trace into smaller subtraces. This problem is essentially equivalent to the problem of (automatically) deciding a slicing criterion in [DCC20], in which case it is left out-of-scope of their research, thus requiring some degree of manual analysis as well.

# Chapter 4

# Integration of Syntia within radare2

## 4.1 Implementation

We have developed a proof-of-concept tool called *r2syntia* that integrates the program synthesis capabilities of *Syntia* within the code analysis and emulation environment provided by *radare2*. The tool has been written in *Python3*. We briefly describe below the main components on top of which it is built and how they operate together.

### 4.1.1 Components

**radare2**

radare2 is a free and open source reverse engineering framework built from scratch and written in *C*, with support for multiple architectures, file formats and operating systems. Some of its capabilities are:

- Disassembly of binaries of several architectures and operating systems.
- Analysis of code and data.
- Low level debugging.
- Exploiting.
- Binary manipulation.

It provides a Command Line Interface (CLI) usually referred as the *radare2 shell*, whose commands are based on mnemonics. Some of its basic commands are:

- **s** - **s**eek
- **px** - **p**rint he**x**dump

- **pd** - **p**rint **d**isassembly
- **wx** - **w**rite he**x**pairs
- **wa** - **w**rite **a**ssembly
- **aa** - **a**nalyze **a**ll
- **ia** - **i**nfo **a**ll

By appending a question mark *'?'* to any command we can get *inline help* and a list of available *subcommands*.

radare2 is built to be extensible and scriptable by nature. Thus, the easiest way to build tools on top of it is by using the *r2pipe* API, which is available to multiple programming languages. r2pipe provides an extremely simple interface to interact with an instance of radare2, consisting only of four methods: *open()*, *cmd()*, *cmdj() quit()*. The first and last are self-explanatory. *cmd()* receives as input a radare2 command and returns the associated radare2 output. *cmdj()* provides JSON deserialization into native objects; receives as input a radare2 command with the suffix *'j'* indicating JSON output and returns a native object of the language (e.g. a *list* or *dictionary* in Python).

The interested reader can refer to [Gàm19b] for a detailed coverage of the radare2 capabilities and practical usage.

#### ESIL emulation engine

*Code emulation* is the process of simulating the execution of code of the same or different CPU. It is a technique usually related to running software (e.g. games) from old or non-native platforms. In the context of reverse engineering, we can leverage code emulation for several use cases:

- Understand a specific snippet of code.
- Avoid risks of native code execution.
- Help code analysis procedures.
- Explore non-native executables.

An *intermediate language* (or *representation*) is the language of an abstract machine that is designed to aid in the analysis of computer programs. They are vital for the process of (de)compilation.

*ESIL* (Evaluable Strings Intermediate Language) is an intermediate language specifically designed to provide emulation capabilities to the radare2 reverse engineering framework. ESIL is built around some basic ideas:

- Small set of instructions.

- Based on reverse polish notation (RPN).
- Designed with emulation and evaluation in mind, not necessarily to be human-friendly.

In general, when we refer to ESIL we are not only talking about the intermediate language, but also the actual emulation engine bundled inside radare2 built on top of this intermediate language. In this sense, the ESIL emulation engine has the following main features:

- Stack based.
- Virtually infinite memory and set of registers.
- Native register aliases.
- Ability to implement custom operations and call external functions.

The interested reader can refer to [Gàm19a; Gàm20] for a specific treatment of radare2's code emulation capabilities with ESIL.

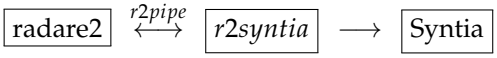
#### Syntia (reduced version)

*Syntia*[1] is a tool developed as a prototype implementation of the stochastic program synthesis method for code deobfuscation proposed in [Bla+17]. In order to integrate it with radare2 and, in particular, with ESIL, we are only interested in the program synthesis part of the approach (as described in Section 3.3.1). Thus, we will use a recent fork[2] of Syntia (by Tim Blazytko, one of the authors of [Bla+17]) that has been updated to Python3 and reduced to its MCTS core to synthesize the semantics of obfuscated code from its I/O behavior.

We have made some small modifications to the Syntia code base in order to fit our needs. The main changes affect the context-free grammar generation. Namely, we have slightly reduced the space of operators included in the terminal symbols of the grammar to the following ones $(+, -, *, \wedge, \vee, \oplus, \sim, -_{(unary)})$.

### 4.1.2 Integration

The idea is that we can call r2syntia from an active radare2 shell were we are performing the analysis of a binary that contains obfuscated code. Then, r2syntia will operate on top of radare2 (using r2pipe) to be able to use ESIL emulation engine in order to generate the I/O pairs of values for the specified variables (registers and memory locations) required by Syntia, which will finally synthesize the code semantics of the output variable with respect to the input variables.




[1] https://github.com/RUB-SysSec/syntia
[2] https://github.com/mrphrazer/syntia

The basic information required by r2syntia is the *bit size*, the *start offset*, the *finish offset*, the *input variables* and the *output variable*.

The interesting part is that all that information can be obtained by analyzing the code within radare2 itself. Hence, the main advantage of the proposed approach using r2syntia is the fact that the synthesis procedure can be leveraged at any time within an active reversing session. Thus, there is no need to step back from the current analysis in order to create a code trace, generate the random I/O sampling and then feed Syntia with them.

**Remark 4.1.** We provide a guided example in Appendix A. Starting from a snippet of non-obfuscated *C* code, we recreate the process of obfuscating it (using Tigress), analyzing the (obfuscated) compiled binary executable with radare2 to locate the obfuscated code (on the assembly level) and using r2syntia in order to retrieve the semantic behavior of the obfuscated code.

**Remark 4.2.** Installation instructions are provided in the code folder attached to this project. A Dockerfile is also included to ease the installation of the tools and environment.

## 4.2 Testing

### 4.2.1 Experimental environment

#### Hardware

The machine were the tests have been executed consists of:

- Processor: Intel i5-6300U CPU @ 2.40GHz (in total, 2 cores and 4 threads).
- Memory (RAM): 16GB DDR4 @ 2133MHz.

#### Software

Apart from the code provided in the attached folder, the exact software versions of the required tools used in the machine were the tests have been executed are:

- Operating system: Pop!_OS 20.04 LTS (based on Ubuntu 20.04 LTS).
- radare2: 4.5.0
- Python: 3.8.2
  - r2pipe python package: 1.4.2
  - z3-solver python package: 4.8.8.0

The versions stated above are provided for the sake of completeness, but r2syntia should be able to run without problems on any UNIX based operating system with any recent version of Python3 and radare2. The versions of r2pipe and z3-solver were provided automatically by the *pip* Python package installer. Thus, the default version installed by pip should work directly as well.

### 4.2.2 Description

We evaluate the simplification of 500 obfuscated MBA expressions using r2syntia. To do so, we use the same testbed provided by Syntia, which has been constructed in the following way:

- Create a *C* program which calls 500 randomly generated functions, each one taking 5 input variables and returning an expression of layer 3 to 5 (i.e. the number or derivations of the context-free grammar in order to generate the expression).
- Obfuscate the functions with Tigress v2.2 using *EncodeArithmetic*[3] and *EncodeData*[4] transformations. The former replaces the original expression with an equivalent MBA expression, as described in Section 2.2.1. The latter encodes integer arguments before calling the function and decodes them at return, corresponding to the *encodings* technique described in Section 1.2.1.

**Remark 4.3.** The original source file, the script used to obfuscate it, the generated obfuscated source file and the compiled obfuscated binary are provided in the code folder attached to this project.

### 4.2.3 Syntia configuration

Syntia offers customization of the algorithm via four different parameters:

- SA-UCT: Represents an adaptation to apply the characteristics of Simulated Annealing (SA) to the Upper Confidence Bound for Trees (UCT) in the context of an MCTS algorithm. It configures the trade-off between exploration and exploitation. Further details can be found in Sections 2.3, 2.4 and 4.1 of [Bla+17].
- MCTS iterations: the maximum number of MCTS iterations.
- I/O samples: the number of I/O pairs generated from the oracle.
- playout depth: represents the maximum playout depth, which is a measure that defines how often a non-terminal symbol can be mapped to rules that contain non-terminal symbols, as described in Section 4.3 of [Bla+17].

We use the same parameter configuration that is used in the original paper for evaluating the same testbed of 500 obfuscated MBA expressions:

| **Parameter** | **Value** |
|---|---|
| SA-UCT | 1.5 |
| MCTS iterations | 50000 |
| I/O samples | 50 |
| playout depth | 0 |

[3] `http://tigress.cs.arizona.edu/transformPage/docs/encodeArithmetic`
[4] `http://tigress.cs.arizona.edu/transformPage/docs/encodeData`

A detailed study of the parameters' choice in different scenarios, as well as a more in-depth treatment of the meaning and motivation behind the SA-UCT and playout depth parameters is out of scope of this work. The interested reader can refer to [Bla+17] for such a reference of these aspects.

### 4.2.4 Results

After running r2syntia on the testbed of 500 obfuscated MBA expressions, we obtained the following general results.

| **Measurement** | **Value** |
|---|---|
| Synthesized tasks | 493/500 (98.6%) |
| Total synthesis time (s) | 5155 (01:25:05) |
| Total success synthesis time (s) | 3165 (00:52:45) |
| Total fail synthesis time (s) | 1990 (00:33:10) |

The tasks that have not been successfully synthesized are the ones that exhausted the limit of 50000 iterations of the MCTS algorithm before reaching a synthesized expression. We observe that a 38.6% (1990/5155) of the time is spent entirely on the 7 tasks not synthesized. If we analyze the time required for individual tasks with respect to their success or failure, as well as the iterations required to complete the 493 successfully synthesized tasks, we obtain the following:

| **Measurement** | **Min** | **Max** | **Mean** |
|---|---|---|---|
| Fail synthesis time (s) | 279.72 | 291.72 | 284.29 |
| Success synthesis time (s) | 0.034 | 77.59 | 5.77 |
| Success synthesis iterations | 0 | 15016 | 1178.75 |

**Remark 4.4.** The minimum number of MCTS iterations for successfully synthesized tasks being 0 means that a candidate was found during first iteration.

These numbers suggest that most of the tasks successfully synthesized where completed in a relative low amount of time and small number of iterations. This behavior is confirmed by the distribution of time and iterations for the successfully synthesized tasks, as illustrated in Figure 4.1 and Figure 4.2.

### 4.2.5 Comparison

As suggested in [Bla+17], we might try to synthesize again the tasks that failed initially. As the program synthesis leveraged by Syntia has a stochastic nature, this could result in new successfully synthesized tasks. Indeed, this approach was used in the evaluation presented in the original paper. They successfully synthesized 448/500 tasks on first run, and only got to 495/500 after nine runs. However, their synthesis times were way faster than ours, in absolute values.

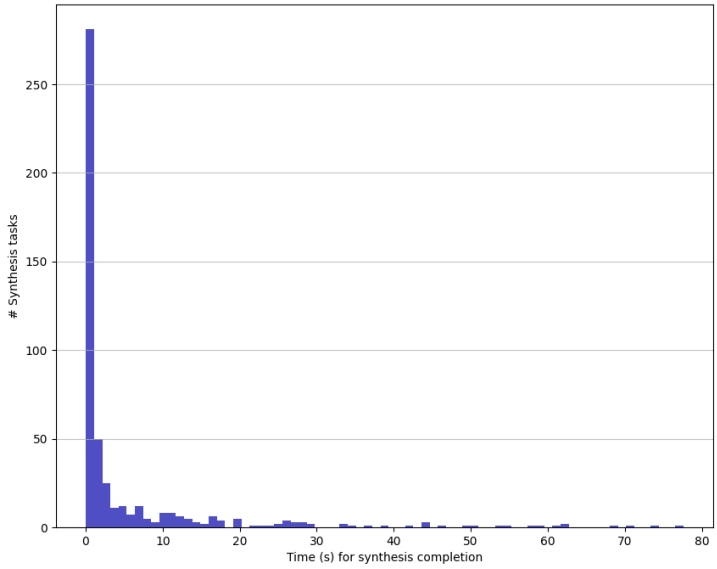



Figure 4.1: Number of tasks successfully synthesized in a certain amount of time (s)

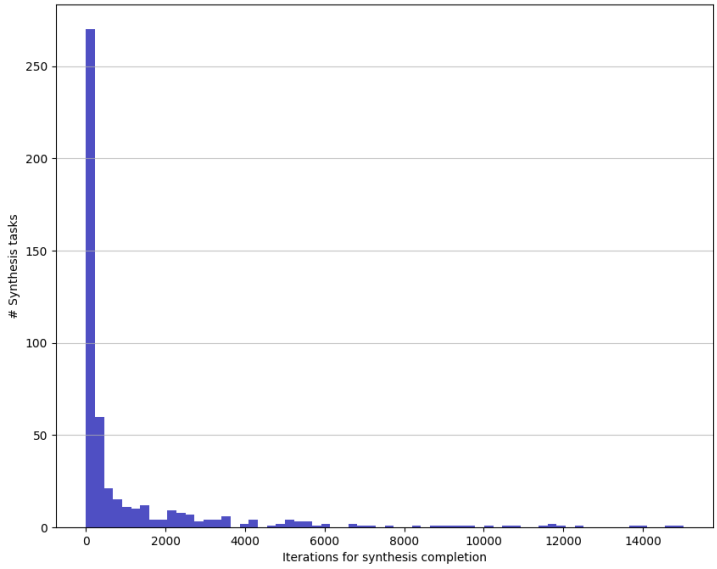



Figure 4.2: Number of tasks successfully synthesized in a certain number of iterations

Despite the apparent divergence in results, these differences are not surprising, but rather expected. On the one hand, it seems clear that our increased success rate comes from the modified (reduced) context-free grammar used to generate candidate expressions, effectively reducing the search space for the MCTS algorithm. On the other hand, it is obvious that their better time performance is due to the fact that the hardware environment where they run the tests is way more powerful than ours by orders of magnitude: two Intel Xeon E5-2667 CPUs (in total, 12 cores and 24 threads) and 96 GB of RAM.

**Remark 4.5.** If one tries to reproduce the evaluation of test cases, consider that running r2syntia different times might result in relatively variable results due to the stochastic nature of Syntia.

### 4.2.6 Improvement proposals

The current implementation of r2syntia has room for improvement in some clear directions. Probably, the most notable are:

- Add a timeout for a single synthesis task.
- Provide a mechanism to tune the Syntia configuration parameters (see Section 4.2.3).
- Allow to define I/O variables having different bit size than the global bit size.
- Allow to use memory locations for the I/O variables.
- Allow to synthesize different output variables at the same time.

It is relevant to mention that Syntia already has the ability to operate with all the described improvements and that radare2 trivially provides the extra required information from the binary. Thus, the only actual work needed to bring these added features to r2syntia would be to define a proper way in which the analyst is able to specify the values for the synthesis timeout, configuration parameters, variables' bit size, memory locations and multiple output variables. Probably, the easiest way to do so would be by registering r2syntia as a radare2's core plugin with associated commands and internal configuration variables within the radare2 shell. This possibility will be explored in the near future, alongside with open sourcing the r2syntia code.

**Remark 4.6.** Note that the different improvement ideas presented hereabove do not apply to the more general field of program synthesis for code deobfuscation, but specifically to current r2syntia proof-of-concept implementation. Some thoughts in the former direction will be provided in the conclusions.

# Conclusions

The main goal of this project was to analyze current state-of-the-art code deobfuscation techniques, with a strong focus on program synthesis approaches addressing data-flow deobfuscation based on MBA expressions. As a secondary goal, we wanted to contribute with practical proposals, test and validate them in a controlled environment. We have successfully accomplished those objectives through a set of specific achievements.

We have analyzed and presented common code obfuscation techniques used in malware threats and commercial software protection, providing basic ideas to assess their quality as well as some theoretical and practical aspects to take into account when addressing code deobfuscation.

During the theoretical development of MBA expressions, we introduced metrics to measure the complexity of such expressions and different simplification approaches. We also showcased the main techniques built of top of MBA expressions in order to perform data-flow code obfuscation.

We have studied the viability of program synthesis approaches to help in the process of code deobfuscation. Moreover, we presented recent research in this direction and discussed their limitations as well.

Finally, our implementation of r2syntia is, to our knowledge, the first integration of a program synthesis tool (and approach) into the workflow of a fully-fledged reverse engineering framework.

## Future work

On the one hand, we could go for a more theoretical continuation of the study of MBA expressions. As it is clearly derived from [Eyr17] and our exposition in Chapter 2, there are quite some open problems involving the formal treatment of MBA expressions. Just as an example, there is still a lot of research to be done involving identification of arithmetic operators from a given ANF representation of an MBA expression, specially multiplication and division.

On the other hand, and from a more practical standpoint, the easiest immediate step to take would be to incorporate the improvements discussed in Section 4.2.6 into r2syntia. Another clear path to follow would be implementing a system similar to the one described in [Con19; DCC20]. The basic idea is to split up obfuscated MBA expressions into smaller subexpressions that could be synthesized on their own. As discussed by Rolf Rolles in his summary[5] of [Con19], from this premise, we should be able to adapt this technique idea to work entirely on a static level, thus not needing to generate a dynamic trace. For the synthesis of subexpressions, we could also arise the possibility of replacing the exhaustive synthesis approach with other alternatives. It would also make a lot of sense to explore mechanisms that verify the semantic equivalence of the synthesized results (i.e. verify the soundness of the program synthesis approach) as well as detecting repeating patterns in order to memorize them and save time for subsequent synthesis tasks.

[5] `https://www.reddit.com/r/ReverseEngineering/comments/eoc0hj/combining_program_synthesis_and_symbolic/`

# Bibliography


[Alb17] Aws Albarghouthi. *A Program Synthesis Primer*. Apr. 24, 2017. URL: https://barghouthi.github.io/2017/04/24/synthesis-primer/ (visited on June 5, 2020).

[And18] D. Andriesse. *Practical Binary Analysis: Build Your Own Linux Tools for Binary Instrumentation, Analysis, and Disassembly*. No Starch Press, Incorporated, 2018. ISBN: 9781593279127. URL: https://books.google.es/books?id=laWgswEACAAJ.

[BA06] Sorav Bansal and Alex Aiken. "Automatic Generation of Peephole Superoptimizers". In: *SIGOPS Oper. Syst. Rev.* 40.5 (Oct. 2006), pp. 394–403. ISSN: 0163-5980. DOI: 10.1145/1168917.1168906. URL: https://doi.org/10.1145/1168917.1168906.

[BC17] Tim Blazytko and Moritz Contag. *Lets break modern binary code obfuscation*. Chaos Communication Congress (34C3). Dec. 27, 2017. URL: https://www.youtube.com/watch?v=TDnAkm6ZTYw (visited on June 8, 2020).

[BD06] Philippe Biondi and Fabrice Desclaux. "Silver Needle in the Skype". In: 2006.

[Bio+17] Fabrizio Biondi et al. "Effectiveness of Synthesis in Concolic Deobfuscation". In: *Computers and Security* 70 (Sept. 2017), pp. 500–515. DOI: 10.1016/j.cose.2017.07.006. URL: https://hal.inria.fr/hal-01241356.

[Bla+17] Tim Blazytko et al. "Syntia: Synthesizing the Semantics of Obfuscated Code". In: *26th USENIX Security Symposium (USENIX Security 17)*. Vancouver, BC: USENIX Association, Aug. 2017, pp. 643–659. ISBN: 978-1-931971-40-9. URL: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/blazytko.

[Bla17a] Tim Blazytko. *Introduction to program synthesis*. Feb. 24, 2017. URL: https://synthesis.to/presentations/introduction_to_program_synthesis.pdf (visited on June 7, 2020).

[Bla17b] Tim Blazytko. *Syntia: Synthesizing the Semantics of Obfuscated Code*. USENIX Security. Aug. 17, 2017. URL: https://www.youtube.com/watch?v=RANGyrCbLe8 (visited on June 8, 2020).

[Bla18] Tim Blazytko. *Breaking State-of-the-Art Binary Code Obfuscation via Program Synthesis*. Black Hat Asia. Mar. 22, 2018. URL: https://www.youtube.com/watch?v=0SvX6F80qg8 (visited on June 8, 2020).

[Bor15] James Bornholt. *Program Synthesis Explained*. Jan. 6, 2015. URL: https://www.cs.utexas.edu/~bornholt/post/synthesis-explained.html (visited on June 5, 2020).

[Bor18] James Bornholt. *Building a Program Synthesizer*. July 10, 2018. URL: https://www.cs.utexas.edu/~bornholt/post/building-synthesizer.html (visited on June 5, 2020).

[BP17] Sebastian Banescu and Alexander Pretschner. "A Tutorial on Software Obfuscation". In: Jan. 2017. DOI: 10.1016/bs.adcom.2017.09.004.

[CN09] Christian Collberg and Jasvir Nagra. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*. 1st. Addison-Wesley Professional, 2009. ISBN: 0321549252.

[Col20] Christian Collberg. *The tigress C obfuscator v3.1*. Feb. 13, 2020. URL: https://tigress.wtf (visited on June 2, 2020).

[Con19] Luigi Coniglio. *Combining program synthesis and symbolic execution to deobfuscate binary code*. Oct. 2019. URL: http://essay.utwente.nl/79934/.

[CTL97] Christian S. Collberg, Clark D. Thomborson, and Douglas Low. "A Taxonomy of Obfuscating Transformations". In: 1997.

[Dan+14] Bruce Dang et al. *Practical Reverse Engineering: X86, X64, ARM, Windows Kernel, Reversing Tools, and Obfuscation*. 1st. Wiley Publishing, 2014. ISBN: 1118787315.

[DCC20] Robin David, Luigi Coniglio, and Mariano Ceccato. "QSynth - A Program Synthesis based approach for Binary Code Deobfuscation". In: Jan. 2020. DOI: 10.14722/bar.2020.23009.

[DG05] Mila Dalla Preda and Roberto Giacobazzi. "Semantic-Based Code Obfuscation by Abstract Interpretation". In: *Automata, Languages and Programming*. Ed. by Luís Caires et al. Berlin, Heidelberg: Springer Berlin Heidelberg, 2005, pp. 1325–1336. ISBN: 978-3-540-31691-6.

[EGV16] Ninon Eyrolles, Louis Goubin, and Marion Videau. "Defeating MBA-based Obfuscation". In: *2nd International Workshop on Software PROtection*. Ed. by ACM. Vienna, Austria, Oct. 2016. DOI: 10.1145/2995306.2995308. URL: https://hal.archives-ouvertes.fr/hal-01388109.

[Eyr15] Ninon Eyrolles. *What theoretical tools are needed to simplify MBA expressions?* Sept. 23, 2015. URL: https://blog.quarkslab.com/what-theoretical-tools-are-needed-to-simplify-mba-expressions.html (visited on May 25, 2020).

[Eyr17] Ninon Eyrolles. "Obfuscation with Mixed Boolean-Arithmetic Expressions : reconstruction, analysis and simplification tools". Theses. Université Paris-Saclay, June 2017. URL: https://tel.archives-ouvertes.fr/tel-01623849.

[Gàm19a] Arnau Gàmez i Montolio. *A journey through ESIL: Understanding code emulation within radare2*. radare2 congress (r2con). Sept. 6, 2019. URL: https://www.youtube.com/watch?v=MaFafykTASw (visited on June 12, 2020).

[Gàm19b] Arnau Gàmez i Montolio. *Overcoming Fear: Reversing With Radare2*. Hack In The Box Security Conference Amsterdam (HITBSecConf). May 9, 2019. URL: https://www.youtube.com/watch?v=317dNavABKo (visited on June 12, 2020).

[Gàm20] Arnau Gàmez i Montolio. *Code emulation for reverse engineers: a deep dive into radare2's ESIL*. RuhrSec. May 26, 2020. URL: https://www.youtube.com/watch?v=4ATseh8aRTE (visited on June 12, 2020).

[GEV16] Adrien Guinet, Ninon Eyrolles, and Marion Videau. "Arybo: Manipulation, Canonicalization and Identification of Mixed Boolean-Arithmetic Symbolic Expressions". In: *GreHack 2016*. Proceedings of GreHack 2016. Grenoble, France, Nov. 2016. URL: https://hal.archives-ouvertes.fr/hal-01390528.

[GPS17] Sumit Gulwani, Alex Polozov, and Rishabh Singh. *Program Synthesis*. Vol. 4. NOW, Aug. 2017, pp. 1–119. URL: https://www.microsoft.com/en-us/research/publication/program-synthesis/.

[GT11] Patrice Godefroid and Ankur Taly. *Automated Synthesis of Symbolic Instruction Encodings from I/O Samples*. Tech. rep. MSR-TR-2011-123. Nov. 2011. URL: https://www.microsoft.com/en-us/research/publication/automated-synthesis-of-symbolic-instruction-encodings-from-io-samples/.

[Gul+11] Sumit Gulwani et al. "Synthesis of Loop-Free Programs". In: *PLDI'11, June 4-8, 2011, San Jose, California, USA*. June 2011. URL: https://www.microsoft.com/en-us/research/publication/synthesis-loop-free-programs/.

[Jha+10] Susmit Jha et al. "Oracle-Guided Component-Based Program Synthesis". In: *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering - Volume 1*. ICSE '10. Cape Town, South Africa: Association for Computing Machinery, 2010, pp. 215–224. ISBN: 9781605587196. DOI: 10.1145/1806799.1806833. URL: https://doi.org/10.1145/1806799.1806833.

[Jun+15] Pascal Junod et al. "Obfuscator-LLVM – Software Protection for the Masses". In: *Proceedings of the IEEE/ACM 1st International Workshop on Software Protection, SPRO'15, Firenze, Italy, May 19th, 2015*. Ed. by Brecht Wyseur. IEEE, 2015, pp. 3–9. DOI: 10.1109/SPRO.2015.10.

[JXY08] Johnson Harold Joseph, Gu Yuan Xiang, and Zhou Yongxin. "System And Method For Interlocking To Protect Software-mediated Program And Device Behaviours". Patent Application WO 2008/101341 A1 (World Intellectual Property Organization). Aug. 28, 2008. URL: https://lens.org/186-219-911-458-517.

[KW13] Dhiru Kholia and Przemysław Węgrzyn. "Looking Inside the (Drop) Box". In: *7th USENIX Workshop on Offensive Technologies (WOOT 13)*. Washington, D.C.: USENIX Association, Aug. 2013. URL: https://www.usenix.org/conference/woot13/workshop-program/presentation/kholia.

[Lab20] NTT Secure Platform Laboratories. *Advanced Binary Deobfuscation*. Feb. 11, 2020. URL: https://github.com/malrev/ABD (visited on June 1, 2020).

[LM91] Xuejia Lai and James L. Massey. "A Proposal for a New Block Encryption Standard". In: *Advances in Cryptology — EUROCRYPT '90*. Ed. by Ivan Bjerre Damgård. Berlin, Heidelberg: Springer Berlin Heidelberg, 1991, pp. 389–404. ISBN: 978-3-540-46877-6.

[Riv01] Ronald L. Rivest. "Permutation Polynomials Modulo 2w". In: *Finite Fields and Their Applications* 7.2 (2001), pp. 287–292. ISSN: 1071-5797. DOI: https://doi.org/10.1006/ffta.2000.0282. URL: http://www.sciencedirect.com/science/article/pii/S107157970090282X.

[Rol09] Rolf Rolles. "Unpacking virtualization obfuscators". In: *Proceedings of the 3rd USENIX Conference on Offensive Technologies* (Jan. 2009).

[Rol14] Rolf Rolles. *Program Synthesis in Reverse Engineering*. Dec. 15, 2014. URL: https://www.msreverseengineering.com/blog/2014/12/12/program-synthesis-in-reverse-engineering (visited on June 5, 2020).

[Rol18a] Rolf Rolles. *A walk-through tutorial, with code, on statically unpacking the finspy VM: part one, x86 deobfuscation*. Jan. 23, 2018. URL: https://www.msreverseengineering.com/blog/2018/1/23/a-walk-through-tutorial-with-code-on-statically-unpacking-the-finspy-vm-part-one-x86-deobfuscation (visited on June 1, 2020).

[Rol18b] Rolf Rolles. *Finspy VM part 2: VM analysis and bytecode disassembly*. Feb. 1, 2018. URL: https://www.msreverseengineering.com/blog/2018/1/31/finspy-vm-part-2-vm-analysis-and-bytecode-disassembly (visited on June 1, 2020).

[Rol18c] Rolf Rolles. *Finspy VM unpacking tutorial part 3: devirtualization*. Feb. 21, 2018. URL: https://www.msreverseengineering.com/blog/2018/2/21/finspy-vm-unpacking-tutorial-part-3-devirtualization (visited on June 1, 2020).

[War12] Henry S. Warren. *Hacker's Delight*. 2nd. Addison-Wesley Professional, 2012. ISBN: 0321842685.

[Zho+07] Yongxin Zhou et al. "Information Hiding in Software with Mixed Boolean-Arithmetic Transforms". In: *Information Security Applications*. Ed. by Sehun Kim, Moti Yung, and Hyung-Woo Lee. Berlin, Heidelberg: Springer Berlin Heidelberg, 2007, pp. 61–75. ISBN: 978-3-540-77535-5.

# Appendices

## A Guided example

We start with a *C* function $f498$ that receives as input five 64-bit unsigned integers and returns as another 64-bit unsigned integer the value obtained by multiplying the second and fourth arguments (see Figure A.1).

```
uint64_t f498 (uint64_t a, uint64_t b, uint64_t c, uint64_t d, uint64_t e)
{
  uint64_t r = (b * d);

  return r;
}
```

Figure A.1: Non-obfuscated *C* source code for $f498$ function

Another function *target*_498 is defined solely as a wrapper calling $f498$. This is done to further obfuscate the final code by encoding the arguments serving as inputs to $f498$. (see Figure A.2).

```
void target_498(uint64_t a, uint64_t b, uint64_t c, uint64_t d, uint64_t e){
    f498(a, b, c, d, e);
}
```

Figure A.2: Non-obfuscated *C* source code for *target*_498 function

Then, we proceed to obfuscate the code with Tigress v2.2. On the one hand, we apply the *EncodeArithmetic*[6] transformation to replace the original expression for the return value in $f498$ with an equivalent MBA expression, as described in Section 2.2.1. (see Figure A.3). On the other hand, we apply the *EncodeData*[7] transformation to encode the integer arguments in *target*_498 before passing them to $f498$. This transformation corresponds to the *encodings* technique described in Section 1.2.1 (see Figure A.4).

[6] http://tigress.cs.arizona.edu/transformPage/docs/encodeArithmetic
[7] http://tigress.cs.arizona.edu/transformPage/docs/encodeData

```
uint64_t f498(uint64_t a , uint64_t b , uint64_t c , uint64_t d , uint64_t e )
{
  uint64_t r ;

  {
  r = ((((1712295344850271019UL * (((774237912431848323UL + b) + 774237912431848323UL * (
  (802669609978884170 6UL - 1712295344850271019UL * b) | (1712295344850271019UL * d - 802669609978884170 7UL)
  )) * ((0xf5415ab881dc147dUL + d) - 774237912431848323UL * ((802669609978884170 6UL -
  1712295344850271019UL * b) | (1712295344850271019UL * d - 802669609978884170 7UL)))) -
  802669609978884170 7UL * ((774237912431848323UL + b) + 774237912431848323UL * ((802669609978884170 6UL -
  1712295344850271019UL * b) | (1712295344850271019UL * d - 802669609978884170 7UL)))) -
  802669609978884170 7UL * ((0xf5415ab881dc147dUL + d) - 774237912431848323UL * ((802669609978884170 6UL -
  1712295344850271019UL * b) | (1712295344850271019UL * d - 802669609978884170 7UL)))) +
  611939574135942807 6UL) + (((1712295344850271019UL * (((774237912431848323UL + b) + 774237912431848323UL
  * ((802669609978884170 6UL - 1712295344850271019UL * b) | (1712295344850271019UL * (108118960912392268 7UL
  - d) - 802669609978884170 7UL))) * ((774237912431848323UL + (108118960912392268 7UL - b)) +
  774237912431848323UL * ((802669609978884170 6UL - 1712295344850271019UL * (108118960912392268 7UL - b)) |
  (1712295344850271019UL * d - 802669609978884170 7UL)))) - 802669609978884170 7UL * ((774237912431848323UL
  + b) + 774237912431848323UL * ((802669609978884170 6UL - 1712295344850271019UL * b) |
  (1712295344850271019UL * (108118960912392268 7UL - d) - 802669609978884170 7UL)))) - 802669609978884170 7UL
  * ((774237912431848323UL + (108118960912392268 7UL - b)) + 774237912431848323UL * ((802669609978884170 6UL
  - 1712295344850271019UL * (108118960912392268 7UL - b)) | (1712295344850271019UL * d -
  802669609978884170 7UL)))) + 611939574135942807 6UL)) - 927713760777885505UL;
  return (1712295344850271019UL * r - 802669609978884170 7UL);
}
}
```

Figure A.3: Obfuscated *C* source code for *f*498 function

```
void target_498(uint64_t a , uint64_t b , uint64_t c , uint64_t d , uint64_t e )
{


  {
  f498(774237912431848323UL * a + 927713760777885505UL, 774237912431848323UL * b + 927713760777885505UL,
       774237912431848323UL * c + 927713760777885505UL, 774237912431848323UL * d + 927713760777885505UL,
       774237912431848323UL * e + 927713760777885505UL);
  return;
}
}
```

Figure A.4: Obfuscated *C* source code for *target*_498 function

After we generate the obfuscated *C* code, we can simply compile it into a binary executable. This executable will contain the obfuscated code as assembly (ASM) instructions. In our case, we compiled it for the Intel x86-64 architecture. Thus, this will be the flavor of assembly we will be dealing with.

$$\boxed{\text{Non-obfuscated } C} \xrightarrow{Tigress} \boxed{\text{Obfuscated } C} \xrightarrow{Compiler} \boxed{\text{Obfuscated } ASM}$$

Now we open the binary executable with radare2 and proceed to perform some basic analysis with command *aa*. This command will essentially find functions within the executable, define them in the context of the current radare2 session and perform some basic analysis to determine their parameters and internal variables. We will be able to access to the defined functions' offsets by a flag name. Note that in our case the flag name will come directly from the original name in *C* source, as the testing binary we generated is non-stripped[8].

In order to obtain a disassembly listing of the function *target_498* we use the following command: $pdf$ @ *sym.target*_498. This command can be read as "print the disassembly listing of the function located at the offset pointed by the flag sym.target_498". Besides the raw disassembly of the instructions that form the function, the output of the previous command will also provide a high level representation of the function declaration as well as meaningful comments (everything after a semicolon ';'). Both the declaration and comments have been automatically added during the former analysis. Figure A.5 shows the radare2's output from this command.

Observe that after the input arguments for *target_498* have been encoded, we have the call (at offset $0x0041b320$) to the function *f498*, which has been given the flag name $sym.f498$. We can list the disassembly of this function in the same way as we did before using the following command: $pdf$ @ $sym.f498$. We include in Figure A.6 the full disassembly of the function $f498$ provided by radare2. Please note that the code in this figure is not intended to be effectively read, but rather to provide a sense of the *dimension* of the obfuscated function $f498$ in the assembly level.

Before running r2syntia, we need to know the information required by it. The *bit size* will be 64, of course. The *start offset* and *finish offset* determine the boundaries of the code to be deobfuscated. These offsets will be used as the boundaries for the I/O generation with ESIL emulation as well. In this case, we are interested in determining the semantics of the obfuscated code starting at the beginning of the function *target*_498 until its end, which will include the call to the function $f498$. Thus, the *start offset* will be $0x0041b264$ and the *finish offset* will be $0x0041b327$. The *input variables* will be the registers that hold the passing arguments to the function calls: *rdi*, *rsi*, *rdx*, *rcx* and *r*8. The *output variable* will be the register *rax*, which holds the return value.

[8] `https://en.wikipedia.org/wiki/Stripped_binary`

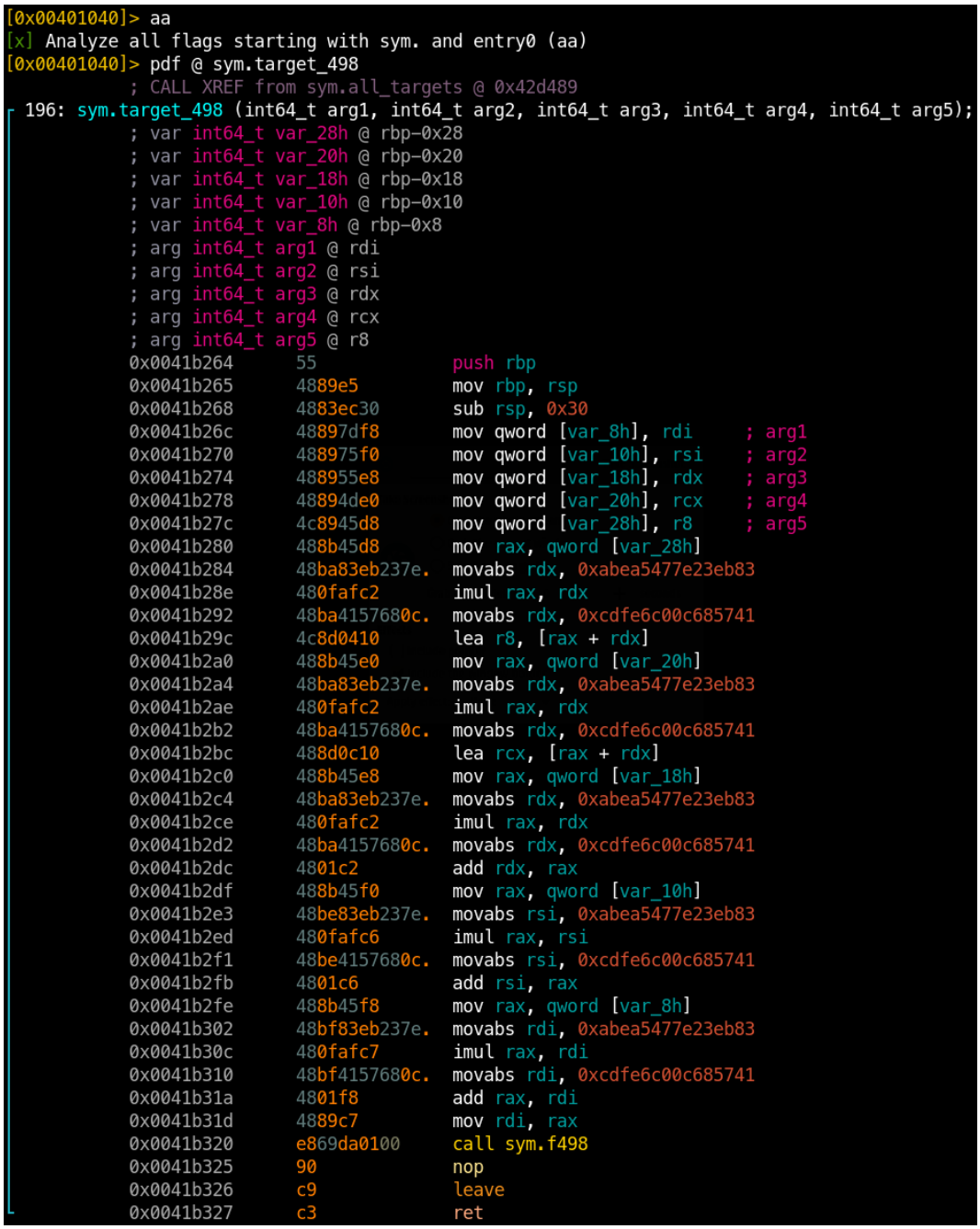



Figure A.5: Obfuscated *ASM* instructions for *target_498* function

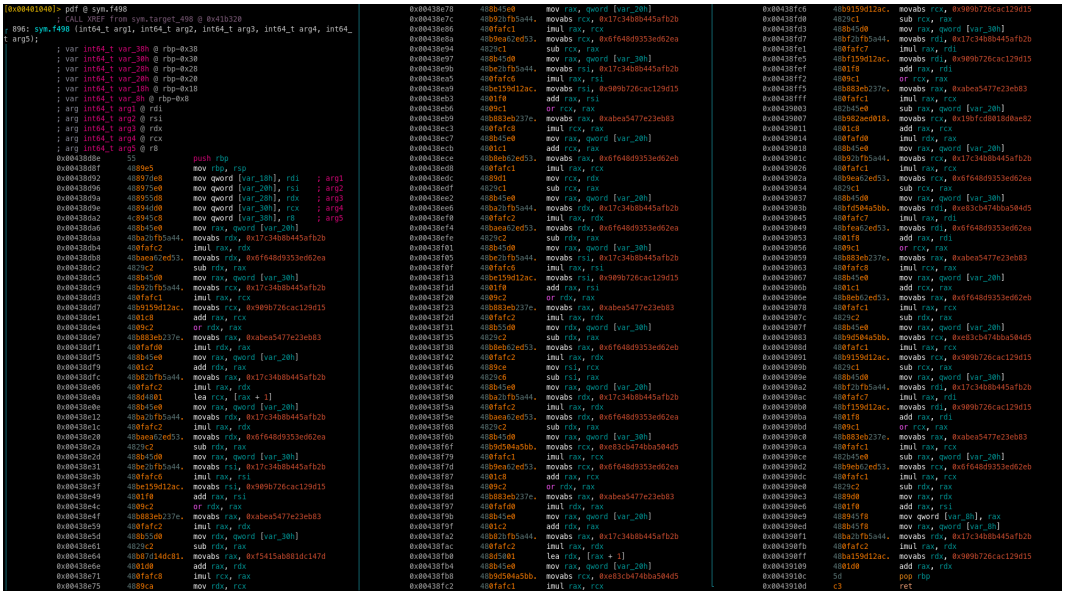


Figure A.6: Obfuscated *ASM* instructions for $f498$ function

Note that in this example the obfuscated code is exactly defined as functions such that it is trivial to obtain the input and output variables used for the corresponding calling convention[9]. The input variables were also displayed as comments added by radare2 in the previous disassembly listing of the functions. In a more general scenario, some manual analysis would be required in order to determine the exact boundaries for the obfuscated code as well as its input and output variables.

Now that we have all the information required by r2syntia, we can run it directly within the radare2 shell itself with the following command:

*#!pipe python [path to r2syntia] [bit size] [start offset] [finish offset] [inputs] [output]*

```
[0x00401040]> #!pipe python ./r2syntia.py 64 0x0041b264 0x0041b327 rdi rsi rdx rcx r8 rax


[*] General information:
---
Bit size: 64
Start at offset: 0x0041b264
Finish at offset: 0x0041b327
Inputs: rdi, rsi, rdx, rcx, r8
Output: rax
===
```

Figure A.7: r2syntia: Invocation and general information

[9] `https://en.wikipedia.org/wiki/X86_calling_conventions#x86-64_calling_conventions`

For this guided example, we have purposely added a lot of verbose information to be displayed during the execution of r2syntia, so the whole process is easier to follow. Such level of verbosity is left in the code attached with this project so the interested reader can reproduce the example more faithfully, but it could be adjusted for future releases. Figure A.7 shows the invocation of r2syntia from the radare2 shell as well as the general information that has been passed. After that, we generate the pairs of random I/O pairs using ESIL. The exact I/O pairs generated that will be used for the current synthesis task are displayed in Figure A.8.

```
[*] Generate I/O pairs:
---
#01 | (9, 1, 1, 1381152488, 24557) -> 1381152488
#02 | (2, 3, 116, 118, 12) -> 354
#03 | (3949893702, 51578, 1, 9, 118) -> 464202
#04 | (2, 118, 2240261273, 14, 3) -> 1652
#05 | (51737, 51737, 3, 116, 10866561597064912082) -> 6001492
#06 | (2, 3, 51737, 38955, 2) -> 116865
#07 | (1981552985124836012, 8198815974032266959, 3011061124, 2, 51737) -> 16397631948064533918
#08 | (1, 0, 3, 0, 10866561597064912082) -> 0
#09 | (2, 51578, 0, 31789, 1) -> 1639613042
#10 | (1981552985124836012, 8198815974032266959, 3011061124, 2, 0) -> 16397631948064533918
#11 | (2516679044, 1, 2, 249, 64612) -> 249
#12 | (1904909487, 31789, 724252492, 17911, 15) -> 569372779
#13 | (10866561597064912082, 1, 1, 9838140950697154623, 64612) -> 9838140950697154623
#14 | (1, 192, 12572928737245035653, 11753754609224741670, 11) -> 6218107978585103488
#15 | (2516679044, 118, 1, 724252492, 1) -> 85461794056
#16 | (1, 116, 2, 1, 3949893702) -> 116
#17 | (31789, 1, 1, 15, 10866561597064912082) -> 15
#18 | (1, 3011061124, 77062414292954027 31, 1, 0) -> 3011061124
#19 | (9, 1, 5, 14, 10365) -> 14
#20 | (14, 727083992, 12, 9, 9) -> 6543755928
#21 | (12, 12, 0, 1, 5) -> 12
#22 | (1, 12572928737245035653, 2516679044, 31789, 11) -> 13674527291293060961
#23 | (1, 51578, 1904909487, 8198815974032266959, 38955) -> 5369162918503966118
#24 | (1, 1, 24557, 2516679044, 3) -> 2516679044
#25 | (1, 2, 9, 1, 77062414292954027 31) -> 2
#26 | (17911, 2, 5272181135695228351, 1, 1) -> 2
#27 | (727083992, 51578, 3, 2, 15) -> 103156
#28 | (1, 249, 12572928737245035653, 8198815974032266959, 11753754609224741670) -> 12363329425983795031
#29 | (2, 1, 8198815974032266959, 2310048880236725136, 2) -> 2310048880236725136
#30 | (0, 2, 179, 2749797051, 1) -> 5499594102
#31 | (3, 192, 2, 1, 38955) -> 192
#32 | (31789, 2, 1, 11, 1) -> 22
#33 | (0, 179, 24557, 15, 10866561597064912082) -> 2685
#34 | (49, 0, 0, 1, 49) -> 0
#35 | (2, 2, 2, 1, 3) -> 2
#36 | (9, 1, 2, 11, 1) -> 11
#37 | (114, 2, 1, 64612, 49) -> 129224
#38 | (9, 12002997734282371094, 5457, 1, 51578) -> 12002997734282371094
#39 | (1381152488, 8198815974032266959, 3, 9, 1) -> 2367471452196167
#40 | (724252492, 1381152488, 3, 2240261273, 0) -> 3094142430973997224
#41 | (14, 77062414292954027 31, 0, 3949893702, 10365) -> 17942697549486332994
#42 | (2, 5, 249, 9, 31789) -> 45
#43 | (2, 12002997734282371094, 0, 2749622122, 116) -> 2433232134294676252
#44 | (12002997734282371094, 14, 249, 15, 192) -> 210
#45 | (12002997734282371094, 11753754609224741670, 0, 31789, 2516679044) -> 13040596583449655 50
#46 | (1, 31789, 179, 724252492, 1) -> 23023262468188
#47 | (1904909487, 12, 2240261273, 2749622122, 724252492) -> 32995465464
#48 | (64612, 11, 38955, 1, 77062414292954027 31) -> 11
#49 | (15, 12002997734282371094, 12572928737245035653, 64612, 14) -> 18122004629301637272
#50 | (1, 14, 2749797051, 3949893702, 5457) -> 55298511828
===
```

Figure A.8: r2syntia: Generated I/O pairs

Then, the context-free grammar that will be used to construct the candidate programs is created. Its components are explicitly displayed as can be seen in Figure A.9. Please note that the last terminal symbol represents the unary operator *neg*, i.e. computes the two's complement negative value. When displaying this verbose output, we chose to use the same symbol for it as the one used for the subtraction since it is commonly represented in this way, but note that the internal implementation (and representation) is of course different. Also note that even if we display operations in infix notation during the synthesis process for the sake of readability, their internal representation uses reverse polish notation (RPN) so we do not need to include parentheses symbols in our grammar.

```
[*] Context-free grammar information:
---
Non-terminals (variables): { u64 }
Terminals: { rdi, rsi, rdx, rcx, r8, +, -, *, &, |, ^, ~, - }
Production rules (RPN):
  u64 -> u64 u64 +      ≡ (u64 + u64)  [infix]
  u64 -> u64 u64 -      ≡ (u64 - u64)  [infix]
  u64 -> u64 u64 *      ≡ (u64 * u64)  [infix]
  u64 -> u64 u64 &      ≡ (u64 & u64)  [infix]
  u64 -> u64 u64 |      ≡ (u64 | u64)  [infix]
  u64 -> u64 u64 ^      ≡ (u64 ^ u64)  [infix]
  u64 -> u64 ~          ≡ (~ u64)      [infix]
  u64 -> u64 -          ≡ (- u64)      [infix]
  u64 -> rdi
  u64 -> rsi
  u64 -> rdx
  u64 -> rcx
  u64 -> r8
Start variable: u64
===
```

Figure A.9: r2syntia: context-free grammar used for the synthesis

Finally, the actual synthesis process takes place. Figure A.10 shows a representation of the MCTS operation, showing the intermediate results (i.e. partial derivations from the context-free grammar) with the associated reward. The synthesis finishes successfully when obtaining a reward of 1.0 for a program candidate. In this case we find that the semantics described by the I/O relations of the obfuscated assembly code (that is, the semantics of the output *rax* with respect to the inputs *rdi*, *rsi*, *rdx*, *rcx* and *r*8) equals to the multiplication of *rcx* and *rsi* registers, which are the second and fourth arguments of the obfuscated function. That is, we found that the semantic behavior of the obfuscated code simply returns the value obtained by multiplying the second and fourth arguments received, which is the exact behavior of the original non-obfuscated *C* code.

```
[*] Starting main synthesis:
---
rdx (0 iterations) (reward: 0.74067205546516)
(u64 | u64) (1 iterations) (reward: 0.3885388146264815)
(u64 + u64) (0 iterations) (reward: 0.755910217292746)
rcx (3 iterations) (reward: 0.8052417929776408)
r8 (1 iterations) (reward: 0.693119848776227)
(u64 - u64) (4 iterations) (reward: 0.7595575455018446)
rcx (2 iterations) (reward: 0.8052417929776408)
r8 (0 iterations) (reward: 0.693119848776227)
rsi (1 iterations) (reward: 0.8122961880577219)
rsi (4 iterations) (reward: 0.8122961880577219)
(u64 + u64) (8 iterations) (reward: 0.7623251733132093)
(~ u64) (2 iterations) (reward: 0.5054530337752783)
rsi (11 iterations) (reward: 0.8122961880577219)
rsi (0 iterations) (reward: 0.8122961880577219)
(u64 & u64) (10 iterations) (reward: 0.8052417929776408)
(~ u64) (1 iterations) (reward: 0.5007939956251491)
(u64 | u64) (3 iterations) (reward: 0.746689809332208)
(u64 * (u64 ^ u64)) (17 iterations) (reward: 0.779549051680586)
(u64 * u64) (3 iterations) (reward: 0.5161429589243006)
(u64 * rdx) (24 iterations) (reward: 0.7820353721822704)
(u64 ^ u64) (5 iterations) (reward: 0.6813433303294846)
(u64 + rsi) (28 iterations) (reward: 0.8674756017423928)
(- (u64 ^ u64)) (32 iterations) (reward: 0.7824086941189229)
(u64 + u64) (8 iterations) (reward: 0.7271283446946413)
(u64 + (u64 & u64)) (35 iterations) (reward: 0.8646243923335981)
(u64 | u64) (9 iterations) (reward: 0.8122961880577219)
(u64 & u64) (12 iterations) (reward: 0.7773685366599268)
(u64 + (u64 | u64)) (17 iterations) (reward: 0.8261261778091818)
(u64 + rsi) (55 iterations) (reward: 0.8646243923335981)
(rcx * rsi) (93 iterations) (reward: 1.0 by random playout)
(u64 * rsi) (93 iterations) (reward: 1.0)
===

[*] Final expression found (w/ reward: 1.0):
---
(rcx * rsi) (93 iterations)
rcx*rsi (simplified)
===

[*] Synthesis finished in 0.4010930061340332 seconds
```

Figure A.10: r2syntia: synthesis process and semantic behavior extraction

# B Planning

The project was divided in concrete tasks to be done with an allocated amount of time to complete each one. Roughly speaking, we set two weeks for each task regarding study and research and one week for each task related to the writing of the report. The concrete task breaking and timing allocation is presented as a Gantt diagram in Figure B.1.

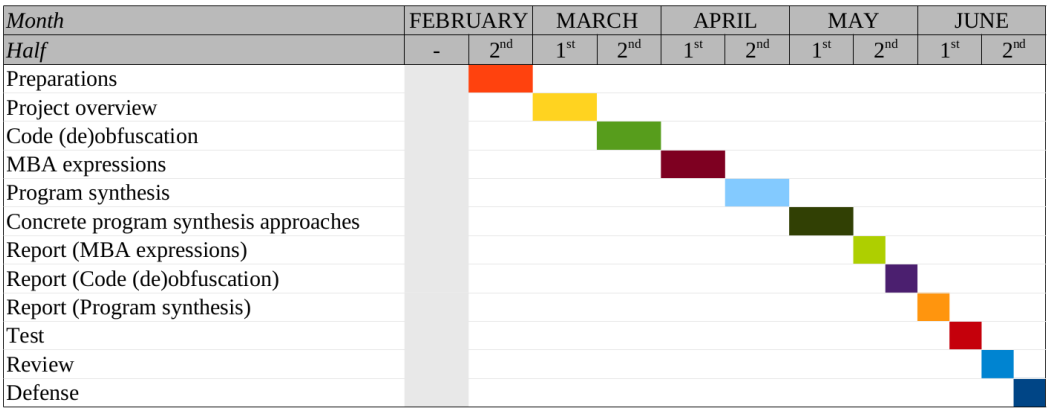



Figure B.1: Gantt diagram representing the project's planning

For each task, we assigned a *description*, *objective* and *outcome* as it is further detailed below:

- [16 - 29 Feb.] Preparations
  - Description: Manage bureaucracy between University of Barcelona and Eurecat. Find and organize resources.
  - Objective: Solve any formal issues with the realization of the project. Get a sense of existing work done related to the field to study.
  - Outcome: Grant with Eurecat to do the project. Initial backlog of potentially useful resources to explore.
- [1 - 15 Mar.] Project overview
  - Description: Review related posts/talks on the topic. Contact with other researchers.
  - Objective: Narrow the topic for the project, getting a clear big picture of the most important aspects to be studied, serving as a basic guide.
  - Outcome: Slides and remote presentation of the project to IT Security group at Eurecat. Draft of initial approach to the project.
- [16 - 31 Mar.] Code (de)obfuscation

  - Description: Study common techniques for code (de)obfuscation.
  - Objective: Understand the basics of code obfuscation techniques: how to classify them and their possible interactions. Moreover, understand the different approaches to code (de)obfuscation, not only theoretically but also from a more practical reverse engineering standpoint.
  - Outcome: Listing of references addressing the problem of code (de)obfuscation: considerations and common techniques.

- [1 - 15 Apr.] MBA expressions
  - Description: Theoretical study of MBA expressions and applicability for code obfuscation.
  - Objective: Establish a grounding foundation for the understanding of MBA expressions formally and its relation to code obfuscation.
  - Outcome: Listing of references discussing MBA expressions aiming to a theoretical formalization and oriented to obfuscation.

- [16 - 30 Apr.] Program synthesis
  - Description: Study about program synthesis in general.
  - Objective: Understand the basics of program synthesis techniques.
  - Outcome: Listing of references and guided examples introducing the field of program synthesis.

- [1 - 15 May] Concrete program synthesis approaches
  - Description: Study concrete approaches of program synthesis applied for code deobfuscation and, in particular, for simplification of MBA expressions.
  - Objective: Understand the current techniques and implementations using program synthesis approaches for code deobfuscation and simplification of MBA expressions.
  - Outcome: Listing of references and resources using program synthesis approaches to code deobfuscation and simplification of MBA expressions.

- [16 - 24 May] Report (MBA expressions)
  - Description: Write the chapter on MBA expressions for the report.
  - Objective: Provide a basic theoretical foundation for MBA expressions, including different approaches to perform MBA obfuscation, complexity metrics and a discussion about simplification approaches. Combine it in an organized and structured chapter.
  - Outcome: Written chapter on MBA expressions for the report.

- [25 - 31 May] Report (Code (de)obfuscation)
  - Description: Write the chapter on code (de)obfuscation for the report.

  - Objective: Combine the different resources studied related to fundamentals of code (de)obfuscation in an organized and structured chapter providing the general considerations as well as practical techniques.
  - Outcome: Written chapter on code (de)obfuscation for the report.

- [1 - 7 Jun.] Report (Program synthesis)
  - Description: Write the chapter on program synthesis for the report.
  - Objective: Combine the different resources studied related to program synthesis and its application to code deobfuscation in an organized and structured chapter detailing the general idea as well as practical current approaches and implementations.
  - Outcome: Written chapter on program synthesis for the report.

- [8 - 14 Jun.] Test
  - Description: Test open source program synthesis approaches for code deobfuscation
  - Objective: Test and compare the application of different program synthesis approaches for code deobfuscation based on simplification of MBA expressions.
  - Outcome: Last chapter (or appendix) at the report showing the tests and results.

- [15 - 21 Jun.] Review
  - Description: Review the final report.
  - Objective: Correct orthographic errors. Polish details. Make sure all formal aspects (structure, paging, referencing, titles...) are correct.
  - Outcome: A final version of the report to be deposited (deadline 21st jun.).

- [22 - 30 Jun.] Defense
  - Description: Prepare the defense of the work that will be held in front of examining court.
  - Objective: Create slides for the defense. Rehearse: timing, speech, possible questions...
  - Outcome: Presentation in form of slides. Potentially a recorded video or remote presentation to be given.

Apart from some minor delays (2-3 days) during the weeks devoted to writing the first three chapters, we managed to follow the initial planning almost entirely. Probably the most relevant mention would be that the week devoted to the *Test* task ended up with a last chapter *and* an appendix, instead of just one of them.

# C Methodology

## C.1 Individual organization

The resources needed to study and go through for the project came from very different format sources: papers, thesis, talks, courses, blogs, etc. Moreover, some degree of experimentation and tests were needed, mostly in the last phase of the project. Because of that, the organization of the different subtasks for the completion of each task of the project has been structured following the agile process of a Kanban board, generalized for all kinds of different subtasks.

In particular, a board in the Trello software has been used to track all the pending tasks, each one having its card. We used several lists on the board, namely:

- *Backlog*: General list with any relevant material. All resources and anything new that is found and could be useful will be put in this list.
- *Queue*: Tasks from backlog that are more relevant or have higher priority are moved here. Order of tasks in this list becomes meaningful.
- *Week*: Tasks planned for current week.
- *Doing*: Tasks being currently worked.
- *Done*: Tasks completed.

This organization offers several benefits. It lets us visualize the big picture of the project as it evolves. Moreover, it helps to limit the number of tasks being worked on simultaneously.

## C.2 Code and report

The required code development has been managed with git control version system and stored remotely on GitHub.

The report has been written in LaTeX typesetting system. We used the online web application Overleaf as the LaTeX editor. The Overleaf project was synced with a git repository stored at GitHub. This let us to manage a version control for the report itself, as well as being able to work offline with a local cloned version of the repository if needed.

## C.3 Contact with directors

Due to the highly specificity of the project's topic, most of the work has been carried out independently. Contact with University directors has been sporadic, mainly to inform of the advances and to ask for feedback for the report as it was being written. Contact with Eurecat's director has been a little more frequent and detailed. In both cases, most of the contact has been done remotely (by mail or other official means).