Universitat Oberta de Catalunya

Master's thesis

Analysis and applications of orthogonal approaches to simplify Mixed Boolean-Arithmetic expressions

Arnau Gàmez i Montolio

supervised by

Enric Hernández Jiménez

May 31, 2022

Abstract

A Mixed Boolean-Arithmetic (MBA) expression is composed of integer arithmetic operators, e.g. $(+, -, \times)$ 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.

In this project, we study and explore approaches to MBA simplification which are orthogonal to current techniques relying on a combination of symbolic execution and program synthesis. The main idea is to be able to extract some *information* from the underlying *mathematical structure* of such expressions. Then, this information can be used either alone or in combination with other techniques to aid in the task of MBA simplification. The focus is set upon recent research literature that develops such ideas, mainly addressing the problem of providing a *normalized* representation of (a subset of) linear MBA expressions as linear combinations with respect to an arbitrary set of *minimal* operators and expressions that form a basis of an *ad hoc* vector space where such linear MBA expressions live.

We analyze the contributions, flaws and limitations of recent research in this regard, and provide practical applications. In particular, we leverage a semantics preserving transformation which reduces the alternation of arithmetic and bitwise operators of an MBA expression in the context of program synthesis based code deobfuscation. This transformation is then used to aid in the problem of verifying the semantic correctness of a synthesized candidate expression, thus improving the soundness of such technique.

Resum

Una expressió Mixta Booleana-Aritmètica (MBA) està formada per operadors aritmètics sobre enters, per exemple $(+, -, \times)$ 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.

En aquest projecte, explorem enfocaments ortogonals a les tècniques actuals per tractar la simplificació d'expressions MBA, les quals es basen en l'ús combinat d'execució simbòlica i síntesi de programes. La idea principal és aconseguir extreure *informació* subjacent a *l'estructura matemàtica* d'aquestes expressions. Així, podem fer servir aquesta informació, siga per si mateixa o en combinació amb altres tècniques, per facilitar la tasca de simplificar expressions MBA. Posem èmfasi en l'estudi d'alguns articles acadèmics recents que desenvolupen aquestes idees, majoritàriament dirigits a proporcionar una representació *normalitzada* (d'un subconjunt) d'expressions MBA lineals com a combinacions lineals respecte a un conjunt arbitrari d'operadors *mínims* i expressions que conformen una base d'un espai vectorial *ad hoc* on habiten aquestes expressions MBA lineals.

Analitzem les contribucions, defectes i limitacions de les investigacions recents en aquest sentit, i en proporcionem aplicacions pràctiques. En particular, aprofitem una transformació que redueix l'alternança d'operadors aritmètics i bit a bit d'una expressió MBA tot preservant-ne el comportament semàntic en el context de la desofuscació de codi basada en la síntesi de programes. Aquesta transformació s'utilitza després per ajudar en el problema de verificar la correcció semàntica d'una expressió candidata sintetitzada, millorant així la solidesa d'aquesta tècnica.

Resumen

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

En este proyecto, exploramos enfoques ortogonales a las técnicas actuales para tratar la simplificación de expresiones MBA, que se basan en el uso combinado de ejecución simbólica y síntesis de programas. La idea principal es conseguir extraer *información* subyacente a *la estructura matemática* de estas expresiones. Así, podemos utilizar esta información, sea por sí misma o en combinación con otras técnicas, para facilitar la tarea de simplificar expresiones MBA. Ponemos énfasis en el estudio de algunos artículos académicos recientes que desarrollan estas ideas, mayoritariamente dirigidos a proporcionar una representación *normalizada* (de un subconjunto) de expresiones MBA lineales como combinaciones lineales respecto a un conjunto arbitrario de operadores *mínimos* y expresiones que conforman una base de un espacio vectorial *ad hoc* donde habitan estas expresiones MBA lineales.

Analizamos las contribuciones, defectos y limitaciones de las investigaciones recientes en este sentido, y proporcionamos aplicaciones prácticas. En particular, aprovechamos una transformación que reduce la alternancia de operadores aritméticos y bit a bit de una expresión MBA preservando su comportamiento semántico en el contexto de la desofuscación de código basada en la síntesis de programas. Esta transformación se utiliza luego para ayudar al problema de verificar la corrección semántica de una expresión candidata sintetizada, mejorando así la solidez de esta técnica.

Contents

Introduction **1**

1 Mixed Boolean-Arithmetic expressions **3**
1.1 Fundamentals . . . 3
1.1.1 Polynomial MBA expressions . . . 3
1.1.2 Linear MBA expressions . . . 3
1.2 MBA expressions in the context of code obfuscation . . . 4
1.2.1 Obfuscation of expressions . . . 4
1.2.2 Obfuscation of constants . . . 6
1.3 Code deobfuscation through MBA simplification . . . 8
1.3.1 Symbolic execution . . . 8
1.3.2 Program synthesis . . . 10

2 Analysis of recent orthogonal approaches to MBA simplification **13**
2.1 Motivation . . . 13
2.1.1 Generating new linear MBA equalities . . . 13
2.2 MBA-Blast . . . 17
2.2.1 Contributions . . . 17
2.2.2 Flaws and limitations . . . 19
2.3 MBA-Solver . . . 21
2.3.1 Contributions . . . 21
2.3.2 Flaws and limitations . . . 26

3 Improving program synthesis based MBA simplification reliability **29**
3.1 Motivation . . . 29
3.2 Method description . . . 29
3.3 Practical application: a guided example . . . 32
3.4 Limitations . . . 39

Conclusions **41**

Bibliography **43**

Appendices **47**
A Proposed objectives . . . 47
A.1 Principal . . . 47
A.2 Secondary . . . 47
A.3 Specifics . . . 47
B Logistics . . . 48

B.1 Temporal planning 48
B.2 Report 48
B.3 Contact with supervisor 48

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 its 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 recent progress on orthogonal approaches to aid in deobfuscation efforts, there is still a huge lack of both theoretical foundations and practical tools to address the question of simplifying/deobfuscating Mixed Boolean-Arithmetic expressions.

Chapter 1

Mixed Boolean-Arithmetic expressions

1.1 Fundamentals

Informally, a Mixed Boolean-Arithmetic (MBA) expression is an *algebraic expression* composed of integer arithmetic operators, e.g. $(+, -, \times)$ and bitwise operators, e.g. $(\wedge, \vee, \oplus, \neg)$. The concept of MBA expressions initially appeared in the context of code obfuscation, in the work by Zhou et al. [Zho+07].

As in Eyrolles' PhD thesis [Eyr17], we choose to define MBA expressions by **explicitly describing the different building blocks** (operators) that compose them and how they are bundled together.

Remark 1.1. We will use interchangeably the terms of *boolean* and *bitwise* operators.

1.1.1 Polynomial MBA expressions

A polynomial MBA expression consists of a sum of terms, each one composed of an n-bit constant a_i times the product of several bitwise expressions on a number t of n-bit variables.

Definition 1.2 (Polynomial MBA expression)**.** *An expression E of the form*

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

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.

Example 1.3. The expression E written as

$$E = 43(x \wedge y \vee z)^2((x \oplus y) \wedge z \vee t) + 2x + 123(x \vee y)zt^2$$

is a polynomial MBA expression.

1.1.2 Linear MBA expressions

Linear MBA expressions are defined as a restriction to the previous definition, by imposing **just one bitwise expression for each term** instead of a product of an arbitrary number of them.

Definition 1.4 (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 1.5. The expression E written as

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

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

1.2 MBA expressions in the context of code obfuscation

The following statement from [Zho+07] is essential to our study.

Proposition 1.6. *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, regardless of the particular iterative composition techniques used to build (obfuscate, complicate) them.

1.2.1 Obfuscation of expressions

Given an MBA expression E_1, we are interested in generating a **semantically equivalent** expression E_2 which is **syntactically more complex** than the initial expression E_1.

This technique was presented in [Zho+07; ZM06] and in several patents with intersecting authors, like [JXY08]. The process relies on two differentiated components, which can be used either alone or combined.

MBA rewriting

A chosen operator (subexpression) is rewritten with an equivalent MBA expression.

Example 1.7.

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

A list of rewrite rules is given in Appendix A of [Eyr17]. Other MBA equalities can be found in [War12] referred as *bit hacks*. A constructive method to generate arbitrary linear rewrite rules is also known, and will be presented in Section 2.1.1.

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 $\mathbb{Z}/2^n\mathbb{Z}$.

Remark 1.8. Both in literature and *in-the-wild*, these functions are often affine mappings, following the construction that was presented in [Zho+07]. However, there is no actual reason why they cannot be any other pair of inverse functions that define a bijection (i.e a *1-to-1* map).

Example 1.9. 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_2 obtained by applying the previous rewrite rule to E_1:

$$E_2 = (x \oplus y) + 2 \times (x \wedge y)$$

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

$$E_{tmp} = f(E_2) = 39 \times E_2 + 23$$
$$E_3 = f^{-1}(E_{tmp}) = 151 \times E_{tmp} + 111$$

Finally, expand E_3 to observe the final obfuscated expression derived from $E_1 = x + y$:

$$E_3 = 151 \times (39 \times ((x \oplus y) + 2 \times (x \wedge y)) + 23) + 111$$

Let's look at an implementation of this construction with the help of an SMT solver[1]. In particular, we leverage *Z3* [MB08] through its *Python API* to check that the functions f and f^{-1} define inverse mappings and prove the semantic equivalence of $E1, E2$ and $E3$.

```
from z3 import *

x, y = BitVecs('x y', 8)

def f(x): return 39*x + 23
def f_inv(x): return 151*x + 111

print("Prove that f and f_inv are inverses:\n---")
prove(f(f_inv(x)) == x)

E1 = x + y
E2 = (x ^ y) + 2*(x & y)
E_tmp = f(E2)
E3 = f_inv(E_tmp)

print("\nObserve the three expressions E1, E2 and E3:\n---")
print("E1 =", E1)
print("E2 =", E2)
print("E3 =", E3)

print("\nProve semantic equivalence of E1, E2 and E3:\n---")
prove(E1 == E2)
prove(E2 == E3)
prove(E3 == E1)
```

If we run it, we obtain:

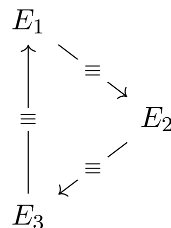
[1] https://en.wikipedia.org/wiki/Satisfiability_modulo_theories

```
Prove that f and f_inv are inverses:
---
proved

Observe the three expressions E1, E2 and E3:
---
E1 = x + y
E2 = (x ^ y) + 2*(x & y)
E3 = 151*(39*((x ^ y) + 2*(x & y)) + 23) + 111

Prove semantic equivalence of E1, E2 and E3:
---
proved
proved
proved
```

Remark 1.10. To prove the semantic equivalence of $E1, E2$ and $E3$, we proved the following diagram of equivalences.



1.2.2 Obfuscation of constants

This technique known as *opaque constant* allows to hide a target constant K (e.g. a secret key used in a decryption routine, known constants for hashing algorithms, etc.). It combines the power of MBA expressions with *permutation polynomials*[2]

A permutation polynomial is a polynomial that acts as a permutation of the elements of the set they apply to (in our case, n-bit values), i.e. defines a *1-to-1* map (bijection) in $\mathbb{Z}/2^n\mathbb{Z}$. As such polynomials define a bijection, an inverse mapping must exist. However, such inverse is not guaranteed to exist in the form of another permutation polynomial (of same degree).

Permutation polynomials were characterized by Rivest in [Riv01], but an inversion algorithm was not provided then. Later, Zhou et al. provided a constructive method to generate a subset of such polynomials of degree m, denoted as $P_m(\mathbb{Z}/2^n\mathbb{Z})$, as well as a formula to find their inverse. This formula and proof of correctness can be consulted on Section 3.3 of [Zho+07].

The method to construct the opaque constant is given by the following proposition.

Proposition 1.11. *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.*

[2] https://en.wikipedia.org/wiki/Permutation_polynomial

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. □

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 value K to perform any computation.

Example 1.12. Working on 8-bit values, let:

$$\begin{aligned} K &= 123 \\ P(X) &= 97X + 248X^2 \\ Q(X) &= 161X + 136X^2 \\ E(x, y) &= x - y + 2(\neg x \wedge y) - (x \oplus y) \end{aligned}$$

where K is the constant value 123 in $\mathbb{Z}/2^8\mathbb{Z}$ that we want to hide, P and Q are a pair of inverse permutation polynomials in 8-bits, i.e. in $\mathbb{Z}/2^8\mathbb{Z}$ and $E(x, y)$ is a non-trivially equal to zero MBA expression on two 8-bit variables x and y.

The opaque constant function $OC(x, y) = P(E(x, y) + Q(K))$ will be given by the expression

$$\begin{aligned} &195 \\ &+ 97x \\ &+ 159y \\ &+ 194\neg(x \vee \neg y) \\ &+ 159(x \oplus y) \\ &+ (163 + x + 255y + 2\neg(x \vee \neg y) + 255(x \oplus y))(232 + 248x + 8y + 240\neg(x \vee \neg y) + 8(x \oplus y)) \end{aligned}$$

That is, for any pair of 8-bit input values x and y, the function $OC(x, y)$ will always produce the constant output value of $K = 123$.

Again, let's look at an implementation of this example in *Python* using *Z3*.

```
from z3 import *

# Define constant K = 123 as an 8-bit value
K = BitVecVal(123, 8)

# Define inverse permutation polynomials
X = BitVec('X', 8)
def P(X): return 97*X + 248*X*X
def Q(X): return 161*X + 136*X*X

print("Prove that P and Q define inverse mappings:\n---")
prove(P(Q(X)) == X)

# Define non-trivially equal to zero MBA expression
```

```
x, y = BitVecs('x y', 8)
def E(x, y): return x-y + 2*(~x&y) - (x^y)

print("\nProve that MBA expression is non-trivially equal to zero:\n---")
prove(E(x, y) == 0)

# Generate opaque constant
OC = P(E(x,y) + Q(K))

# Apply basic simplification rules (group terms)
OC = simplify(OC)

# Print opaque constant function
print("\nOpaque constant generated:\n---")
print(OC)

print("\nProve that OC function is semantically equivalent to the constant 123\n---")
prove(OC == 123)
```

If we run it, we obtain:

```
Prove that P and Q define inverse mappings:
---
proved

Prove that MBA expression is non-trivially equal to zero:
---
proved

Opaque constant generated:
---
195 +
97*x +
159*y +
194*~(x | ~y) +
159*(x ^ y) +
(163 + x + 255*y + 2*~(x | ~y) + 255*(x ^ y))*
(232 + 248*x + 8*y + 240*~(x | ~y) + 8*(x ^ y))

Prove that OC function is semantically equivalent to the constant 123
---
proved
```

1.3 Code deobfuscation through MBA simplification

1.3.1 Symbolic execution

Symbolic execution is a technique that lets the analyst transform the control-flow and data-flow of the program into symbolic expressions.

Essentially, it can be thought as a *computer algebra system* for a programming language, an assembly representation of a given computer architecture or even intermediate languages (IL), also known as intermediate representations (IR).

Symbolic execution can be leveraged to extract the formula for the value a variable will hold at some point in the program, with respect to the inputs defined at a starting point of the analysis (basic block, function, etc.). Similarly, it can also be used to extract the path constraints that encode the branching conditions of a basic block with respect to the variables involved in it.

The basic operation of a symbolic execution engine is as follows:

- Define an initial state mapping for the variables (registers, memory)
- Translate assembly into formulas (possibly using an IR)
- *Execute* current instruction and extract its *semantics*
- Update the state mapping to account for the effects of the *executed* instruction

Plugging an SMT solver

One of the main strengths of symbolic execution is when it gets combined with an SMT solver [JGY15; PRV11].

SMT solvers can find *satisfying* solutions to a set of constraints. Three possible outcomes exist:

- *SAT*: A concrete assignment exists satisfying the constraints
- *UNSAT*: There is no solution for the given set of constraints
- *UNK*: Answer not found within the resource boundaries (usually timeout)

When the outcome is *SAT*, a concrete assignment known as a *model* is also retrieved.

Combining SMT solvers with symbolic execution comes very naturally:

- In control-flow analysis, the symbolic execution engine is used to extract the formulae (constraints) for a given path branching condition. The constraints are fed into an SMT solver that can prove the feasibility of such paths.
- In data-flow analysis, the symbolic execution engine is usually leveraged to extract the formula for the return value of a function with respect to its input parameters. Then, this formula can be fed to the SMT solver to craft an input that makes the function return the desired value (bypass checks, etc.) or treated as an obfuscated (MBA) expression that we aim to simplify.

These techniques have been shown to work nicely when addressing the issue of control-flow based obfuscation in which we need to check the satisfiability of the obtained symbolic constraints [SBP18]. However, complete automation might not scale on large and complex targets due to path explosion, as the symbolic state needs to be forked at every branching control, which can lead to resource exhaustion.

The results are not so convincing when addressing the task of data-flow deobfuscation (for example, MBA simplification or VM instruction handlers behavior extraction from VM-based obfuscation) where 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 *classic* 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 obfuscation transformations) [Ban+16].

1.3.2 Program synthesis

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 its 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 [BA06; Gul+11; GT11; Rol14; Bio+17; Bla+17; DCC20; Men+21]. 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.

Example 1.13. Consider the following function describing an obfuscated MBA expression:

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

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

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

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

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

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 out by the *synthesizer*.

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[3]). 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 P described before, we could provide as the specification a list of input/output values like:

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

- 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.

We want to recover (learn) the semantics of some obfuscated code (expression) whose syntactic complexity has been arbitrarily increased to the point where an SMT solver is not *enough* to adequately simplify the obfuscated code (expression) into a *simple enough* representation of its semantics.

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

While there are many *flavors* of program synthesis [Gul10; GPS17], the nature of our problem leads to an **inductive oracle-guided program synthesis** style, using the obfuscated code as an I/O oracle:

- Generate a set of I/O pairs from the obfuscated code (oracle).
- Determine the best candidate program that matches the I/O behavior.

Notable existing work

Syntia (2017) [Bla+17]: Monte Carlo Tree Search (MCTS) based stochastic program synthesis:

- Convert the problem of finding a candidate program into a stochastic optimization problem
- At each iteration we generate intermediate results instead of actual candidate programs
- Evolve towards a global optima (best candidate program) guided by a cost function.

A public implementation is available[4] as well as an integration into the radare2[5] reverse engineering framework, called r2syntia[6] which was presented in [Mon20].

QSynth (2020) [DCC20]: Offline enumerative program synthesis:

- Given a context-free grammar, generate *all* programs up to a certain number of derivations
- Create *offline* lookup tables mapping each candidate program to its I/O behavior
- Perform an exhaustive search for candidate programs matching the oracle's I/O behavior

The most significant contribution of the QSynth approach is the ability to split an obfuscated expression into smaller subexpressions, synthesize them individually and then reconstruct the total simplified expression. Public implementations exist, namely msynth[7] on top of Miasm[8], and QSynthesis[9] on top of Triton[10] [SS15].

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.

[4] `https://github.com/RUB-SysSec/syntia`
[5] `https://github.com/radareorg/radare2`
[6] `https://github.com/arnaugamez/r2syntia`
[7] `https://github.com/mrphrazer/msynth`
[8] `https://miasm.re/`
[9] `https://github.com/quarkslab/qsynthesis`
[10] `https://triton.quarkslab.com/`

Chapter 2

Analysis of recent orthogonal approaches to MBA simplification

2.1 Motivation

As we have discussed, symbolic execution has a hard time dealing with MBA expressions, as SMT solvers are easily thwarted by arbitrarily increasing the syntactic complexity of such expressions through MBA rewrite rules and insertion of identities [Ban+16]. While program synthesis addresses this problematic by only considering these expressions from a semantic point of view, existing approaches still have scaling limitations when the encoded semantics of the obfuscated expression are not *simple enough*, and usually they cannot guarantee the correctness of a simplified candidate, i.e. its semantic equivalence to the original obfuscated expression.

These limitations motivate the search for orthogonal approaches to MBA simplification that take into account underlying mathematical properties. Thus, we want to extract MBA invariants and other relevant information that can help in our understanding of MBA expressions *by themselves* and derive simplification strategies that could be leveraged alone, or even plugged into the existing techniques discussed for a better overall outcome.

The main contributions from both of the recent articles analyzed in this chapter are built on top of the following method to construct non-trivially equal to zero linear MBA expressions, from which new linear MBA equalities can be easily generated.

2.1.1 Generating new linear MBA equalities

Theorem 2.1 ([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,*

- $e_j(X_1,\dots,X_t) = \begin{pmatrix} f_j(X_{1,0},\dots,X_{t,0}) \\ \vdots \\ f_j(X_{1,n-1},\dots,X_{t,n-1}) \end{pmatrix}$ *with* $X_{k,i}$ *the i-th bit of* X_k *and*

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

- $F = \begin{pmatrix} f_0(0) & \dots & f_{s-1}(0) \\ \vdots & & \vdots \\ f_0(2^t-1) & \dots & f_{s-1}(2^t-1) \end{pmatrix}$ *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. We can construct non-trivially equal to zero linear MBA expressions and derive rewrite rules from these by *moving terms* at different sides of the equality. The method is based on the following corollary.

Corollary 2.2. *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 2.3. 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, a non-trivial solution to the system of equations derived from $F \cdot V = 0$ is:

$$V = \begin{pmatrix} -1 \\ 1 \\ 1 \\ 2 \\ -2 \end{pmatrix}$$

Meaning that:

$$-f_0 + f_1 + f_2 + 2f_3 - 2f_4 = 0$$

by replacing each f_i by the bitwise expression it represents.

This yields the following linear MBA equation:

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

which is a non-trivially equal to zero MBA expression.

Finally, we can derive many equalities from this equation leading to MBA rewrite rules:

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

Let's look at an implementation of this construction in *Python*. We leverage *NumPy*[1] for matrix manipulations and *Z3* to solve the system of equations.

[1] `https://numpy.org/`

```
import numpy as np
from z3 import *

# Truth table vectors
tt_x          = np.array([0,0,1,1])
tt_y          = np.array([0,1,0,1])
tt_x_xor_y    = np.array([0,1,1,0])
tt_x_or_not_y = np.array([1,0,1,1])
tt_neg_1      = np.array([1,1,1,1])

# Convert into matrix from column vectors
F = np.column_stack((tt_x, tt_y, tt_x_xor_y, tt_x_or_not_y, tt_neg_1))

# Define incognites for solution
x1, x2, x3, x4, x5 = Ints('x1 x2 x3 x4 x5')
V = [x1, x2, x3, x4, x5]

# Initialize the Z3 solver engine
solver = Solver()

# Add the conditions for the system of equations
print ("System of equations:\n---")
for row in range(F.shape[0]):
    eq = F[row][0]*V[0]
    for col in range(1, F.shape[1]):
        eq += F[row][col]*V[col]

    print(f"{eq} = 0")
    solver.add(eq == 0)

# We don't want the trivial solution
solver.add(
    Not(
        And(
            x1 == 0, x2 == 0, x3 == 0, x4 == 0, x5 == 0
        )
    )
)

# Check and extract the model (i.e. a particular solution)
solver.check()
m = solver.model()

# Sort and print the output of the model nicely
ordered = (sorted (m.decls(), key = lambda x: str(x)))
print("\nCandidate solution:\n---\nV =", ordered, '=', [m[i] for i in ordered])

# Define x and y as BitVectors of 8 bits (you can change the bitsize)
x, y = BitVecs('x y', 8)

# We prove that the derived MBA expression is non-trivially equal to zero
print("\nCheck that MBA expression is non-trivially equal to zero:\n---")
prove(-x + y + (x^y) + 2*(x | (~y)) + 2 == 0)
```

If we run this snippet of code, we obtain:

```
System of equations:
---
0*x1 + 0*x2 + 0*x3 + 1*x4 + 1*x5 = 0
0*x1 + 1*x2 + 1*x3 + 0*x4 + 1*x5 = 0
1*x1 + 0*x2 + 1*x3 + 1*x4 + 1*x5 = 0
1*x1 + 1*x2 + 0*x3 + 1*x4 + 1*x5 = 0

Candidate solution:
---
V = [x1, x2, x3, x4, x5] = [-1, 1, 1, 2, -2]

Check that MBA expression is non-trivially equal to zero:
---
proved
```

2.2 MBA-Blast

2.2.1 Contributions

The core contribution that Binbin Liu et al. present in [Liu+21] is a proof for the converse direction of the previous theorem. They state and prove it as follows:

Theorem 2.4 ([Liu+21]). *Let $E = \sum_{j=0}^{s-1} a_j e_j$ be an MBA expression, where a_j are integers and e_j are boolean functions $f_j(X_1, X_2, \ldots, X_t)$ taking t variables $X_1, X_2, \ldots, X_t$ as input. Each variable has n bits. We use $X_{k,i}$ to represent the ith bit of the kth input variable in e_j. Let M be the $2^t \times s$ boolean matrix representing the truth table of $e_0, e_1, \ldots, e_{s-1}$.*

$$\vec{v} = \begin{pmatrix} a_0 \\ a_1 \\ \ldots \\ a_{s-1} \end{pmatrix}$$

is an s dimension vector consisting of all the coefficients in E.

Then, $E \equiv 0$ if and only if the linear system $M\vec{v} = 0$.

Proof. The sufficiency has already been proved in Theorem 2.1, that is, if $M\vec{v} = 0$, then $E \equiv 0$. Now we prove the necessity, namely, if $E \equiv 0$, then $M\vec{v} = 0$.

If $E \equiv 0$, then

$$E = 2^0 \cdot E_0 + 2^1 \cdot E_1 + \cdots + 2^{n-1} \cdot E_{n-1} = 0$$

where E_i is the calculation of E on the ith bit of input variables:

$$E_i = \sum_{j=0}^{s-1} a_j f_j(X_{1,i}, \ldots, X_{t,i})$$

We prove $E_i = 0$ by contradiction.

Suppose $\exists k, E_k = \sum_{j=0}^{s-1} a_j f_j(X_{1,k}, \ldots, X_{t,k}) = \bar{e} \neq 0$. We construct a group of inputs $X'_1, X'_2, \ldots, X'_t$ where

$$\begin{aligned} X'_{1,i} &= X_{1,k} \\ X'_{2,i} &= X_{2,k} \\ &\cdots \\ X'_{t,i} &= X_{t,k} \end{aligned}$$

for $i = 1, 2, \ldots, n$

Feed $X'_1, X'_2, \ldots, X'_t$ to E, then $\forall i = 1, 2, \ldots, n$, $E_i = \bar{e}$

$$\begin{aligned} E &= 2^0 \cdot E_0 + 2^1 \cdot E_1 + \cdots + 2^{n-1} \cdot E_{n-1} \\ &= 2^0 \cdot \bar{e} + 2^1 \cdot \bar{e} + \cdots + 2^{n-1} \cdot \bar{e} \\ &= (2^n = 1)\bar{e} \end{aligned}$$

Since $E \equiv 0$,

$$\begin{aligned} (2^n = 1)\bar{e} = 0 \\ \bar{e} = 0 \end{aligned}$$

This contradicts the supposition that $\bar{e} \neq 0$. Hence, our supposition is false, so for any input $X_{1,i}, \ldots, X_{t,i}$,

$$\begin{aligned} E_i = \sum_{j=0}^{s-1} a_j f_j(X_{1,i}, \ldots, X_{t,i}) = 0 \\ a_0 e_0 + a_1 e_1 + \cdots + a_{s-1} e_{s-1} = 0 \end{aligned}$$

Therefore,

$$M\vec{v} = 0$$

□

Thanks to this proof, we have now the ability to reduce n-bit MBA expressions into 1-bit MBA expressions and back. Given an n-bit obfuscated MBA expression E_n, the goal is to find a *simpler* and semantically equivalent n-bit expression E'_n. Within this context, they propose a method to address the problem of MBA simplification as follows:

1. Transform E_n in n-bit space to E_1 in 1-bit space.
2. Find a simplified MBA expression E'_1 in 1-bit space, such that $E_1 - E'_1 \equiv 0$
3. Transform E'_1 in 1-bit space to E'_n in n-bit space.

The authors of the paper state that steps 1 and 3 are given by the previous proof. For the step 2, they perform a truth table based (bruteforce) simplification strategy on this 1-bit space.

We will not get into the details of such method, which is rather simple and can be consulted on the original paper [Liu+21]. Nevertheless, it is important to point out several flawed statements and issues with this approach that do not seem to have been contemplated, but assumed to work *universally* instead.

2.2.2 Flaws and limitations

MBA definition

The first thing to note is that the definition that the paper provides for an MBA expression is actually the definition of a linear MBA expression, excluding the more general definition for a polynomial MBA expression.

Thus, from the very beginning we must know that the paper will only deal with a subset of MBA expressions: linear MBA expressions. This fact is not mentioned anywhere.

Obfuscation through insertion of identities is not considered

Their approach has been formulated in a way that targets obfuscated MBA expressions that have been constructed through MBA rewrite rules. Thus, obfuscated MBA expressions that leverage insertion of identities (encodings) might thwart this methodology, as they explicitly state [Liu+21]:

> It is also possible that attackers combine MBA obfuscation with other data encoding techniques to create complex expressions with bitwise and arithmetic operations but does not meet the MBA definition in this paper.

The method is not general, even for linear MBA expressions

The authors state [Liu+21]:

> The two-way feature in current MBA obfuscation implies that *any* n-bit obfuscated MBA expression can be simplified in 1-bit space. Consequently, the MBA reduction in 1-bit space is equivalent to that in n-bit space.

This affirmation is not correct, or at least not entirely precise. On the one hand, it would not apply to polynomial (non-linear) MBA expressions. This is trivial, as the theorem (and proof) that brings this results is applied only to linear MBA expressions.

However, the statement is flawed even if we restrict the definition of MBA expressions to consider only linear MBA expressions.

In [Eyr17], Eyrolles already points out that for the motivating Theorem 2.1 to hold, the bitwise operations cannot contain constants other than 0 and -1. The reason for that is pretty simple: the only way to map an n-bit constant to a 1-bit constant (and back) without losing information is if the n-bit constant is either 0 ($0 \dots 0$ in binary) or -1 ($1 \dots 1$ in binary).

We can also find linear MBA expressions without problematic constants that do not hold their formulation [Liu+21]:

> Any n-bit MBA identity is equivalent to the same form on 1-bit space.
>
> $$E_n - E'_n \equiv 0 \Leftrightarrow E_1 - E'_1 \equiv 0$$

Example 2.5. Let

$E_1 = x + y, E'_1 = x \oplus y, E_1 - E'_1 = x + y - (x \oplus y)$ in 1 bit.

$E_2 = x + y, E'_2 = x \oplus y, E_2 - E'_2 = x + y - (x \oplus y)$ in 2 bits.

In this case, we have

$$E_1 - E'_1 \equiv 0, E_2 - E'_2 \not\equiv 0$$

Let's prove it with the help of *Z3*.

```
from z3 import *

x, y = BitVecs("x y", 1)
E1 = x + y
E2 = x ^ y

print("Prove that (x + y) - (x ^ y) is equivalent to 0 in 1-bit space:\n---")
prove(E1 == E2)

x, y = BitVecs("x y", 2)
E1 = x + y
E2 = x ^ y

print("\nProve that (x + y) - (x ^ y) is NOT equivalent to 0 in 2-bit space:\n---")
prove(E1 == E2)
```

If we run this snippet of code, we obtain:

```
Prove that (x + y) - (x ^ y) is equivalent to 0 in 1-bit space:
---
proved

Prove that (x + y) - (x ^ y) is NOT equivalent to 0 in 2-bit space:
---
counterexample
[y = 3, x = 1]
```

The reason why this example (and many others) fails might not be obvious. It comes from the actual limitations of the underlying theorem itself. As we have seen, the method to construct the linear MBA equalities which justifies the MBA-Blast approach requires that the column vectors obtained from the truth table of the bitwise expressions form a linearly dependent set of vectors. But, in this case, we have

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

with column vectors truth tables for:

$$f_0(x, y) = x$$
$$f_1(x, y) = y$$
$$f_2(x, y) = (x \oplus y)$$

For the method to work, their premises must hold. This translates to the fact that the rank of the matrix F must be strictly less than the number of columns.

In terms of linear algebra justification, we are essentially computing solutions for homogeneous systems of linear equations and are interested in obtaining non-trivial solutions. This is only

possible if the rank of the matrix (bounded by the number of rows, being 2^t for t the number of variables) is strictly less than the total number of unknowns of the solution (which is equal to the number of columns i.e. the number of bitwise expressions of the underlying linear MBA expression). In this scenario, we have a compatible indeterminate system which has other solutions than the trivial (indeed, the system will have an entire *family* of infinite solutions that hold some given relation on the unknowns).

In this example, the rank of the matrix is 3, which is equal to the number of columns. Thus, the homogeneous system defined will have a single solution (the trivial one), which means that column vectors are not linearly dependent. Therefore, the theorem premise does not hold.

In conclusion, for the MBA-Blast approach to work we need to impose several restrictions to the MBA expressions:

- They must be linear MBA expressions
- Bitwise expressions cannot contain constants other than 0 or -1.
- The truth tables for the bitwise expressions must generate a set of linearly dependent vectors.

Moreover, the method might not work successfully if obfuscation through insertion of identities is in place, as stated before.

2.3 MBA-Solver

2.3.1 Contributions

The research by Dongpeng Xu et al. [Xu+21] presents a semantics preserving transformation method for reducing an MBA expression into a more *digestible* form to be consumed by SMT solvers when checking for semantic equivalence against other expressions.

First, they analyze how several MBA complexity metrics affect the solving performance. The complexity metrics considered are the following:

- MBA Type: linear, polynomial (non-linear) and non-polynomial.
- Number of Variables.
- MBA Alternation: the number of operations that connect arithmetic and bitwise operations.
- MBA Length: considering the MBA expression as a string.
- Number of Terms.
- Coefficient: how large the coefficients are in every term.

A deeper study and formalization of several MBA complexity metrics can be found in [Eyr17].

They conclude that MBA alternation is the dominant factor influencing SMT solver's performance. Hence, they specifically construct their semantics preserving transformation to reduce MBA alternation. The underlying idea is that by reducing the MBA alternation of a given *complex* MBA expression, SMT solvers with have a higher chance to solve them.

Next, we discuss the core components and approach proposed.

Signature vector

Definition 2.6. *The signature vector $\vec{s}$ of a linear MBA expression is the product of the MBA truth table matrix M and the coefficients vector $\vec{v}$.*

The linear MBA truth table matrix refers to the same matrix construction presented in the motivating Theorem 2.1.

Example 2.7. [Xu+21] For the linear MBA expression

$$E = 2(x \vee y) - (\neg x \wedge y) - (x \wedge \neg y)$$

we have the following M and $\vec{v}$

$$M = \begin{pmatrix} x \vee y & \neg x \wedge y & x \wedge \neg y \\ 0 & 0 & 0 \\ 1 & 1 & 0 \\ 1 & 0 & 1 \\ 1 & 0 & 0 \end{pmatrix}, \; \vec{v} = \begin{pmatrix} 2 \\ -1 \\ -1 \end{pmatrix}$$

Thus, the signature vector $\vec{s}$ is

$$\vec{s} = M\vec{v} = \begin{pmatrix} 0 \\ 1 \\ 1 \\ 2 \end{pmatrix}$$

Groups of equivalent linear MBA expressions

The signature vector provides an invariant property for linear MBA expressions with the same semantic behavior. In other words, the signature vector encodes the semantics of a group of equivalent linear MBA expressions.

Theorem 2.8. *Given two linear MBA expressions E_1 and E_2 and their respective signature vectors $\vec{s_1}$ and $\vec{s_2}$, E_1 and E_2 are semantically equivalent if and only if $\vec{s_1}$ and $\vec{s_2}$ are equal, i.e.*

$$E_1 \equiv E_2 \Leftrightarrow \vec{s_1} = \vec{s_2}$$

Proof.

$$\vec{s_1} = \vec{s_2} \Leftrightarrow M_1\vec{v_1} = M_2\vec{v_2} \Leftrightarrow M_1\vec{v_1} - M_2\vec{v_2} = 0$$

Which can be written as

$$[M_1 \; M_2] \begin{pmatrix} \vec{v_1} \\ -\vec{v_2} \end{pmatrix} = 0 \Leftrightarrow E_1 - E_2 \equiv 0 \Leftrightarrow E_1 \equiv E_2$$

□

Linear MBA expressions from signature vectors

From linear algebra, we know that any vector can be represented as a linear combination of a set of base vectors. In our particular scenario, such base vectors can be thought as truth table representations of a given bitwise expression on a certain number of variables of the linear MBA expression.

Example 2.9. For two variables x, y we could construct the following *canonical* base by truth table representation:

x	y	$\neg x \wedge \neg y$	$\neg x \wedge y$	$x \wedge \neg y$	$x \wedge y$
0	0	1	0	0	0
0	1	0	1	0	0
1	0	0	0	1	0
1	1	0	0	0	1

Using this base, the previous signature vector

$$\vec{s} = \begin{pmatrix} 0 \\ 1 \\ 1 \\ 2 \end{pmatrix}$$

of the linear MBA expression

$$E = 2(x \vee y) - (\neg x \wedge y) - (x \wedge \neg y)$$

could be expressed as follows:

$$\begin{pmatrix} 0 \\ 1 \\ 1 \\ 2 \end{pmatrix} = 0 \cdot \begin{pmatrix} 1 \\ 0 \\ 0 \\ 0 \end{pmatrix} + 1 \cdot \begin{pmatrix} 0 \\ 1 \\ 0 \\ 0 \end{pmatrix} + 1 \cdot \begin{pmatrix} 0 \\ 0 \\ 1 \\ 0 \end{pmatrix} + 2 \cdot \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix}$$

which corresponds to the following linear MBA expression:

$$E' = (\neg x \wedge y) + (x \wedge \neg y) + 2(x \wedge y)$$

By Theorem 2.8, we have that E and E' are semantically equivalent.

Choosing the right vector base

We are interested in minimizing the MBA alternation within a given expression. Thus, we will choose a set of base vectors that minimizes the number of bitwise expressions.

In particular, for 2 variables, the following set of *normalized* base vectors is chosen:

x	y	$x \wedge y$	-1
0	0	0	1
0	1	0	1
1	0	0	1
1	1	1	1

These vectors consist of the truth table representations of the variables x, y alone, the constant -1 and the only bitwise expression $x \wedge y$.

Example 2.10. To represent the previous signature vector

$$\vec{s} = \begin{pmatrix} 0 \\ 1 \\ 1 \\ 2 \end{pmatrix}$$

in this *normalized* base, we need to solve the following system of linear equations:

$$\begin{pmatrix} 0 \\ 1 \\ 1 \\ 2 \end{pmatrix} = C_1 \cdot \begin{pmatrix} 0 \\ 0 \\ 1 \\ 1 \end{pmatrix} + C_2 \cdot \begin{pmatrix} 0 \\ 1 \\ 0 \\ 1 \end{pmatrix} + C_3 \cdot \begin{pmatrix} 0 \\ 0 \\ 0 \\ 1 \end{pmatrix} + C_4 \cdot \begin{pmatrix} 1 \\ 1 \\ 1 \\ 1 \end{pmatrix}$$

which produces the following result:

$$C_1 = 1$$
$$C_2 = 1$$
$$C_3 = 0$$
$$C_4 = 0$$

Therefore,

$$\begin{pmatrix} 0 \\ 1 \\ 1 \\ 2 \end{pmatrix} = \begin{pmatrix} 0 \\ 0 \\ 1 \\ 1 \end{pmatrix} + \begin{pmatrix} 0 \\ 1 \\ 0 \\ 1 \end{pmatrix}$$

which corresponds to the following linear MBA expression:

$$E'' = x + y$$

By Theorem 2.8, we have that E, E' and E'' are semantically equivalent.

Dealing with non-linear MBA expressions

To address non-linear (either polynomial or non-polynomial) MBA expressions, the basic idea is to simplify the linear subexpressions that conform them, down to the plain bitwise expressions on some of the variables.

For any bitwise expression, its signature vector will be computed. This signature vector will always be composed of 1's and 0's by the truth table nature of this construction for *plain* bitwise expressions. Thus, an exhaustive simplification mapping table for all possible truth table values of signature vectors for bitwise expressions is pre-computed, expressing each signature vector in terms of the *normalized* base chosen to reduce MBA alternation. The signature vector for the bitwise expression will be looked up in the mapping table and substituted by the equivalent *normalized* expression.

After that, the resulting MBA expression will showcase a minimal MBA alternation complexity, allowing for further simplifications to merge terms and produce a more concise form.

Using the previous *normalized* base for 2 variables, we have the following simplification table mapping signature vectors ($\vec{s}$) to MBA expressions (E) of *plain* bitwise expressions:

$\vec{s}$	E
$(0,0,1,1)$	x
$(0,1,0,1)$	y
$(0,0,0,1)$	$x \wedge y$
$(1,1,1,1)$	-1
$(0,0,0,0)$	0
$(0,0,1,0)$	$x - (x \wedge y)$
$(0,1,0,0)$	$y - (x \wedge y)$
$(0,1,1,0)$	$x + y - 2(x \wedge y)$
$(0,1,1,1)$	$x + y - (x \wedge y)$
$(1,0,0,0)$	$-x - y + (x \wedge y) - 1$
$(1,0,0,1)$	$-x - y + 2(x \wedge y) - 1$
$(1,0,1,0)$	$-y - 1$
$(1,0,1,1)$	$-y + (x \wedge y) - 1$
$(1,1,0,0)$	$-x - 1$
$(1,1,0,1)$	$-x + (x \wedge y) - 1$
$(1,1,1,0)$	$-(x \wedge y) - 1$

Even if substituting bitwise expressions with their representation in the *normalized* base seems to increase the length of the expression, as all of them are now expressed within the same terms, the possibilities for merging and canceling terms appear naturally.

Example 2.11. Consider the following non-linear MBA expression:

$$(x \wedge \neg y) \cdot (\neg x \wedge y) + (x \wedge y) \cdot (x \vee y)$$

Let's transform it by mapping each bitwise expression within the simplification table for 2

variables shown above and we obtain:

$$
\begin{aligned}
&(x \wedge \neg y) \cdot (\neg x \wedge y) + (x \wedge y) \cdot (x \vee y) = \\
&(x - x \wedge y) \cdot (y - x \wedge y) + (x \wedge y) \cdot (x + y - x \wedge y) = \\
&xy - x \cdot (x \wedge y) - (x \wedge y) \cdot y + (x \wedge y)^2 + (x \wedge y) \cdot x + (x \wedge y) \cdot y - (x \wedge y)^2 = \\
&xy
\end{aligned}
$$

The interested reader can refer to [Xu+21] for a discussion on some other *classical* optimization techniques used alongside the process as well as a detailed algorithm description and implementation decisions.

Moreover, a prototype tool is available[2], which includes not only the source code, but also the simplification tables for expressions on 2, 3 and 4 variables as well as the full dataset for the testbed used throughout the evaluation.

2.3.2 Flaws and limitations

MBA alternation as the dominant factor influencing solvers' performance

The authors analyze the impact of several MBA complexity metrics in the scalability issues of SMT solvers to address MBA solving. The metric they decide to focus upon within their approach is MBA alternation (i.e. the number of operations that connect arithmetic and bitwise operations). While MBA alternation is definitely one of the dominant factors, it does not seem to be the only one.

From the very same data and graphics they provide, it looks like both the MBA length (considering the MBA expression as a character string) and the number of terms of the MBA expression play a similar dominant role.

Constants in bitwise expressions

As it has already been pointed out when discussing flaws in the MBA-Blast [Liu+21] simplification approach, the MBA expressions addressed within the MBA-Solver transformation cannot contain constants other than 0 and -1 within the bitwise expressions in them, due to the very same reason: the possibility of using a 1-bit truth table mapping required for these bitwise expressions relies on the n-bit to 1-bit bijective transformation for linear MBA expressions we have presented before.

Such mapping is not be possible if we have arbitrary constants on the bitwise expressions, as there is no possible *1-to-1* semantics preserving mapping between an n-bit constant to a 1-bit constant other than 0 and -1.

Problem definition

While the authors state the problem of MBA simplification in general, the actual problem they are evaluating, referred as *MBA Solving* throughout the paper, is actually proving the semantic equivalence between an original *ground truth* MBA expression and an obfuscated MBA expression which has been transformed into a *simpler* version by reducing MBA alternation.

[2] `https://github.com/softsec-unh/MBA-Solver`

This is of course relevant, but it is important to remark that verifying the semantic equivalence between different MBA expressions is a different program analysis problem than finding a simplified (ideally *minimal*) MBA expression from an arbitrarily obfuscated one (without access to the original non-obfuscated ground truth).

Chapter 3

Improving program synthesis based MBA simplification reliability

3.1 Motivation

Through the study of the orthogonal approaches to reason about and simplify MBA expressions that we have explored in the previous chapter, we have developed a better understanding of some underlying mathematical properties of MBA expressions. Focusing on a subset of linear MBA expressions, we are now able to produce a semantics preserving transformation (simplification) of a target MBA expression so that SMT solvers have a better chance to handle them.

Even if such approach does not seem to derive the *simplest* expression possible, the nature of this semantics preserving transformation allows us to improve the reliability of simplification attempts based of program synthesis.

In particular, we can use this simplified expression as the reference expression against which the semantic equivalence of a synthesized candidate can be checked and verified.

3.2 Method description

We propose the following methodology which combines symbolic execution, program synthesis and orthogonal reasoning to address the problem of simplifying (deobfuscating) MBA expressions.

- Starting with a snippet of assembly code (either from static analysis or an execution trace), we leverage a symbolic execution engine to extract the formula which encodes the value of a given variable (register or memory location) at the end of the obfuscated code being analyzed, with respect to the input variables (registers and/or memory locations) that play a role in the computation of the output variable we are targeting.
- Then, we apply *classic* simplification strategies to the formula retrieved in the previous step, usually with the help of an SMT solver. These simplification strategies mainly rely on isolated simplifications on the boolean and arithmetic subexpressions, as well as term grouping. This simplification process is semantics preserving by nature. Thus, we arrive to our first simplified candidate. Unfortunately, as we have discussed in Section 1.3.1, an adversary might thwart such approach by arbitrarily increasing the syntactic complexity

of the obfuscated code, leading to an expression which is still far from showcasing the underlying semantics of the code in a clear and concise way.

- To overcome these limitations, we introduce a program synthesis based simplification process. In particular, we leverage the *QSynth* algorithm to derive a synthesized expression. This expression will fit the semantic behavior of the obfuscated code based on the I/O behavior. The I/O samples that describe the behavior of the obfuscated code are obtained by picking a set of input values, evaluating the obfuscated expression (or emulating the obfuscated code) and collecting the output value. Although this approach tends to work really nice, this synthesis process does not guarantee the semantic equivalence between the synthesized expression and the original (obfuscated) one, as it is derived from a *necessarily finite* set of I/O samples.

- We could try to plug an SMT solver to check for the correctness of the synthesized expression, by attempting to prove its semantic equivalence against the original expression (or the first simplification). However, the chance of success of the SMT solver for this task is directly tied to the syntactic complexity of the expression against which the synthesized candidate is being compared. This complexity can be controlled and arbitrarily increased by the adversary, as we have discussed in 1.3.1, even for the first simplified expression. This is due to the fact that the *classic* simplification strategies mainly rely on simplifying boolean and arithmetic subexpressions as separate entities. Thus, they cannot break MBA obfuscation that might have been deliberately placed at the heart of the obfuscated code.

- In order to verify the correctness of the synthesized expression, we propose the introduction of an extra simplification step based on the *MBA-Solver* orthogonal approach. Starting from the first simplified expression, we leverage the methodology discussed in Section 2.3 to produce a second simplified expression which deliberately reduces the MBA alternation complexity by construction, while exposing more opportunities for *classic* simplification approaches to work during the process. This transformation process is semantics preserving by construction.

- Finally, now we can leverage an SMT solver to verify the correctness of the synthesized expression by checking its semantic equivalence against the newly simplified expression. In this case, such simplified expression is deliberately constructed to be *easily digestible* by the SMT solver. Thus, making the semantic equivalence check feasible from a practical standpoint.

At the end of the process, we end up with a *verified synthesized expression* than matches the semantic behavior of the obfuscated expression (code).

Remark 3.1. In case the semantic equivalence check would not succeed, the SMT solver would provide a counterexample (i.e. an input for which the expressions do not produce the same output). This input can then be used to guide a subsequent program synthesis iteration, where a candidate program will have to be synthesized accounting for the previous I/O mismatch.

The method described effectively improves the reliability of current state-of-the-art approaches (which end up generating a synthesized candidate expression) by means of making the correctness of the synthesized candidate expression to be easily verifiable.

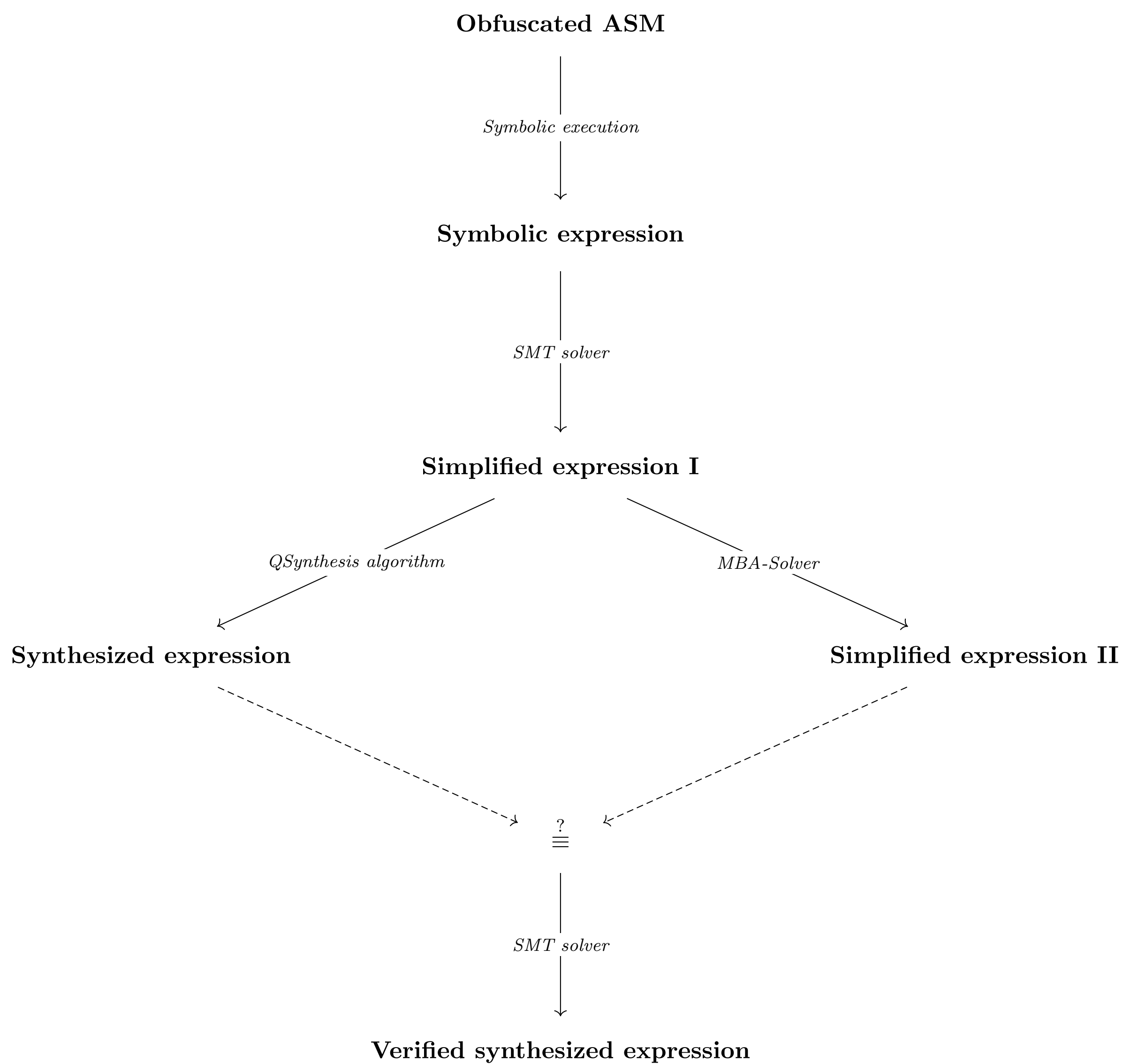


Figure 3.1: Workflow diagram of the methodology proposed

3.3 Practical application: a guided example

Consider the following (obfuscated) code in *x86-64* assembly taken from the sample executable file *monster*. In particular, this is a function with a single basic block, which consists of 1889 assembly instructions. The function receives two parameters in the registers *rdi* and *rsi*, makes (a lot of) computations and stores the resulting value in the register *rax* to be returned[1].

An excerpt of the first and last instructions is shown below.

```
0x00001149      f30f1efa       endbr64
0x0000114d      55             push rbp
0x0000114e      4889e5         mov rbp, rsp
0x00001151      53             push rbx
0x00001152      48897df0       mov qword [var_10h], rdi     ; arg1
0x00001156      488975e8       mov qword [var_18h], rsi     ; arg2
0x0000115a      488b55e8       mov rdx, qword [var_18h]
0x0000115e      488b45f0       mov rax, qword [var_10h]
0x00001162      4c8d0402       lea r8, [rdx + rax]
0x00001166      488b45f0       mov rax, qword [var_10h]
0x0000116a      482b45e8       sub rax, qword [var_18h]
0x0000116e      4889c2         mov rdx, rax
0x00001171      488b4de8       mov rcx, qword [var_18h]
0x00001175      488b45f0       mov rax, qword [var_10h]
0x00001179      4c8d0c01       lea r9, [rcx + rax]
0x0000117d      488b45f0       mov rax, qword [var_10h]
0x00001181      482b45e8       sub rax, qword [var_18h]
[...]
0x00002b43      488d0c00       lea rcx, [rax + rax]
0x00002b47      488b45f0       mov rax, qword [var_10h]
0x00002b4b      482b45e8       sub rax, qword [var_18h]
0x00002b4f      4801c8         add rax, rcx
0x00002b52      488d0c07       lea rcx, [rdi + rax]
0x00002b56      488b45e8       mov rax, qword [var_18h]
0x00002b5a      4801c8         add rax, rcx
0x00002b5d      4801c2         add rdx, rax
0x00002b60      488b45e8       mov rax, qword [var_18h]
0x00002b64      4801d0         add rax, rdx
0x00002b67      4801c0         add rax, rax
0x00002b6a      4929c1         sub r9, rax
0x00002b6d      4c89c8         mov rax, r9
0x00002b70      4c09c0         or rax, r8
0x00002b73      5b             pop rbx
0x00002b74      5d             pop rbp
0x00002b75      c3             ret
```

Now, we will use the symbolic execution framework Miasm[2] to symbolically execute this function and retrieve the formula for the output value of the register *rax* with respect to the input values in registers *rdi* and *rsi*.

To do so, we create a function that receives a path to an executable file and a start address, and returns the symbolic expression for the register *rax* after symbolically executing the basic block where the start address is located.

[1] https://en.wikipedia.org/wiki/X86_calling_conventions#System_V_AMD64_ABI

[2] https://github.com/cea-sec/miasm

```python
"""
Adapted from ./msynth/scripts/symbolic_simplification.py
"""
from miasm.analysis.binary import Container
from miasm.analysis.machine import Machine
from miasm.core.locationdb import LocationDB
from miasm.ir.symbexec import SymbolicExecutionEngine

def getRaxExpr(file_path, start_addr):
    # symbol table
    loc_db = LocationDB()

    # open the binary for analysis
    container = Container.from_stream(open(file_path, 'rb'), loc_db)

    # cpu abstraction
    machine = Machine(container.arch)

    # init disassemble engine
    mdis = machine.dis_engine(container.bin_stream, loc_db=loc_db)

    # initialize intermediate representation
    lifter = machine.lifter_model_call(mdis.loc_db)

    # disassemble the function at address
    asm_block = mdis.dis_block(start_addr)

    # lift to Miasm IR
    ira_cfg = lifter.new_ircfg()
    lifter.add_asmblock_to_ircfg(asm_block, ira_cfg)

    # init symbolic execution engine
    sb = SymbolicExecutionEngine(lifter)

    # symbolically execute basic block
    sb.run_block_at(ira_cfg, start_addr)

    # return the expression of rax (return value)
    return sb.eval_exprid(lifter.arch.regs.RAX)
```

We call this function with the appropriate file path and address, and print the resulting symbolic expression extracted.

```python
file_path = "./monster"
addr = 0x1149

rax_exp = getRaxExpr(file_path, addr)
print(f"RAX value from symbolic execution:\n---\n{rax_exp}\n")
```

If we run this code, we obtain the following *massive* expression that encodes the computed value of the register *rax* returned by the function, with respect to initial values in registers *rdi* and *rsi*.

Remark 3.2. The font size for the output expression has been deliberately reduced to fit in a single page.

```
RAX value from symbolic execution:
---

(RDI + RSI + (RDI * 0x6 + RSI * 0xFFFFFFFFFFFFFFFA + (RSI & (RDI ^ 0xFFFFFFFFFFFFFFFF)) * 0x2 + (RDI + RSI + (RDI & RSI) * 0xFFFFFFFFFFFFFFFE
↪  + -(RSI & (RDI ^ 0xFFFFFFFFFFFFFFFF))) * 0xFFFFFFFFFFFFFFFE + ((RSI & (RDI ^ 0xFFFFFFFFFFFFFFFF)) + RDI * 0xFFFFFFFFFFFFFFFC + RSI *
↪  0xFFFFFFFFFFFFFFFC + (RDI & RSI) * 0x2 + (RDI + (RDI & RSI) + (-RSI + (RDI | RSI)) * 0xFFFFFFFFFFFFFFFE + -RSI + (RDI ^ RSI)) * 0x2 + (RDI
↪  * 0xFFFFFFFFFFFFFFFE + RSI * 0x2 + (((RDI | RSI) ^ 0xFFFFFFFFFFFFFFFF) * 0xFFFFFFFFFFFFFFFE + -RDI + -(RDI | RSI) + 0xFFFFFFFFFFFFFFFE) *
↪  0xFFFFFFFFFFFFFFFE + (-RSI + (RDI | RSI)) * 0x2 + -(RDI & RSI) + -(RDI ^ RSI) + 0xFFFFFFFFFFFFFFFF) * 0xFFFFFFFFFFFFFFFE + (-(RDI & RSI) +
↪  -(RDI ^ RSI) + 0xFFFFFFFFFFFFFFFF) * 0xFFFFFFFFFFFFFFFE + 0xFFFFFFFFFFFFFFFC) * 0x2 + ((RSI & (RDI ^ 0xFFFFFFFFFFFFFFFF)) + RDI *
↪  0xFFFFFFFFFFFFFFFE + RSI * 0xFFFFFFFFFFFFFFFE + (RDI & RSI) * 0x2 + (-(RDI & RSI) + -(RDI ^ RSI) + 0xFFFFFFFFFFFFFFFF) *
↪  0xFFFFFFFFFFFFFFFE + 0xFFFFFFFFFFFFFFFE) * 0x2 + (RDI * 0x3 + RSI * 0x3 + (RDI & RSI) * 0xFFFFFFFFFFFFFFFE + (RDI + (RDI & RSI) + (-RSI +
↪  (RDI | RSI)) * 0xFFFFFFFFFFFFFFFE + -RSI + (RDI ^ RSI)) * 0xFFFFFFFFFFFFFFFE + (-(RDI & RSI) + -(RDI ^ RSI) + 0xFFFFFFFFFFFFFFFF) * 0x2 +
↪  -(RSI & (RDI ^ 0xFFFFFFFFFFFFFFFF)) + 0x2) * 0xFFFFFFFFFFFFFFFE + (RDI * 0x5 + RSI * 0x5 + (RDI & RSI) * 0xFFFFFFFFFFFFFFFE + (RDI + (RDI
↪  & RSI) + (-RSI + (RDI | RSI)) * 0xFFFFFFFFFFFFFFFE + -RSI + (RDI ^ RSI)) * 0xFFFFFFFFFFFFFFFE + ((RDI & RSI) + RDI * 0x3 + RSI *
↪  0xFFFFFFFFFFFFFFFD + (RSI + RDI * 0x2 + (-(RDI ^ RSI) + (RDI | RSI)) * 0xFFFFFFFFFFFFFFFE + ((RDI | RSI) ^ 0xFFFFFFFFFFFFFFFF) * 0x2 +
↪  (RDI | RSI) + 0x2) * 0xFFFFFFFFFFFFFFFE + (((RDI | RSI) ^ 0xFFFFFFFFFFFFFFFF) * 0xFFFFFFFFFFFFFFFE + -RDI + -(RDI | RSI) +
↪  0xFFFFFFFFFFFFFFFE) * 0x2 + (-RSI + (RDI | RSI)) * 0xFFFFFFFFFFFFFFFE + (RDI ^ RSI)) * 0xFFFFFFFFFFFFFFFE + (RDI * 0xFFFFFFFFFFFFFFFE +
↪  RSI * 0x2 + (((RDI | RSI) ^ 0xFFFFFFFFFFFFFFFF) * 0xFFFFFFFFFFFFFFFE + -RDI + -(RDI | RSI) + 0xFFFFFFFFFFFFFFFE) * 0xFFFFFFFFFFFFFFFE +
↪  (-RSI + (RDI | RSI)) * 0x2 + -(RDI & RSI) + -(RDI ^ RSI) + 0xFFFFFFFFFFFFFFFF) * 0x2 + (-(RDI & RSI) + -(RDI ^ RSI) + 0xFFFFFFFFFFFFFFFF)
↪  * 0x2 + -(RSI & (RDI ^ 0xFFFFFFFFFFFFFFFF)) + 0x4) * 0xFFFFFFFFFFFFFFFE + -(RDI ^ RSI) + (RDI | RSI)) * 0xFFFFFFFFFFFFFFFE) | (RDI + RSI *
↪  0xF + (RDI & RSI) * 0xFFFFFFFFFFFFFFFE + (RSI & (RDI ^ 0xFFFFFFFFFFFFFFFF)) * 0xFFFFFFFFFFFFFFFD + (RDI + (RDI & RSI) + (-RSI + (RDI |
↪  RSI)) * 0xFFFFFFFFFFFFFFFE + -RSI + (RDI ^ RSI)) * 0xFFFFFFFFFFFFFFFE + ((RDI & RSI) + RDI * 0x3 + RSI * 0xFFFFFFFFFFFFFFFD + (RSI + RDI *
↪  0x2 + (-(RDI ^ RSI) + (RDI | RSI)) * 0xFFFFFFFFFFFFFFFE + ((RDI | RSI) ^ 0xFFFFFFFFFFFFFFFF) * 0x2 + (RDI | RSI) + 0x2) *
↪  0xFFFFFFFFFFFFFFFE + (((RDI | RSI) ^ 0xFFFFFFFFFFFFFFFF) * 0xFFFFFFFFFFFFFFFE + -RDI + -(RDI | RSI) + 0xFFFFFFFFFFFFFFFE) * 0x2 + (-RSI +
↪  (RDI | RSI)) * 0xFFFFFFFFFFFFFFFE + (RDI ^ RSI)) * 0xFFFFFFFFFFFFFFFE + ((RDI & RSI) + RDI * 0x5 + RSI * 0xFFFFFFFFFFFFFFFB + (RSI + RDI *
↪  0x2 + (-(RDI ^ RSI) + (RDI | RSI)) * 0xFFFFFFFFFFFFFFFE + ((RDI | RSI) ^ 0xFFFFFFFFFFFFFFFF) * 0x2 + (RDI | RSI) + 0x2) *
↪  0xFFFFFFFFFFFFFFFE + (RDI * 0x4 + RSI * 0x3 + (RSI + (RSI & (RDI ^ 0xFFFFFFFFFFFFFFFF)) * 0xFFFFFFFFFFFFFFFE + -RDI + -(RDI | RSI) + (RDI
↪  ^ RSI) + 0xFFFFFFFFFFFFFFFF) * 0x2 + (RDI * 0x2 + RSI * 0xFFFFFFFFFFFFFFFE + (RSI & (RDI ^ 0xFFFFFFFFFFFFFFFF)) * 0x2 + (RDI + RSI + (RDI
↪  & RSI) * 0xFFFFFFFFFFFFFFFE + -(RSI & (RDI ^ 0xFFFFFFFFFFFFFFFF))) * 0xFFFFFFFFFFFFFFFE + -(RDI ^ RSI) + (RDI | RSI)) * 0xFFFFFFFFFFFFFFFE
↪  + (-(RDI ^ RSI) + (RDI | RSI)) * 0xFFFFFFFFFFFFFFFE + ((RDI | RSI) ^ 0xFFFFFFFFFFFFFFFF) * 0x2 + (RDI | RSI) + 0x4) * 0xFFFFFFFFFFFFFFFE +
↪  (RDI * 0xFFFFFFFFFFFFFFFD + RSI * 0xFFFFFFFFFFFFFFFE + (RSI + (RSI & (RDI ^ 0xFFFFFFFFFFFFFFFF)) * 0xFFFFFFFFFFFFFFFE + -RDI + -(RDI |
↪  RSI) + (RDI ^ RSI) + 0xFFFFFFFFFFFFFFFF) * 0xFFFFFFFFFFFFFFFE + (-(RDI ^ RSI) + (RDI | RSI)) * 0x2 + ((RDI | RSI) ^ 0xFFFFFFFFFFFFFFFF) *
↪  0xFFFFFFFFFFFFFFFE + -(RDI | RSI) + 0xFFFFFFFFFFFFFFFC) * 0x2 + (((RDI | RSI) ^ 0xFFFFFFFFFFFFFFFF) * 0xFFFFFFFFFFFFFFFE + -RDI + -(RDI |
↪  RSI) + 0xFFFFFFFFFFFFFFFE) * 0x2 + (-RSI + (RDI | RSI)) * 0xFFFFFFFFFFFFFFFE + (RDI ^ RSI)) * 0xFFFFFFFFFFFFFFFE + ((RSI & (RDI ^
↪  0xFFFFFFFFFFFFFFFF)) + RDI * 0xFFFFFFFFFFFFFFFB + RSI * 0xFFFFFFFFFFFFFFFB + (RDI + (RDI & (RSI ^ 0xFFFFFFFFFFFFFFFF)) *
↪  0xFFFFFFFFFFFFFFFE + -(RSI | (RDI ^ 0xFFFFFFFFFFFFFFFF)) + 0xFFFFFFFFFFFFFFFF) * 0x2 + (RSI + RDI * 0xFFFFFFFFFFFFFFFE + (RDI & (RSI ^
↪  0xFFFFFFFFFFFFFFFF)) * 0x2 + (-(RDI & (RSI ^ 0xFFFFFFFFFFFFFFFF)) + (RDI ^ RSI)) * 0xFFFFFFFFFFFFFFFE + (RSI | (RDI ^
↪  0xFFFFFFFFFFFFFFFF))) * 0xFFFFFFFFFFFFFFFE + (RDI * 0x3 + RSI * 0xFFFFFFFFFFFFFFFE + (RDI & (RSI ^ 0xFFFFFFFFFFFFFFFF)) *
↪  0xFFFFFFFFFFFFFFFE + (RDI + RSI + (RDI & (RSI ^ 0xFFFFFFFFFFFFFFFF)) + (RSI + (RDI | (RSI ^ 0xFFFFFFFFFFFFFFFF)) + 0x1) *
↪  0xFFFFFFFFFFFFFFFE + -(RDI ^ RSI)) * 0xFFFFFFFFFFFFFFFE + (-(RDI & (RSI ^ 0xFFFFFFFFFFFFFFFF)) + (RDI ^ RSI)) * 0x2 + -(RSI | (RDI ^
↪  0xFFFFFFFFFFFFFFFF)) + 0xFFFFFFFFFFFFFFFF) * 0x2 + (RDI * 0xFFFFFFFFFFFFFFFC + RSI * 0x3 + (RDI & (RSI ^ 0xFFFFFFFFFFFFFFFF)) * 0x2 + (RDI
↪  + RSI + (RDI & (RSI ^ 0xFFFFFFFFFFFFFFFF)) + (RSI + (RDI | (RSI ^ 0xFFFFFFFFFFFFFFFF)) + 0x1) * 0xFFFFFFFFFFFFFFFE + -(RDI ^ RSI)) * 0x2 +
↪  (RDI * 0xFFFFFFFFFFFFFFFE + RSI * 0xFFFFFFFFFFFFFFFE + (RSI + (RDI | (RSI ^ 0xFFFFFFFFFFFFFFFF)) + 0x1) * 0x2 + ((RSI & (RDI ^
↪  0xFFFFFFFFFFFFFFFF)) * 0xFFFFFFFFFFFFFFFE + -RDI + -(RDI | (RSI ^ 0xFFFFFFFFFFFFFFFF)) + 0xFFFFFFFFFFFFFFFE) * 0xFFFFFFFFFFFFFFFE + -(RDI
↪  & (RSI ^ 0xFFFFFFFFFFFFFFFF)) + (RDI ^ RSI) + 0xFFFFFFFFFFFFFFFE) * 0xFFFFFFFFFFFFFFFE + (-(RDI & (RSI ^ 0xFFFFFFFFFFFFFFFF)) + (RDI ^
↪  RSI)) * 0xFFFFFFFFFFFFFFFE + (RSI | (RDI ^ 0xFFFFFFFFFFFFFFFF))) * 0xFFFFFFFFFFFFFFFE + (-RSI + (RSI | (RDI ^ 0xFFFFFFFFFFFFFFFF))) *
↪  0xFFFFFFFFFFFFFFFE + -(RDI ^ RSI) + 0xFFFFFFFFFFFFFFFA) * 0xFFFFFFFFFFFFFFFE + ((RSI & (RDI ^ 0xFFFFFFFFFFFFFFFF)) + RDI *
↪  0xFFFFFFFFFFFFFFFD + RSI * 0xFFFFFFFFFFFFFFFD + (RDI + (RDI & (RSI ^ 0xFFFFFFFFFFFFFFFF)) * 0xFFFFFFFFFFFFFFFE + -(RSI | (RDI ^
↪  0xFFFFFFFFFFFFFFFF)) + 0xFFFFFFFFFFFFFFFF) * 0x2 + (RSI + RDI * 0xFFFFFFFFFFFFFFFE + (RDI & (RSI ^ 0xFFFFFFFFFFFFFFFF)) * 0x2 + (-(RDI &
↪  (RSI ^ 0xFFFFFFFFFFFFFFFF)) + (RDI ^ RSI)) * 0xFFFFFFFFFFFFFFFE + (RSI | (RDI ^ 0xFFFFFFFFFFFFFFFF))) * 0xFFFFFFFFFFFFFFFE + (-RSI + (RSI
↪  | (RDI ^ 0xFFFFFFFFFFFFFFFF))) * 0xFFFFFFFFFFFFFFFE + -(RDI ^ RSI) + 0xFFFFFFFFFFFFFFFC) * 0xFFFFFFFFFFFFFFFE + ((RSI & (RDI ^
↪  0xFFFFFFFFFFFFFFFF)) + (-RSI + (RSI | (RDI ^ 0xFFFFFFFFFFFFFFFF))) * 0xFFFFFFFFFFFFFFFE + -RDI + -RSI + -(RDI ^ RSI) + 0xFFFFFFFFFFFFFFFE)
↪  * 0xFFFFFFFFFFFFFFFE + (RDI * 0x2 + RSI * 0x2 + (RDI + (RDI & (RSI ^ 0xFFFFFFFFFFFFFFFF)) * 0xFFFFFFFFFFFFFFFE + -(RSI | (RDI ^
↪  0xFFFFFFFFFFFFFFFF)) + 0xFFFFFFFFFFFFFFFF) * 0xFFFFFFFFFFFFFFFE + (-RSI + (RSI | (RDI ^ 0xFFFFFFFFFFFFFFFF))) * 0x2 + -(RSI & (RDI ^
↪  0xFFFFFFFFFFFFFFFF)) + (RDI ^ RSI) + 0x2) * 0x2 + (RDI * 0x4 + RSI * 0x4 + (RDI + (RDI & (RSI ^ 0xFFFFFFFFFFFFFFFF)) * 0xFFFFFFFFFFFFFFFE
↪  + -(RSI | (RDI ^ 0xFFFFFFFFFFFFFFFF)) + 0xFFFFFFFFFFFFFFFF) * 0xFFFFFFFFFFFFFFFE + (RSI + RDI * 0xFFFFFFFFFFFFFFFE + (RDI & (RSI ^
↪  0xFFFFFFFFFFFFFFFF)) * 0x2 + (-(RDI & (RSI ^ 0xFFFFFFFFFFFFFFFF)) + (RDI ^ RSI)) * 0xFFFFFFFFFFFFFFFE + (RSI | (RDI ^
↪  0xFFFFFFFFFFFFFFFF))) * 0x2 + (RDI * 0x3 + RSI * 0xFFFFFFFFFFFFFFFE + (RDI & (RSI ^ 0xFFFFFFFFFFFFFFFF)) * 0xFFFFFFFFFFFFFFFE + (RDI + RSI
↪  + (RDI & (RSI ^ 0xFFFFFFFFFFFFFFFF)) + (RSI + (RDI | (RSI ^ 0xFFFFFFFFFFFFFFFF)) + 0x1) * 0xFFFFFFFFFFFFFFFE + -(RDI ^ RSI)) *
↪  0xFFFFFFFFFFFFFFFE + (-(RDI & (RSI ^ 0xFFFFFFFFFFFFFFFF)) + (RDI ^ RSI)) * 0x2 + -(RSI | (RDI ^ 0xFFFFFFFFFFFFFFFF)) + 0xFFFFFFFFFFFFFFFF)
↪  * 0xFFFFFFFFFFFFFFFE + (-RSI + (RSI | (RDI ^ 0xFFFFFFFFFFFFFFFF))) * 0x2 + -(RSI & (RDI ^ 0xFFFFFFFFFFFFFFFF)) + (RDI ^ RSI) + 0x4) * 0x2
↪  + (RDI * 0xFFFFFFFFFFFFFFFC + RSI * 0x4 + (RSI + RDI * 0x2 + (-(RDI ^ RSI) + (RDI | RSI)) * 0xFFFFFFFFFFFFFFFE + ((RDI | RSI) ^
↪  0xFFFFFFFFFFFFFFFF) * 0x2 + (RDI | RSI) + 0x2) * 0x2 + (RDI * 0xFFFFFFFFFFFFFFFD + RSI * 0xFFFFFFFFFFFFFFFE + (RSI + (RSI & (RDI ^
↪  0xFFFFFFFFFFFFFFFF)) * 0xFFFFFFFFFFFFFFFE + -RDI + -(RDI | RSI) + (RDI ^ RSI) + 0xFFFFFFFFFFFFFFFF) * 0xFFFFFFFFFFFFFFFE + (-(RDI ^ RSI) +
↪  (RDI | RSI)) * 0x2 + ((RDI | RSI) ^ 0xFFFFFFFFFFFFFFFF) * 0xFFFFFFFFFFFFFFFE + -(RDI | RSI) + 0xFFFFFFFFFFFFFFFC) * 0xFFFFFFFFFFFFFFFE +
↪  (((RDI | RSI) ^ 0xFFFFFFFFFFFFFFFF) * 0xFFFFFFFFFFFFFFFE + -RDI + -(RDI | RSI) + 0xFFFFFFFFFFFFFFFE) * 0xFFFFFFFFFFFFFFFE + (-RSI + (RDI |
↪  RSI)) * 0x2 + -(RDI & RSI) + -(RDI ^ RSI) + 0xFFFFFFFFFFFFFFFF) * 0x2 + (RDI * 0xFFFFFFFFFFFFFFFE + RSI * 0x2 + (((RDI | RSI) ^
↪  0xFFFFFFFFFFFFFFFF) * 0xFFFFFFFFFFFFFFFE + -RDI + -(RDI | RSI) + 0xFFFFFFFFFFFFFFFE) * 0xFFFFFFFFFFFFFFFE + (-RSI + (RDI | RSI)) * 0x2 +
↪  -(RDI & RSI) + -(RDI ^ RSI) + 0xFFFFFFFFFFFFFFFF) * 0x2 + (-(RDI & RSI) + -(RDI ^ RSI) + 0xFFFFFFFFFFFFFFFF) * 0x2 + (-(RSI & (RDI ^
↪  0xFFFFFFFFFFFFFFFF)) + (RDI ^ RSI)) * 0x2 + -(RDI | (RSI ^ 0xFFFFFFFFFFFFFFFF)) + 0x5)
```

Then, we plug an SMT solver to produce a first simplified expression. Again, we use *Z3* through its convenient *Python API*. Concretely, after translating the Miasm expression into *Z3*'s representation, we call the *Z3*'s method *simplify*.

This procedure will essentially apply several *classic* simplification strategies to the boolean and arithmetic subexpressions. Another significant outcome achieved through this step will be the grouping of terms.

```
from z3 import *
from miasm.ir.translators.z3_ir import TranslatorZ3

translator = TranslatorZ3()
rax_exp_z3 = translator.from_expr(rax_exp)
rax_exp_z3_simp = simplify(rax_exp_z3)
print(f"RAX after z3 simplification:\n---\n{rax_exp_z3_simp}\n")
```

Running this code, we obtain our first simplified expression. We can easily observe a substantial reduction in total terms and a more comfortable term grouping.

```
RAX after z3 simplification:
---
128 +
49*RDI +
73*RSI +
18446744073709551592*~(~RSI | RDI) +
18446744073709551496*~(~RDI | ~RSI) +
126*(RDI | RSI) +
18446744073709551570*(RDI ^ RSI) +
128*~(RDI | RSI) |
59 +
RDI +
147*RSI +
18446744073709551570*~(~RDI | ~RSI) +
18446744073709551489*~(~RSI | RDI) +
18446744073709551596*(RDI | RSI) +
80*~(RDI | RSI) +
160*~(~RDI | RSI) +
60*(RSI | ~RDI) +
18446744073709551535*(RDI | ~RSI)
```

Nevertheless, this expression still showcases a huge syntactic complexity: we are not able to *grasp* its semantic behavior at all.

Next, we leverage the *QSynth* algorithm implemented in msynth[3] to produce a synthesized expression which encodes the semantic behavior of the obfuscated expression (code).

Remark 3.3. In this example, we deliberately synthesize the original extracted expression instead of the first simplified iteration. This is done to emphasize the fact that through program synthesis approaches we are able to reason about the semantic behavior regardless of the syntactic complexity of the expression (code) analyzed.

```
from msynth import Simplifier

# Path for the serialized oracle tables for QSynth
oracle_path = "./msynth/oracle.pickle"

# Initialize QSynth-based simplifier
synthSimplifier = Simplifier(oracle_path)
```

[3] https://github.com/mrphrazer/msynth

```
rax_exp_synth = synthSimplifier.simplify(rax_exp)
rax_exp_synth_z3 = translator.from_expr(rax_exp_synth)

print(f"RAX after synthesis simplification:\n---\n{rax_exp_synth_z3}\n")
```

After running this code, we obtain the synthesized candidate expression.

```
RAX after synthesis simplification:
---
RDI + RSI | RDI ^ RSI
```

At this point, we have the candidate expression

$$RDI + RSI \vee RDI \oplus RSI$$

that has been synthesized to match the I/O behavior of the obfuscated code. Despite the promising result, we cannot guarantee the correctness of the synthesized expression, as the construction method necessarily relies on a finite set of I/O pairs being considered.

We would like to verify that the synthesized expression (*rax_exp_synth_z3*) is semantically equivalent to the original expression (*rax_exp_z3*), but this is unfeasible from a practical standpoint, as the SMT solver cannot handle such equivalence checking in a *reasonable* amount of time. We could think of checking its equivalence against the previously simplified expression (*rax_exp_z3_simp*), which was obtained through a semantics preserving transformation. However, this expression is still way too complex (from a syntactic point of view) for the SMT solver to handle within an equivalence checking against the synthesized expression.

In order to verify the semantic equivalence of the synthesized expression, we will use the methodology from *MBA-Solver* [Xu+21] to produce another level of simplification through a semantics preserving transformation that will reduce the MBA alternation of the resulting expression to boost the performance of the SMT solver when dealing with it.

Note that the simplified expression *rax_exp_z3_simp* is a non-polynomial MBA expression which is composed of two different linear MBA expression being *ORed* together.

```
128 + 49*RDI + 73*RSI + 18446744073709551592*~(~RSI | RDI) +
↪   18446744073709551496*~(~RDI | ~RSI) + 126*(RDI | RSI) + 18446744073709551570*(RDI
↪   ^ RSI) + 128*~(RDI | RSI)}

|

59 + RDI + 147*RSI + 18446744073709551570*~(~RDI | ~RSI) + 18446744073709551489*~(~RSI
↪   | RDI) + 18446744073709551596*(RDI | RSI) + 80*~(RDI | RSI) + 160*~(~RDI | RSI) +
↪   60*(RSI | ~RDI) + 18446744073709551535*(RDI | ~RSI)
```

The *MBA-Solver* implementation prototype[4] works by passing both the obfuscated and the ground truth expressions, and then *solving* them (i.e. checking their semantic equivalence) after applying the simplification reducing MBA alternation complexity. In our case, we just want to perform the simplification step.

[4] https://github.com/softsec-unh/MBA-Solver

Thus, we create a custom function that will *just* simplify linear MBA expressions. Then, we use this code to simplify the linear MBA subexpressions at each side of the *OR*.

```
import sys
sys.path.append("./MBA-Solver/tools")

from mba_string_operation import verify_mba_unsat, variable_list
from svector_simplify import SvectorSimplify
from truthtable_search_simplify import PMBASimplify

def just_simplify_lmba(datafile):
    svObj = SvectorSimplify()

    with open(datafile, "rt") as fr:
        for line in fr:
            if "#" not in line:
                cmbaExpre = line.strip()
                vnumber = len(variable_list(cmbaExpre))
                simExpre = svObj.standard_simplify(cmbaExpre, vnumber)
                print(simExpre)

just_simplify_lmba("./monster.txt")
```

As you can observe, it will consume linear MBA expressions from a (plain text) data file passed as parameter. In particular for this example, we called it *monster.txt*.

Let's create this file, which will contain the two linear MBA subexpressions we are targeting.

Remark 3.4. We need to *massage* the expressions from their Z3 representation for them to be consumed by the *MBA-Solver* prototype code:

- Replace variables names: $RDI \to x, RSI \to y$
- Substitute the *big* numbers from their unsigned representation (*Z3*'s native) to their signed counterparts. This is currently required due to a type error from an inconsistent use of *NumPy* in the current *MBA-Solver* prototype implementation, probably through a non-handled implicit casting.

After these technical modifications, we create the file *monster.txt* with the two linear MBA subexpressions in plain text as follows:

```
# left sub-expression of OR operation
128+49*x+73*y-24*~(~y|x)-120*~(~x|~y)+126*(x|y)-46*(x^y)+128*~(x|y)
# right sub-expression of OR operation
59+x+147*y-46*~(~x|~y)-127*~(~y|x)-20*(x|y)+80*~(x|y)+160*~(~x|y)+60*(y|~x)-81*(x|~y)
```

Remark 3.5. Lines with # are ignored by the file parser.

When we run the previous code, the output is produced almost immediately. We can now retrieve the two simplified subexpressions.

```
1*(x^y)
1*~(x|~y)+1*(x&~y)+2*(x&y)
```

As we can see, the simplified subexpressions are:

$$
\begin{aligned}
l_{or} &= x \wedge y \\
r_{or} &= \neg(x \vee \neg y) + (x \wedge \neg y) + 2(x \wedge y)
\end{aligned}
$$

Now, the total simplified expression obtained through the semantics preserving transformation from *MBA-Solver* can be created simply by *ORing* together the two simplified subexpressions retrieved above.

Finally, let's plug the original variable names back and check that we can verify the semantic equivalence between the synthesized expression and the newly simplified one.

```
RDI, RSI = BitVecs("RDI RSI", 64)

l_or = RDI^RSI
r_or = ~(RDI|~RSI) + (RDI&~RSI)  + 2*(RDI&RSI)

rax_exp_z3_simp_transform = l_or | r_or

print(f"RAX after MBA-Solver simplification on z3 simplified
↪  expression:\n---\n{rax_exp_z3_simp_transform}\n")

print(f"RAX synthesized candidate expression (recall) :\n---\n{rax_exp_synth_z3}\n")

print("Prove that synthesized candidate and MBA-Solver simplified expression are
↪  equivalent:\n---")
prove(rax_exp_z3_simp_transform == rax_exp_synth_z3)
```

If we run the code, we observe that it successfully verifies the semantic equivalence of the two expressions almost immediately.

```
RAX after MBA-Solver simplification on z3 simplified expression:
---
RDI ^ RSI | ~(RDI | ~RSI) + (RDI & ~RSI) + 2*(RDI & RSI)

RAX synthesized candidate expression (recall) :
---
RDI + RSI | RDI ^ RSI

Prove that synthesized candidate and MBA-Solver simplified expression are equivalent:
---
proved
```

Therefore, we have proved that the initial *heavily obfuscated* assembly code actually represents the following *extremely simple* semantic behavior:

$$
RAX = RDI + RSI \vee RDI \oplus RSI
$$

In other words, the obfuscated function simply performs a 64-bit *OR* operation on the values obtained after *ADDing* and *XORing* the two 64-bit inputs, respectively.

3.4 Limitations

On the one hand, this methodology encompasses a process relying on oracle-guided program synthesis for the generation of candidate expressions that match the I/O behavior of the obfuscated code. Thus, the very same limitations for such a process that were mentioned in Section 1.3.2 apply.

In terms of our specific contribution incorporating orthogonal reasoning to help verifying the correctness of the synthesized candidates, the core limitations come more from a practical standpoint. The prototype implementation for *MBA-Solver* is exactly that; a prototype.

Indeed, we have already found some *more or less problematic* issues when plugging it into our workflow, from having to adapt the variable names into hardcoded ones to needing to reformulate *Z3*'s expressions by changing unsigned values to signed counterparts in order to avoid triggering a bug in its integration with *NumPy*.

Conclusions

The main goal of this project was to study and analyze orthogonal approaches to symbolic execution and program synthesis for simplification of MBA expressions in the context of code deobfuscation. As a secondary goal, we wanted to review recent research literature in this regard and apply it to the problem of MBA simplification in practical scenarios. We have successfully accomplished these objectives through a set of specific achievements.

We introduced the fundamental theoretical concepts and practical mechanisms to contextualize the study of MBA expressions in the field of code obfuscation as well as state-of-the-art deobfuscation approaches and techniques.

Throughout the study and analysis of orthogonal approaches for MBA simplification, we unveiled the most relevant contributions by [Liu+21] and [Xu+21]. We also pointed out some flaws and limitations that were not contemplated in the original research papers.

We have proposed a novel methodology to integrate contributions from the orthogonal approaches analyzed into a simplification workflow, which already combined symbolic execution and program synthesis. Our proposal leverages a semantics preserving transformation on the obfuscated expression by means of reducing its MBA alternation complexity. This transformation enables an SMT solver to deal with such expressions in practice, leading to a feasible verification of the semantic correctness of a synthesized candidate expression.

Finally, we have tested and validated our methodology in a lab environment, providing a detailed guided example to showcase its workflow in a practical scenario. Starting from an arbitrarily obfuscated snippet of assembly code, we advance all the way down to recovering its semantics through a synthesized expression which is verified to be semantically equivalent.

Future work

From a theoretical standpoint, there is clearly still a lot of research that needs to be done to improve our knowledge of MBA expressions in general, and with respect to their underlying mathematical properties in particular. For instance, current orthogonal approaches only address properties of linear MBA expressions (or the embedded linear subexpressions for non-linear MBA expressions), but there is not any known research published on the possibility of extracting some valuable information, invariant properties or semantics preserving transformations that rely on the *mathematical structure* of non-linear MBA expressions.

From a practical standpoint, while we have great frameworks for symbolic execution and reasonably good prototypes for program synthesis and orthogonal approaches, we are still far from having fully integrated software systems that provide all the required mechanisms to build automated and reliable tooling that can handle *any kind* of MBA obfuscated expression from a more *holistic* point of view.

Bibliography

[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.

[Ban+16] Sebastian Banescu et al. "Code Obfuscation against Symbolic Execution Attacks". In: *Proceedings of the 32nd Annual Conference on Computer Security Applications.* ACSAC '16. Los Angeles, California, USA: Association for Computing Machinery, 2016, pp. 189–200. ISBN: 9781450347716. DOI: 10.1145/2991079.2991114. URL: https://doi.org/10.1145/2991079.2991114.

[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.

[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.

[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.

[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/.

[Gul10] Sumit Gulwani. "Dimensions in Program Synthesis". In: *PPDP '10 Hagenberg, Austria.* Jan. 2010. URL: https://www.microsoft.com/en-us/research/publication/dimensions-program-synthesis/.

[JGY15] Xiangyang Jia, Carlo Ghezzi, and Shi Ying. “Enhancing Reuse of Constraint Solutions to Improve Symbolic Execution”. In: *Proceedings of the 2015 International Symposium on Software Testing and Analysis*. ISSTA 2015. Baltimore, MD, USA: Association for Computing Machinery, 2015, pp. 177–187. ISBN: 9781450336208. DOI: 10.1145/2771783.2771806. URL: https://doi.org/10.1145/2771783.2771806.

[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.

[Liu+21] Binbin Liu et al. “MBA-Blast: Unveiling and Simplifying Mixed Boolean-Arithmetic Obfuscation”. In: *30th USENIX Security Symposium (USENIX Security 21)*. USENIX Association, Aug. 2021, pp. 1701–1718. ISBN: 978-1-939133-24-3. URL: https://www.usenix.org/conference/usenixsecurity21/presentation/liu-binbin.

[MB08] Leonardo de Moura and Nikolaj Bjørner. “Z3: An Efficient SMT Solver”. In: *Tools and Algorithms for the Construction and Analysis of Systems*. Ed. by C. R. Ramakrishnan and Jakob Rehof. Berlin, Heidelberg: Springer Berlin Heidelberg, 2008, pp. 337–340. ISBN: 978-3-540-78800-3.

[Men+21] Grégoire Menguy et al. “Search-Based Local Black-Box Deobfuscation: Understand, Improve and Mitigate”. In: *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. CCS ’21. Virtual Event, Republic of Korea: Association for Computing Machinery, 2021, pp. 2513–2525. ISBN: 9781450384544. DOI: 10.1145/3460120.3485250. URL: https://doi.org/10.1145/3460120.3485250.

[Mon20] Arnau Gàmez i Montolio. “Code deobfuscation by program synthesis-aided simplification of Mixed Boolean-Arithmetic expressions”. Bachelor’s Thesis. 2020. URL: http://diposit.ub.edu/dspace/handle/2445/176925.

[PRV11] Corina S. Păsăreanu, Neha Rungta, and Willem Visser. “Symbolic Execution with Mixed Concrete-Symbolic Solving”. In: *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. ISSTA ’11. Toronto, Ontario, Canada: Association for Computing Machinery, 2011, pp. 34–44. ISBN: 9781450305624. DOI: 10.1145/2001420.2001425. URL: https://doi.org/10.1145/2001420.2001425.

[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.

[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 06/05/2020).

[SBP18] Jonathan Salwan, Sébastien Bardin, and Marie-Laure Potet. “Symbolic Deobfuscation: From Virtualized Code Back to the Original”. In: *Detection of Intrusions and Malware, and Vulnerability Assessment*. Ed. by Cristiano Giuffrida, Sébastien Bardin, and Gregory Blanc. Cham: Springer International Publishing, 2018, pp. 372–392. ISBN: 978-3-319-93411-2.

[SS15] Florent Saudel and Jonathan Salwan. “Triton: A Dynamic Symbolic Execution Framework”. In: *Symposium sur la sécurité des technologies de l’information et des communications*. SSTIC. Rennes, France, June 2015, pp. 31–54.

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

[Xu+21] Dongpeng Xu et al. "Boosting SMT Solver Performance on Mixed-Bitwise-Arithmetic Expressions". In: *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*. PLDI 2021. Association for Computing Machinery. New York, NY, USA: Association for Computing Machinery, 2021, 651–664. ISBN: 9781450383912. DOI: `10.1145/3453483.3454068`. URL: `https://doi.org/10.1145/3453483.3454068`.

[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.

[ZM06] Yongxin Zhou and Alec Main. *Diversity Via Code Transformations: A Solution For NGNA Renewable Security*. Tech. rep. The NCTA Technical Papers, 2006.

Appendices

A Proposed objectives

A.1 Principal

- Study and analysis of orthogonal approaches to symbolic execution and program synthesis for simplification of MBA expressions.

A.2 Secondary

- Review and apply recent orthogonal approaches for MBA simplification.

A.3 Specifics

- Study and review the paper *MBA-Blast: Unveiling and Simplifying Mixed Boolean-Arithmetic Obfuscation* [Liu+21].
- Study and review the paper *Boosting SMT Solver Performance on Mixed-Bitwise-Arithmetic Expressions* [Xu+21].
- Test and validate new proposals in a lab environment.

B Logistics

B.1 Temporal planning

The project was divided in concrete tasks, each one with an allocated amount of time to be completed. The concrete task breaking and timing allocation is presented as a Gantt diagram in Figure B.1.

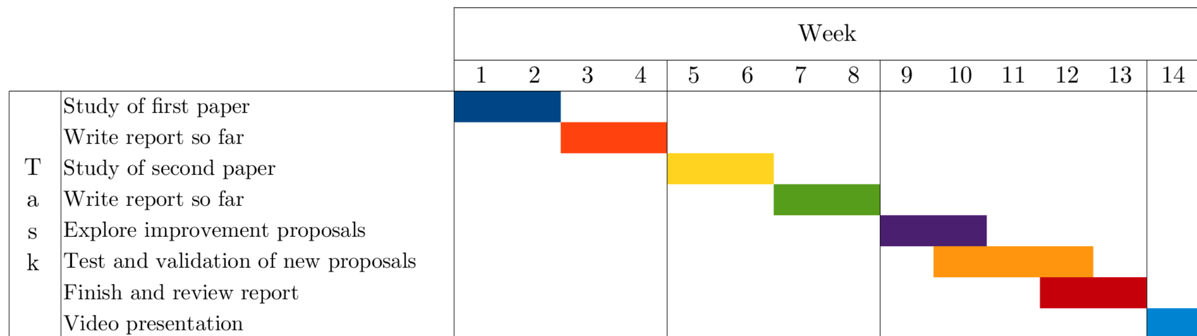


Figure B.1: Gantt diagram representing the project's planning

B.2 Report

The report has been written in LaTeX. We used the online web application Overleaf as the editor. The project was synced with a git repository stored at GitHub. This allowed 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.

B.3 Contact with supervisor

Due to the great specificity of the project's topic, the work has been carried out completely independently. Contact with supervisor has been sporadic, mainly to inform of the incremental advances in the report and collect feedback.