UNIVERSITÀ DEGLI STUDI
DI TRENTO

UNIVERSITY OF TWENTE.

DEPARTMENT OF INFORMATION ENGINEERING AND COMPUTER SCIENCE

Combining Program Synthesis and Symbolic Execution to Deobfuscate Binary Code

Student:
Luigi Coniglio (205007)

Supervisor:
Dr. Mariano Ceccato

Co-Supervisor:
Prof. Andreas Peter

Master's Degree in

Computer Science

Final Dissertation

Academic year 2018/2019

Abstract

Program synthesis consists in automatically derive a program from a high-level specification. In the field of reverse engineering, program synthesis is gaining popularity as a way to deobfuscate obfuscated programs, given their input/output behaviour.

However, most state-of-the-art deobfuscation approaches based on program synthesis assume only black-box oracle access to the obfuscated program, thus trying to solve a harder problem than practical code deobfuscation.

We present a novel program deobfuscation method combining program synthesis and symbolic execution. Our approach works by using symbolic execution to extract the semantic of the obfuscated program and construct an Abstract Syntax Tree (AST) representation of the operations executed. This information is then used to reduce synthesis search space to independent sub-portions of the program. In particular our approach involves the use program synthesis to iteratively simplify the program AST. Our simplification method is independent from the synthesis technique in use.

In the context of our work we also illustrate and apply a program synthesis technique based on pre-computed lookup-tables.

We validate our approach on three datasets of different levels of difficulty, consisting each of 500 randomly generated expressions obfuscated using the popular obfuscation tool *Tigress*.

The results on our datasets show that our approach outperforms current state-of-the-art deobfuscation techniques based on program synthesis.

Acknowledgements

I would like to express my sincere gratitude to my supervisor Dr. Mariano Ceccato for his guidance and continuous support through each stage of my research. I would also like to thank my co-supervisor Prof. Andreas Peter, which teachings I will never forget.

This work would have not been possible without the help and support of my internship supervisor Sébastien Kaczmarek and all the wonderful people I had the honor to work with during my internship at Quarkslab.

In particular I would like to thank my friends and colleagues Robin David and Tim van de Kamp for their help in proofreading this thesis.

Finally, nobody has been more important to me than my family and my girlfriend Jy-Ying in the pursuit of my studies. I would like to thank them for their loving support and understanding.

Contents

List of Figures

List of Tables

Glossary

Abstract Syntax Tree A representation of a program in the form of a tree, where each node represent a term belonging to the program's grammar.

Cyclomatic complexity Software metric indicating the complexity of a program in term of number independent paths.

Dynamic Abstract Syntax Tree An Abstract Syntax Tree constructed using only the portion of the program covered by a given execution.

Dynamic Backward Slice The set of instructions, in a given execution, which contributed in computing a certain target value.

Expression Layer The number of derivations needed to obtain a certain expression. Also used as a complexity metric.

Grammar A context-free grammar defining a set of derivation rules and symbols to describe programs.

Non-terminal Derivation Derivation done applying a non-terminal derivation rule.

Non-terminal Derivation Rule Derivation rule of a grammar which introduces some non-terminal symbol.

Opaque Predicate Expression which value is known at obfuscation time but is difficult to obtain after obfuscation, typically used as predicate in conditional branches.

Shim A form of instrumentation where the original call to a function is intercepted and substituted by a call to another routine.

Symbolic Abstract Syntax Tree Any Dynamic Abstract Syntax Tree where a set of concrete values (i.e. leaves) has been replaced by symbolic variables of the same size.

Synthesis Primitive Any inductive program synthesis technique based on black-box access to an I/O oracle.

Acronyms

ANF Algebraic Normal Form.

AST Abstract Syntax Tree.

BDSE Backward Dynamic Symbolic Execution.

CFG Control Flow Graph.

CG Call Graph.

DBA Dynamic Binary Analysis.

DBI Dynamic Binary Instrumentation.

DSE Dynamic Symbolic Execution.

I/O Inputs/Outputs.

IDEA International Data Encryption Algorithm.

JIT Just In Time.

LUT Lookup-Table.

MBA Mixed Boolean-Arithmetic.

MCTS Monte Carlo Tree Search.

ROP Return Oriented Programming.

SE Symbolic Execution.

SMT Satisfiability Modulo Theories.

Chapter 1

Introduction

Over the last decade software obfuscation has gained in popularity as an approach to protect programs against malicious reverse engineering and tampering. Software obfuscation consists in a semantic-preserving transformation of the original instance of a program P in a "unintelligible" version $\mathscr{O}(P)$. During this transformation, the code is made harder to understand, with the goal of increasing the reverse engineering effort needed by an attacker to recover the original form of the program.

In their work Hosseinzadeh et al. [1] identify some of the most common goals behind the usage of obfuscation techniques in the software security domain. This include:

- *Making reverse engineering of the program logic more difficult*: most of the work on obfuscation aim at making harder to reverse engineer software, protecting code against static and/or dynamic analysis.
- *Preventing unauthorized modification of software*: reducing the understandability of a program has proven to be an effective way of increase resistance against tampering.
- *Hiding data*: obfuscation is often used to hide sensitive static data inside a program (e.g. cryptographic keys as with *white box cryptography*).
- *Prevent exploitation and widespread vulnerabilities*: obfuscation can be used to increase the difficulty of exploiting vulnerabilities and/or as a mean to enhance software diversification to make serial exploitation more challenging.

As today there are a large variety of free and commercial obfuscation software implementing several obfuscation techniques, for instance: insertion of dead code, control flow flattening, arithmetic encoding and use of encryption to pack and unpack software [2, 3, 4].

It is important to highlight however, how "practical"[1] code obfuscation, similarly to other security measures relaying on the *security by obscurity* paradigm, does not provide any strong security guarantee. Indeed obfuscation only contributes to slow down attackers during their analysis, hopefully at the point where the cost of deobfuscating the program surpasses the potential gain [5].

Parallel to the advancements in code obfuscation, the research community has grown an interest in finding smart ways to defeat such protections. This with the goal of strengthening current obfuscation solution and/or facilitate detection and study of malware, where obfuscation is often used to hide malicious code.

Various static and dynamic analysis techniques have been proposed over the years to tackle obfuscation. For instance, techniques such as program slicing, abstract interpretation, tainting and symbolic execution have demonstrated effective against some types of obfuscations [6, 7]. Due to the difficulty of creating a more generic deobfuscation approach, several tools have been developed to target just a particular obfuscation software or technique [8, 9, 10]. However the literature does not lack of more comprehensive deobfuscation approaches. Remarkable is for example the case of Dynamic Binary Analysis (DBA) frameworks such as *Triton* [11] which has demonstrated effective against some of the transformation passes of *Tigress* [3], a state-of-the-art obfuscation software. Other powerful approaches involve the use of program synthesis to deobfuscate a program only

[1] Here we refer to *practical* obfuscation to indicate obfuscation techniques used within real software, as as opposed to *cryptographic* obfuscation techniques which are still considered too costly to be used in real-world scenarios.

based on its input/output behaviour. Synthesis-based approaches treat the obfuscated routine as a black box and are only affected by the semantic complexity of the original operations.

However, in spite of the progresses in anti-obfuscation techniques, the effort needed to break state-of-the-art obfuscations techniques is still non negligible.

1.1 Context of the work

The work object of this master thesis has been conducted in the context of an internship at the company *Quarkslab*, as part of the EIT Digital Master School double degree in Security and Privacy, under the supervision of Sébastien Kaczmarek (*Quarkslab*) as well as Dr. Mariano Ceccato (University of Trento) and co-supervisor Prof. Andreas Peter (University of Twente).

Quarkslab is a cyber security company offering products and services related to the reverse engineering, binary analysis, vulnerability detection and software protection domains. Beside working as a security consulting firm, *Quarkslab* is well known in the security field for the development of commercial solutions for file analysis and software obfuscation (IRMA [12] and Epona [13]) as well as for being behind several tools and research works on the topic of reverse engineering and security testing.

Among the past and on-going works realized at *Quarkslab* relevant to the subject of this thesis we can include: *Triton* [11], a dynamic binary analysis framework, *QBDI* [14], a dynamic binary instrumentation framework, *Arybo* [15], a software for mixed boolean-arithmetic symbolic expressions manipulation, *SSPAM* [16], an expression simplification tool, as well as many others projects focusing on software analysis and deobfuscation. The internship has been conducted in the data-analysis team of *Quarkslab*, which work is mostly focused on reverse engineering and binary analysis.

1.2 Problem definition

Despite the promising results showed by recent works on the topic, we found very little literature examining the application of program synthesis to deobfuscation. For this reason we believe that the potential of program-synthesis applied to the context of program deobfuscation is yet to be fully explored.

Most of the proposed approaches in the literature are based on a "raw" conception of synthesis, were only the input/output behaviour of a program is taken into account. By considering the program under analysis as a black-box oracle, however, those approaches try to solve in practice a harder problem than obfuscation.

Nonetheless an obfuscated program has much more information to offer than only its behaviour in terms of inputs/outputs, such as the operations performed in the input data, the presence/absence of loops, the number of instructions executed, etc. We believe that this information could be exploited to enhance synthesis performance and surpass current state-of-the-art synthesis deobfuscation techniques.

The objective of our work is to design and to validate empirically a novel deobfuscation method for obfuscated binaries. Our approach is based on program synthesis and symbolic execution. In particular, our novel approach takes advantage of symbolic execution to extract the semantic of the program under analysis and use the extracted semantic to reduce synthesis search space and enhance deobfuscation performance.

1.3 Contributions

In this master thesis we make the following contributions:

- We propose an hybrid deobfuscation approach based on the combination of program synthesis and symbolic execution outperforming current state-of-the-art deobfuscation techniques based on program synthesis;
- We propose a lookup-table based synthesis method which can be used to synthesize expression (up to a certain level of complexity) in constant time and is therefore suitable for synthesis-heavy tasks;
- We performed an empirical validation involving comparison with *Syntia* [17], a state-of-the-art synthesis-based deobfuscation tool.

1.4 Outline

In Chapter 2 we provide some background illustrating some of the most popular obfuscation methods and program analysis techniques used to counter obfuscation. In Chapter 3 we illustrate some work on the deobfuscation domain which is strongly related to the work object of this thesis. In Chapter 4 we present our approach, introducing the concepts of Dynamic Abstract Syntax Tree, lookup-table based synthesis primitive and our Abstract Syntax Tree simplification algorithm. In Chapter 5 we discuss some details regarding our implementation of the approach in Chapter 4. In Chapter 6 we present and discuss the result yield by our approach on three different datasets of variable complexity. Finally in Chapter 7 we draw some conclusions and discuss possible improvements to our approach.

Chapter 2

Background

In this chapter we provide some background to our work. In Section 2.1, we introduce the concept of obfuscation as a software protection technique against malicious reverse-engineering and describe some of the most widely used obfuscations. In Section 2.2, we introduce various software analysis techniques often used to defeat obfuscation measures.

2.1 Obfuscation techniques

Obfuscation is a protection measure used to secure software against reverse-engineering *man-at-the-end* (MATE) attacks[1].

An obfuscator can be defined as a (probabilistic) algorithm $\mathscr{O}$ taking as input a program P and returning as a result $\mathscr{O}(P)$, an unintelligible version of P preserving the same functionalities and involving, at most, a polynomial slowdown (i.e. the size and running time of the $\mathscr{O}(P)$ are at most polynomially larger than the those of the original program P).

An ideally-performing obfuscator would transform P in a *virtual black box* in such a way that nothing that could be learned about P by examining $\mathscr{O}(P)$ could not be learned by simply accessing $\mathscr{O}(P)$ as a black box oracle. Such ideal black-box obfuscator has been shown by Barak et al. [18] not to exist.[2]

Practically, current real-world software obfuscation techniques, such as those proposed by Collberg et al. [6] in their work, do not guarantee any form of secrecy over the original program, but simply aim at increasing the effort necessary to reverse-engineer the obfuscated program. This is done by concealing a set of properties of the original program, such as its control flow, the data used during computation, or the arithmetic operations involved in the computations.

In this section we introduce some of the most common obfuscation techniques used in real-world software. We partition them in four categories: *control-flow based*, *data-flow based*, *file format based* and *hybrid techniques*. Other anti-reverse engineering techniques such as anti-debugging and detection of virtual machines will not be discussed in this section.

2.1.1 Control-flow obfuscation

The control-flow indicates control dependencies with which the instructions of a program are executed. This is often expressed in the form of a directed graph using a so-called Control-Flow Graph (CFG). Each vertex of the CFG represents a jump-free portion of code (also called basic block) and each edge represents an explicit jump from a basic block to another. In a CFG we can find all execution paths that a program may take.

Figure 2.1b shows the CFG of the function in 2.1a. Figure 2.1c illustrates the same CFG as displayed by the well-known disassembler and debbugger IDA Pro[3].

[1] A *man-at-the-end* scenarios implies an attacker with full read/write access to the code of target application.

[2] Barak et al. also propose the weaker notion of *indistinguishable obfuscation*. More recently Goldwasser et al. [19] proposed a stronger notion (still more relaxed than black-box) of *best-possible obfuscation*.

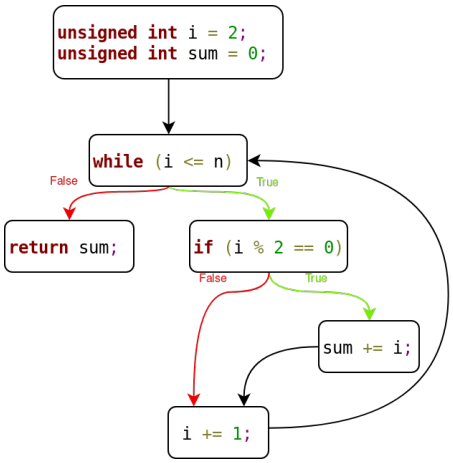
[3] https://www.hex-rays.com/products/ida/

The Call Graph (CG) is another important representation enclosing a lot of important information regarding the control flow of a program. The CG is a directed graph describing all inter-procedural relationships between the routines of a program. Each vertex of the CG represents a function and each edge represents a call from one function to another.

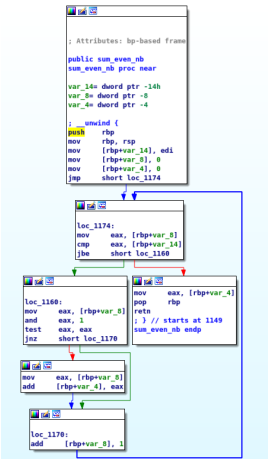
A lot of information about a program, such as the presence of loops, conditional statements, recursion, cyclomatic complexity etc., can be extracted by simply looking at its control-flow. For this reason knowledge about the control-flow is extremely valuable for reversing-engineering purposes. Consequently, several obfuscation techniques aim at hiding the original control-flow of a program to make reverse-engineering more challenging. In this section we describe some of those techniques.

```c
unsigned int sum_even_nb(unsigned int n){

  unsigned int i = 2;
  unsigned int sum = 0;

  while (i <= n){
    if (i % 2 == 0){
      sum += i;
    }
    i += 1;
  }

  return sum;
}
```

(a) Function to compute the sum of all even numbers up to n.



(b) CFG of *sum_even_nb*.



(c) CFG of *sum_even_nb* displayed by IDA Pro.

Figure 2.1: Example of CFG of a function.

Function inlining and function splitting: Function inlining consist in substituting a call to a function f with the body of the function itself. While this technique was originally designed as an optimization aimed at removing the cost of functions calls (especially useful for example in the case of functions called repeatedly in a loop), it has the side effect of hiding functions from the program's CG as well as increasing the size of the CFG of the caller function.

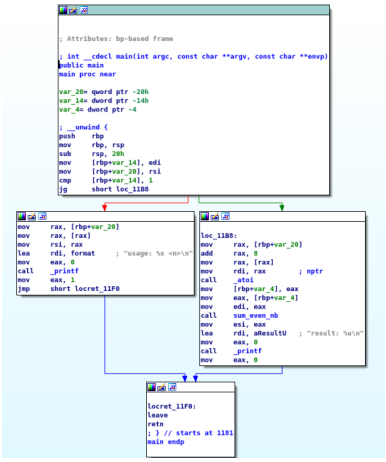
Figure 2.2 illustrates the effect of inlining on the CFG of the caller function. Here Figure 2.2b shows the original (i.e. without inlining) CFG of the function *main* (Figure 2.2a). This first CFG exclusively represent the portion of the program relative to *main* and contains only four basic blocks and no loops. Differently, Figure 2.2c illustrates the CFG of *main* after function *sum_even_nb* (Figure 2.1a) has been inlined: this second CFG contains both function *main* and *sum_even_nb*. After applying inlining the CFG of *main* results more complex, containing nine basic blocks and a loop. Moreover, by mixing together in single routine *main* and *sum_even_nb*, the semantic boundary between this two functions turns out to be much less explicit than before.

The opposite approach to function inlining is called function splitting (often referred as function outlining). In this case fragments of a target function f are replaced with a call to a function f_i serving the exact same

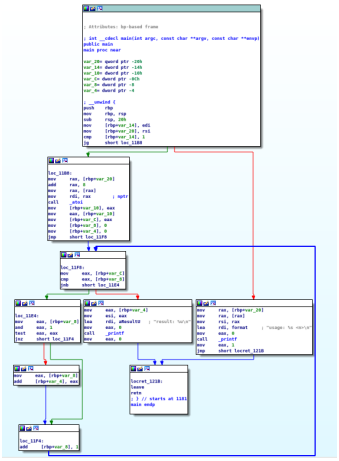
purpose as the correspondent fragment. Here, once again, the CG of the program is altered and the target function f is made harder to comprehend, given its numerous calls to mysterious sub-routines.

```c
int main(int argc, const char** argv){

  if (argc < 2){
   printf("usage: %s <n>\n", argv[0]);
   return 1;
  }

  unsigned int n = atoi(argv[1]);
  printf("result: %u\n",sum_even_nb(n));
  return 0;
}
```

(a) Function *main* calling *sum_even_nb*.



(b) CFG of *main* without inlining.



(c) CFG of *main* with inlining.

Figure 2.2: Effects of function inlining on the CFG of the caller function.

Opaque predicates: Opaque predicates are expressions which value is known at obfuscation time but is made difficult to obtain after obfuscation. Usually opaque predicates always evaluate to the same result independently of the input values and are implemented using well known mathematical identities or based on information hard to obtain without running the program [6].

Table 2.1 illustrates some examples of opaque predicates found in literature and obfuscated software. Let us take as example the first predicate in Table 2.1, this inequality will always be true no matter the values of y or x where x and y are two integers. In other words, it can be shown that the equation $7y^2 - 1 = x^2$ does not admit any discrete solution[4].

This kind of opaque predicates are traditionally used as boolean expressions in conditional branches in order to trick the reverser into analyzing non reachable parts of the program (Figure 2.3). Adding dead *junk* code by the mean of opaque branches has also the result of complicating the control flow and making the program more difficult to understand.

Another construct consist in using an opaque predicate which can evaluate to both true or false, as shown in Figure 2.4 This opaque predicate is then used as a branch condition pointing to two equivalent portions of code, which are usually obfuscated (for example using identities) to make them look different to the eyes of the reverser.

Finally another opaque construct consist in using a *Dirac* function (also known as point function). A *Dirac* function is a function which evaluates always to the same value, except for a particular (possibly hard to find) given input. Let us consider this example extracted from [22]:

[4]For an even or odd integer y we have respectively $y^2 \equiv 0 \mod 4$, or $y^2 \equiv 1 \mod 4$. This implies $7y^2 - 1 \equiv 3 \mod 4$, or $7y^2 - 1 \equiv 2 \mod 4$, meaning that $7y^2 - 1$ can not be a square.

$7y^2 - 1 \neq x^2$
$2x(x+1)$
$2 \mid x(x+1)(x+2)$
$x^2 > 0$
$7x^2 + 1 \not\equiv 0 \mod 7$
$x^2 + x + 7 \not\equiv 0 \mod 81$
$x > 0$ for $x \in I$ random where $I \subset \mathbb{N} = \mathbb{Z}_{>0}$ is a random interval

Table 2.1: Example of opaque predicates ($\mid$ indicates the bitwise OR operation).

Source: *"When Are Opaque Predicates Useful?"* L. Zobernig et al. [20]

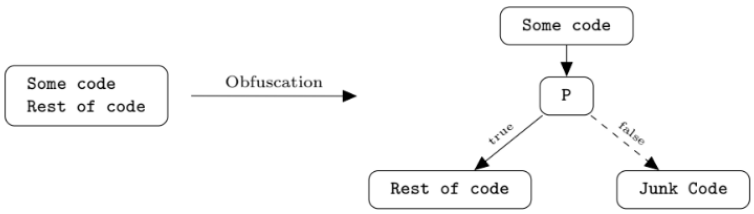


Figure 2.3: Classical opaque predicate construct with P always true.

Source: N. Eyrolles [21]

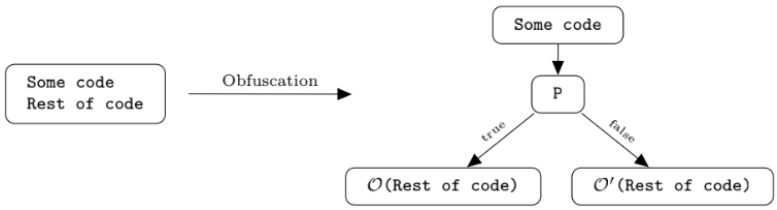


Figure 2.4: Classical opaque predicate construct with P randomly true or false.

Source: N. Eyrolles [21]

```
def f(X):
  T = ((X+1)&(~X))
  C = ((T | 0x7AFAFA697AFAFA69) & 0x80A061440A061440)\
      + ((~T & 0x10401050504) | 0x1010104)
  return C
```

This point function evaluates to `0xa061440b071544` for every input except `0x7fffffffffffff` for which instead it evaluates to `0x80a061440b071544`. This technique can be used to trick the reverser into thinking that a given branch will never be taken.

Control-Flow flattening: This obfuscation technique completely hides the control flow of a function under an additional level of indirection. The edges of the original CFG are removed and encoded somewhere-else in the program.

A typical control-flow flattening implementation consist in using a central dispatcher which decides which block to execute next depending on a state variable. Figure 2.5 shows a classic control-flow flattening implementation using a switch dispatch. Here the state variable *next* indicates which block should be executed next. The variable *next* is initialized to the first block to execute and modified by each block to make it point to the consequent block.

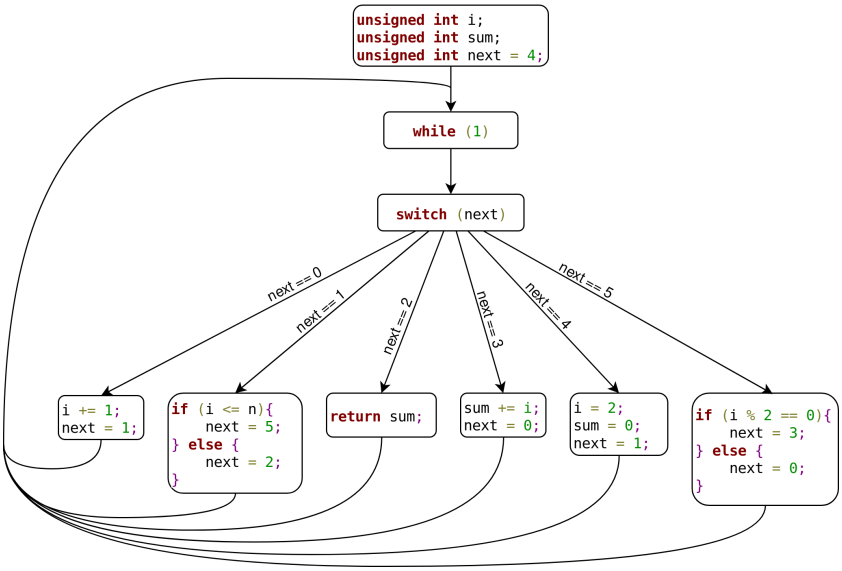


Figure 2.5: CFG of *sum_even_nb* after control-flow flattening using a switch dispatch.

The dispatch mechanism can be implemented also using *goto* statements, a jump table or even function calls. Control-flow flattening can be further complicated by using duplicate blocks, junk blocks which are never executed, multiple dispatchers, non-deterministic dispatchers or by better concealing the state variable.

2.1.2 Data-flow obfuscation

Data-flow is a broad term to indicate any stream of data processed by a program. The origin and computation steps involved in the creation of each piece of data at any point during execution may be represented using a so called Data Flow Graph (DFG). The DFG is a directed acyclic graph (DAG) where each node represent an operation and each edge a data dependency between two operations. In this sense, data-flow obfuscation techniques include all those methods which goal is to hide the data in use during computation, altering the original structure of the DFG.

Constants unfolding: Modern compilers are able to recognize, evaluate and propagate constant values known at compile time. This optimization is known as *constant folding*. On the other side the opposite process of expanding constant values to larger and more complex expressions, known as *constant unfolding*, can be used to obfuscate constants. Constants obfuscated in this way are harder to recover using classical static analysis

techniques. To further complicate things, the constants can be computed using information known only at execution time.

Encoding: To make life harder for a reverse-engineer the data used by the program can be stored using non-conventional encodings or encodings specifically engineered to make comprehension more challenging.

Data can be concealed using traditional encryption schemes or obscure unknown algorithms.

A very common example of encoding (regularly used in obfuscated malware) is *XOR-encoding*, here the data to conceal (eg. a well-known payload) is simply stored in memory XORed with a constant value [23, 24]. As simple as it is, *XOR-encoding* is often enough to bypass most anti-virus detection mechanism [25].

In general the final encoding may consist in the combination of multiple sub-encoding steps. Indeed, since different techniques can be easily piled up one on top of the other, it is not rare to encounter extremely convoluted encoding in real world software [26].

Obfuscation can be further strengthen by employing *homomorphic* encodings which make it possible to perform operations on the data without the need of first decoding it [27].

Mixed Boolean-Arithmetic expressions: Mixed Boolean-Arithmetic (MBA) expressions are expressions that mix arithmetic operators (additions, subtractions, multiplication, etc.) with bitwise operators (*AND, OR, XOR, rotation* etc.). MBA expressions are long known in the literature, especially in context of cryptography. For example, they have been used as building block to implement numerous ciphers and hash functions: such as the International Data Encryption Algorithm (IDEA), ChaCha or, for instance, constructs based on *add-rotate-xor* (ARX) networks. Their strength derives from the combination of operations from two different algebraic structures (modular arithmetic and bit-vector logic) which do not "work well together". In fact as today there exist very little work and no general theory regarding reduction and simplification of MBA expressions [28, 21].

The usage of complex MBA for obfuscation purposes was first formalized by Zhou et al. [29, 30]. In practice any expression can be transformed in an equivalent, as more complex as desired, MBA expression by iteratively applying any of the two following transformations:

- **Expressions matching and rewriting**: a portion p of the original expression is matched and replaced using a list of known rewriting rules. For example, if p is an addition $x+y$ (x and y being constants, variables or even expressions) it can be rewritten with the equivalent expression $(x \vee y)+(x \wedge y)$. Table 2.2 shows some additional examples of rewriting rules.
- **Insertion of identities**: given an invertible function f, any portion p of the original expression can be replaced with the equivalent expression $f^{-1}(f(p))$.

$x+y \rightarrow (x \vee y)+y-(\neg x \wedge y)$
$x \oplus y \rightarrow (x \vee y)-y+(\neg x \wedge y)$
$x \wedge y \rightarrow (x \vee y)+y+x$
$x \vee y \rightarrow (x \oplus y)+y-(\neg x \wedge y)$

Table 2.2: Example of MBA rewriting rules.

In other words, this method can be used to obfuscate statements in a program by replacing them with longer equivalent statements. Figure 2.6 illustrate an example of MBA expression computing `a ^ b` generated by the obfuscation tool *Tigress*.

Identities: It is often possible to express the same functionality performed with one or more instruction using a different set of instructions. For example, the instruction `call rax` can be rewritten as `push rax; ret`. The operation performed by this last is equivalent to a simple call, but longer and much less explicit than the original version.

This mechanism can be abused for obfuscation purposes to reduce readability, augment variety and make the code less obvious to understand.

A well-known example of obfuscation by identities is the *Movfuscator* [31], where every instruction is rewritten using exclusively x86_64's `mov` instructions, taking advantage of the Turing completeness of `mov`

```
(((((((a + ~ b) + 1UL) & ~ (((((a & ~ (- b - 1UL)) + (- b - 1UL)) | (((a
+ (- b - 1UL)) + 1UL) + ((- a - 1UL) | (- (- b - 1UL) - 1UL)))) + (((a &
~ (- b - 1UL)) + (- b - 1UL)) | (((a + (- b - 1UL)) + 1UL) + ((- a - 1UL)
| (- (- b - 1UL) - 1UL))))) - (((a & ~ (- b - 1UL)) + (- b - 1UL)) ^ (((a
+ (- b - 1UL)) + 1UL) + ((- a - 1UL) | (- (- b - 1UL) - 1UL)))))) << 1UL)
- (((a + ~ b) + 1UL) ^ (((((a & ~ (- b - 1UL)) + (- b - 1UL)) | (((a + (-
b - 1UL)) + 1UL) + ((- a - 1UL) | (- (- b - 1UL) - 1UL)))) + (((a & ~ (-
b - 1UL)) + (- b - 1UL)) | (((a + (- b - 1UL)) + 1UL) + ((- a - 1UL) | (-
(- b - 1UL) - 1UL))))) - (((a & ~ (- b - 1UL)) + (- b - 1UL)) ^ (((a + (-
b - 1UL)) + 1UL) + ((- a - 1UL) | (- (- b - 1UL) - 1UL)))))) ^ 2UL) -
((~ (((((a + ~ b) + 1UL) & ~ (((((a & ~ (- b - 1UL)) + (- b - 1UL)) |
(((a + (- b - 1UL)) + 1UL) + ((- a - 1UL) | (- (- b - 1UL) - 1UL)))) +
(((a & ~ (- b - 1UL)) + (- b - 1UL)) | (((a + (- b - 1UL)) + 1UL) + ((- a
- 1UL) | (- (- b - 1UL) - 1UL))))) - (((a & ~ (- b - 1UL)) + (- b - 1UL))
^ (((a + (- b - 1UL)) + 1UL) + ((- a - 1UL) | (- (- b - 1UL) - 1UL)))))
<< 1UL) - (((a + ~ b) + 1UL) ^ (((((a & ~ (- b - 1UL)) + (- b - 1UL)) |
(((a + (- b - 1UL)) + 1UL) + ((- a - 1UL) | (- (- b - 1UL) - 1UL)))) +
(((a & ~ (- b - 1UL)) + (- b - 1UL)) | (((a + (- b - 1UL)) + 1UL) + ((- a
- 1UL) | (- (- b - 1UL) - 1UL))))) - (((a & ~ (- b - 1UL)) + (- b - 1UL))
^ (((a + (- b - 1UL)) + 1UL) + ((- a - 1UL) | (- (- b - 1UL) - 1UL)))))))
& 2UL) + (~ (((((a + ~ b) + 1UL) & ~ (((((a & ~ (- b - 1UL)) + (- b -
1UL)) | (((a + (- b - 1UL)) + 1UL) + ((- a - 1UL) | (- (- b - 1UL) -
1UL)))) + (((a & ~ (- b - 1UL)) + (- b - 1UL)) | (((a + (- b - 1UL)) +
1UL) + ((- a - 1UL) | (- (- b - 1UL) - 1UL))))) - (((a & ~ (- b - 1UL)) +
(- b - 1UL)) ^ (((a + (- b - 1UL)) + 1UL) + ((- a - 1UL) | (- (- b - 1UL)
- 1UL)))))) << 1UL) - (((a + ~ b) + 1UL) ^ (((((a & ~ (- b - 1UL)) + (- b
- 1UL)) | (((a + (- b - 1UL)) + 1UL) + ((- a - 1UL) | (- (- b - 1UL) -
1UL)))) + (((a & ~ (- b - 1UL)) + (- b - 1UL)) | (((a + (- b - 1UL)) +
1UL) + ((- a - 1UL) | (- (- b - 1UL) - 1UL))))) - (((a & ~ (- b - 1UL)) +
(- b - 1UL)) ^ (((a + (- b - 1UL)) + 1UL) + ((- a - 1UL) | (- (- b - 1UL)
- 1UL)))))) & 2UL)));
```

Figure 2.6: Example of MBA expression written in C generated using *Tigress*'s *EncodeArithmetic* transformation. The expression in this example computes `a ^ b` where `a` and `b` are two 64-bits unsigned integers.

[32]. The code in Figure 2.7 shows, for example, how it possible to compare the values of two registers (*rax* and *rbx* in this case) by only using `mov` instructions. Here *rax* and *rbx* are used as memory address to store respectively 0 and 1 (in this same order). If *rax* and *rbx* both point to the same address, then the 1 written by the second instruction will overwrite the 0 written by the first instruction. The result of the comparison is stored in the address pointed by *rax* and is finally copied in the register *rax*.

```
mov [rax], 0
mov [rbx], 1
mov rax, [rax]
```

Figure 2.7: Example of mov-based obfuscation of registers comparison. Here registers *rax* and *rbx* are compared. The result of the comparison is then stored in *rax* (1 if the registers are equal 0 otherwise). As side effect, the comparison in this example overwrites all content initially pointed by *rax* and *rbx*.

2.1.3 File format based

Numerous reverse-engineering tools (e.g. disassemblers) rely on file format information: headers, symbols, sections, etc. generally assume their correctness. There exist several hacks taking advantage of this assumption to break some kind of automated analysis or misled and slow down the reverser giving erroneous information. We briefly cite them here for completeness in spite of not being closely related to our work.

File format based obfuscation techniques can be divided in two categories:

- **Information removal**: suppression of information non strictly necessary to run the program.
- **Information rewriting**: rewriting of any file-format information without changing the program behaviour.

Taking as example the ELF file format. Tools such as the *strip* utility (used to remove symbols from object files) as well as other "cleaning" techniques such as the removal of the *section header table* fall under the first category.

The second category includes all sort of alterations. In practice any "removable" part of the program can be also carefully modified to mislead analysis. For instance, J. Baines in [33] illustrates how flipping the executable bit in the section header or inserting a fake entry point, is enough to stop most disassemblers from analyzing portions of the program. Furthermore a piece of information does not have to be strictly "removable" in order to be modified, but is also possible to modify parts of the binary that are actually used during execution and still come up with a perfectly working program. D. Barry [34] shows, for example, how it is possible to make different ELF data structures overlap with one another.

2.1.4 Hybrid techniques

In practice the techniques that we presented in the previous sections are often combined to create even more robust obfuscation techniques [6]. Here we present some of the most popular ones.

Virtualization: This obfuscation techniques turns a piece of code in an interpreter of a custom *virtual instruction set*. The original instructions (or statements) are translated and stored as bytecode performing the exact same functionality. The result obfuscated program embeds both the interpreter and a bytecode-encoded version of the original code. Upon execution the interpreter is used to fetch, decode and execute the bytecode instructions using the appropriate instruction handler.

Depending of the implementation the VM may make use of a *virtual stack* and and/or a number of *virtual registers*. Figure 2.8 illustrates Tigress [3] implementation of the *Virtualize* transformation consisting in a virtual stack machine.

Virtualization completely hides the control-flow and data-flow of the original program under an additional level of abstraction. To further strengthen obfuscation it is possible to use multiple nested levels of virtualization.

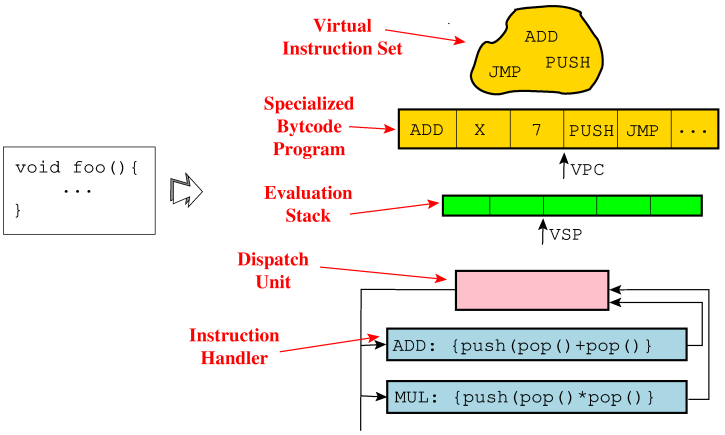


Figure 2.8: Virtualization-based obfuscation (Tigress implementation).

Source: Tigress - *"Function Virtualization"* [3]

Jitting: Just In Time (JIT) compilation (often referred as *jitting*) is a widely used technique to optimize execution of interpreted code. At run-time, the interpreter constantly evaluates portions of code and decides whether compiling it to native code will likely result in an increase in performances. If this is the case (e.g. when the code under analysis is inside a loop with a very high number of estimated iterations) the interpreter invokes the JIT compiler and stores a faster, native version of the code.

The same concept of JIT code generation can be used to obfuscate parts of a program by forcing them to be generated at run-time [35]. JIT-compilers can also introduce differences among distinct compilations of the same code, thus making more challenging for a reverser to pinpoint and analyze the code.

2.2 Software analysis and deobfuscation techniques

While great progresses have been made in the software obfuscation domain, more and more sophisticated techniques have been created to counter such protections, in a never ending cat-and-mouse game.

In this section we illustrate some well-known software analysis techniques broadly used in the field of reverse engineering. Since exploring all known analysis techniques would be unfeasible, we limit this section to a set of approaches which understanding is essential for our work.

2.2.1 Program tracing

In reverse engineering, program tracing is a dynamic analysis technique consisting in the observation and collection of information regarding the execution of a program. Tracing can be limited to a small set of properties, such as function calls, as well as applied to the whole program state (i.e. all executed instructions, registers values and memory content).

Tracing is useful for debugging purposes and has found many applications in the reverse engineering and program comprehension domain, where it is often coupled with other analysis techniques such as *symbolic execution*. Different approaches can be used to monitor program execution in order to implement tracing:

- **Debugging**

  Software monitoring and tracking can be implemented taking advantage of classical debugging API offered by the underlying system (e.g. the kernel). An example of such API is the *ptrace* [36] system call offered by the Linux kernel. This API can be used to break the execution of the program (*debuggee*) at any point as well as intercept events such as system calls, memory reads, memory write, etc. and access register and memory values.

  In practice debugging involves a continuous exchange of information between the debuggee, the underlying system and the debugger. This process is particularly inefficient when debugging userland programs

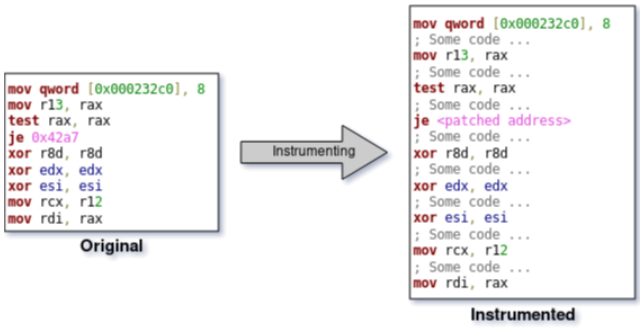


Figure 2.9: Dynamic program instrumentation at single-instruction granularity.

under an operative system, since it involves several context switches (between the kernel and the debuggee, and the kernel and the debugger) and requires the debuggee and the debugger to be preempted and rescheduled by the kernel at every context switch.

- **Instrumentation**

  A better performing technique consist in inserting additional intermediate code responsible for collecting run-time information, to the program under analysis (Figure 2.9). The so added code will have complete access to all of program's memory and registers values, since it constitute part of the actual program. This process of inserting additional monitoring code to a program is called instrumentation.

  The reason why program instrumentation performs better than debugging is the absence of costly context switches between the kernel and userland programs. Formally speaking, an instrumented program play both the roles of debugger and debuggee and does not necessarily interact with other programs but himself, thus introducing a much smaller overhead.

  Instrumentation can be done statically (i.e. before executing the program) or dynamically (i.e. at runtime). Static instrumentation is typically easier to implement but can not be applied to self-modifying programs such as those employing code morphing based obfuscation techniques (e.g. JIT).

Some examples of well-known tracing software are *strace* [37], a program taking advantage of Linux's *ptrace* API to trace system calls; *ltrace* [38], an utility using dynamically inserted Shims to trace calls to external libraries.

There exist also a number of *Dynamic Binary Instrumentation* (DBI) frameworks allowing any user to dynamically instrument programs with custom code.

2.2.2 Symbolic Execution

Symbolic Execution (SE) is a software analysis technique used to reconstruct the data flow of a program. Symbolic values are used as input parameters instead of the concrete values normally used during execution. During SE program inputs are replaced by symbolic values, Symbolic variables are then propagated according to the logic of the instructions executed and the content of the program's variable is translated into so-called *symbolic expressions*.

Figure 2.10 illustrates how symbolic values are propagated during SE and corresponding symbolic expressions are assigned to variables. Prior to SE the variables a and b are initialized (i.e. symbolized) to the symbolic values A and B. During SE all operations on symbolic values A and B are tracked, as well as any transfer/usage of a symbolic values from a variable to another. Let us take consider for example the state of variables a and b immediately after the execution of line 1. While variable b is unmodified, to take into account the operation in line 1 the state of variable a has been updated to $A + B$. Finally, we can observe how the symbolic expression in variable a immediately after line 3, indicates that variable a's value (at that point during execution) does not depend on b. This little example shows how SE can be used as an automated method to gain insights on the executed code.

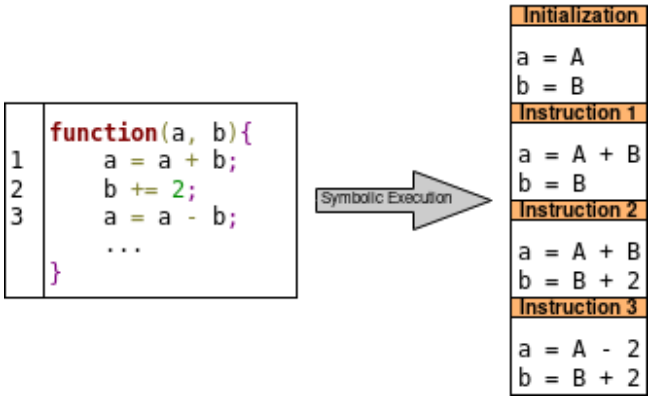


Figure 2.10: Example of Symbolic Execution (capital *A* and *B* indicate the symbolic input values of variables *a* and *b*).

The analyst can be take advantage of the obtained symbolic expression to constraint the set of values a variable can take at a certain point during execution, based on the inputs of the program. SE has a large number of applications in fields such as software testing, automated reverse engineering and exploitation of software vulnerabilities.

Chapter 3

Related Work

In this chapter we present the relevant literature, related to our research goal.

3.1 Program Synthesis

Program synthesis consist in automatically deriving a program from a given high-level specification. In the context of reverse engineering program synthesis approaches usually involve black-box oracle access to the obfuscated program or a set of input/outputs pairs to be provided. Inputs/Outputs (I/O) information about the program under analysis are then used to inductively synthesize a candidate program with same I/O behaviour.

Contrarily from other deobfuscation techniques, the performance of synthesis approaches solely based on I/O behaviour and is not influenced by the inner complexity of the obfuscated program but only by its semantic complexity.

In our work we use a *divide and conquer* strategy involving the use of program synthesis to deobfuscate small portions of the original obfuscated program, extracted using dynamic symbolic execution. For this reason deobfuscation techniques based on program synthesis are complementary to our work.

There exist numerous and various approaches to synthesis. In this section we introduce a selection of synthesis approaches which are of interest to our work.

Syntia: In their work, Blazytko et al. [17] propose a synthesis-based deobfuscation tool called *Syntia*. Their tool synthesizes obfuscated programs by heuristically searching, on a pre-defined grammar, programs with equivalent I/O behaviour for a given set of I/O pairs.

For the search they use a Monte Carlo Tree Search (MCTS) implementation combined with *simulated annealing* to maximize the chances to escape local maxima and find the global optimal solution. MCTS is a probabilistic search algorithm used to efficiently search large spaces, such the space of possible chess games given a chessboard configuration, and it is driven by some metrics estimating the quality of each node in the tree (e.g. each chessboard configuration).

In *Syntia* MCTS is used to explore the space of all possible programs that can be generated using a given grammar. Their MCTS is guided by similarity metrics estimating the degree of correctness of each synthesized expression. This similarity metrics include for example the Hamming distance between the output generated by the synthesized expression and the original output, and the difference in number of leading/trailing zeros and/or ones (which estimates whether two given values are on the same numerical range).

Blazytko et al. apply their synthesis algorithm to the instructions contained in an execution trace (which they obtain using *Unicorn* [39], a CPU emulator). The trace is then dissected in several trace windows using some heuristics. Each trace window is then synthesized separately. After synthesis the obtained expression is simplified using Z3's *simplify* function [40].

In general *Syntia* has demonstrated to be able to deobfuscate expression obfuscated using MBA, virtualization and programs based on Return-Oriented Programming (ROP) [41].

The approach of Blazytko et al. is of particular interest for our work in that, similarly to our approach, it combines I/O based inductive program synthesis with program tracing and some form of trace analysis. While *Syntia* uses some heuristics to determine which trace windows should be synthesized separately, we use symbolic execution to build an AST representation of the program and iteratively reduce it to a simpler form using program synthesis. Differently from our approach, the trace portioning done by *Syntia* could end up chopping the obfuscated program trace in the middle of an operation of the original version of the program, thus producing trace windows which are not semantically meaningful.

Drill and Join: Biondi et al. [42, 43] propose a deobfuscation approach based on the *drill and join* method for inductive program synthesis first introduced by Balaniuk [44] to deobfuscate obfuscated conditionals using black-box synthesis. The general idea behind the *drill and join* method is to use a *divide and conquer* approach to synthesize each bit-component of the target function independently. Each bit produced by the obfuscated vectorial Boolean function F is considered as a separate Boolean function f_i. The function f_i is then iteratively decomposed in its own sub-spaces until the basis have been found. Finally f_i is recombined using its basis and, all bit-components are merged together to encode F.

Their approach is of interest to our research since it has shown to be effective in deobfuscating MBA-obfuscated expressions, one of the obfuscation techniques which we target in this work. However the complexity of the approach of Biondi et al. grows exponentially with the size in bits of the input, which make it impractical for deobfuscating expressions using multiple input variables. The reported time needed to synthesize an MBA expression of 96 input bits is around 20 seconds.

Component-Based Synthesis: Jha et al. approach [45] consists in synthesizing loop-free programs only based on their I/O behaviour by transforming the synthesis problem into a satisfiability problem and employing an SMT solver to obtain a suitable program candidate with correspondent behaviour. This is done by selecting a list of base operations, called *components*, and encoding in a formula the space of all possible programs which can be obtained using the given *components*. The final formula encodes well-formedness and behavioural constraints, respectively ensuring the syntactic correctness of the generated program and its equivalence to the target program in terms of input/output behaviour. An SMT solver is used to solve the given constraints and generate a candidate program which can in turn be checked using a validation oracle. In case the generated program semantic did not correspond to the desired one, new input/output pairs are generated to further constraint the formula.

Jha et al. implemented this approach in a tool called *Brahama* and tested it on a dataset of 22 bit-manipulating programs extracted from the book *Hacker's Delight* [46] and 3 obfuscated programs. *Brahama* has demonstrated effective in synthesizing the given set of programs finding a suitable candidate for almost all samples within an average execution time of 31 seconds.

The work of Jha et al. is particularly focused on automatic generation of programs performing non-obvious bit manipulations, often difficult to write for a human. For this reason their proposed approach is solely driven by the I/O behaviour of a program and, contrarily to our approach, it does not exploit any kind of information extracted from the obfuscated code.

However, despite the design differences, their approach is of interest for our work in that it can be used to deobfuscate binaries, and it shares some similar problematic to our work when it comes to the selection of base *components* for synthesis. The question of *components* selection of Jha et al. is comparable to that of grammar selection for our lookup-tables based synthesis primitive discussed in Sections 4.2.1 and 5.3. In their work Jha et al. relegate to the user the selection of the base *components* for synthesis, which should be chosen accordingly to the application domain.

While the approach of Jha et al. has shown promising results for automatic generation of unintuitive bit-manipulating programs (especially useful for engineering integrated circuits), its effectiveness as deobfuscation technique should be at least further investigated.

Superoptimization: Superoptimization is the process of finding, given an instructions set, the shortest program to compute a function f. The concept of superoptimization was first introduced by H. Massalin [47] which

proposed a superoptimizer based on exhaustive search over a selected subset of the machine's instruction set. To reduce search time they propose two optimization methods. The first method consists in probabilistically testing each generated program on a carefully chosen set of inputs instead of rigorously testing its equivalency to the program to optimize using a so-called *boolean program verifier*. The second optimization method consist in skipping redundant instructions patterns during the search. The method proposed by Massalin guarantees the optimal result given a sound order of search.

The GNU Superoptimizer (GSO) introduced by Granlund et al. [48] improves on Massalin approach by including a set of optimization aimed at reducing the search space. In GSO for example the choice of instructions' operands is narrowed to the outputs generated by previous instructions and, in order to avoid redundant searches, only one ordering for commutative operations is used.

Schkufza et al. [49] introduced *STOKE*, a superoptimizer based on Monte Carlo Markov Chain method, capable of exploring the space of possible programs faster than previous approaches, but without guarantees of optimality.

Differently, a feasibility study on superoptimization by Embecosm [50] proposes a *constructive* superoptimizer design, based on dataflow DAG simplifications using an SMT solver.

While the original use-case of superoptimization consisted in optimizing compiled code (e.g. by finding peephole optimizations missed by the compiler), there is a big overlap between the research conducted on this field and the field of program synthesis and synthesis-driven deobfuscation. For this reason superoptimization is relevant to our study.

Furthermore, work on superoptimization whose objective is to optimize the exploration time of the space of potential programs, is relevant to our proposal in that it can be applied to improve lookup-tables generation time and grammar selection (Section 5.3).

3.2 Expression simplification

In this section we illustrate some approaches tackling the deobfuscation problem from an expression simplification perspective. The works presented in this section are particularly targeted toward the deobfuscation of MBA expressions, which also represent one of the goals of our approach.

Differently from synthesis-based deobfuscation, approaches based on simplification benefit from white-box access to the operations performed by the obfuscated program, and usually work on an algebraic representation of the expression to simplify. This representation is similar to that used by our approach (Section 4.1), with the only difference that it does not admit usage of program's constructs other than boolean and arithmetic operators.

We consider expression simplification approaches complementary to our work, in that they can be used to simplify parts of the obfuscated program, prior to deobfuscation, or to simplify parts of the result yield by our approach, after deobfuscation. The use of expression simplification before deobfuscation may facilitate synthesis and speed up the execution of the algorithm presented in Section 4.3. Similarly, the use of expression simplification on the deobfuscated expression may improve the readability of the result by cleaning out unnecessary operations. This last application of expression simplification has been, for example, adopted by Blazytko et al. [17] to improve the quality of the expressions synthesized by their tool *Syntia* [1].

Algebraic simplification: Biondi et al. [43] propose an algebraic simplification approach to polynomial MBA expressions. Their approach is aimed at reducing the complexity of polynomial MBAs to MBAs of degree one, in order to ease deobfuscation or satisfiability check. The technique proposed by Biondi et al. works however only on a certain subset of MBA expression following a specific construct.

SSPAM: Eyrolles et al. [16] propose an expression simplification tool named *SSPAM* based on pattern matching and rewriting. *SSPAM* works similarly to Z3 [40] *simplify* function but it is particularly targeted on simplification of MBA expressions and is based on the intuition that the same rewriting rules used to obfuscate

[1] Blazytko et al. use the expression simplification API *simplify* offered by the tool Z3 [40].

programs with MBA can be used to simplify their correspondent MBA-obfuscated expressions. Usually the rules used for simplification correspond to the inverse of those used for obfuscation, however this is not always the case.

The workflow of *SSPAM* consist of two steps which are applied iteratively until a given fixed point has been reached (e.g. the tool runs out of rewriting rules). At first a terms rewriting step is performed where *SSPAM* uses one of the known rewriting rules to simplify the expression. A second step consist in simplifying arithmetically the obtained expression using a computer algebra system.

SSPAM has demonstrated effective against MBA obfuscation, being able to reduce to 50% the number of nodes in the obfuscated expression terms graph. However, as highlighted by the authors the effectiveness of *SSPAM* highly depends on the set of substitution rules used for the rewriting step.

To our knowledge the approach proposed by Eyrolles et al. has not yet been applied to compiled binaries, therefore additional work is needed in this direction to asses whether it is possible to effectively reconstruct from compiled code the original MBA expression generated by the obfuscator (e.g. taking into account eventual compiler transformations/optimizations).

Bit-blasting: As discussed in Section 2.1.2 there does not exist as today a general theory for reduction and simplification of MBA expressions. This fact has pushed researchers into trying to bring the MBA simplification problem into a well-studied domain: boolean algebra.

This approach, also known as "bit-blasting", consists in handling all obfuscated expression's operations using bit-vector logic. A boolean variable is created for every bit of the expression, representing the constraints of the expression for that particular bit. Simplification is then done bit-by-bit by applying well-known boolean identities.

Despite several software, such as various SMT solvers or computer algebra systems, implementing bit-vector logic, most of them are only focused on SAT solving or do not encode arithmetic operations.

At the time of writing, *Arybo*, the tool presented by Guinet et al. [15], is to the best of our knowledge the only simplification-focused software supporting bit-vector boolean and arithmetic operations (i.e. able to handle MBA expressions). *Arybo* works by constructing a bit-level symbolic representation of a given expression where each bit is canonicalized using the Algebraic Normal Form (ANF). The work of [15] shows that simplification based on bit-blasting is quite effective on expressions with a low number of bits.

3.3 Symbolic Execution

In our approach we rely on Symbolic Execution (SE) to build an abstract-syntax-tree (AST) representation of a portion of interest of the obfuscated program (Sections 4.1 and 5.2). For this reason, work on SE and, in particular, on the application of SE to deobfuscation of binaries, is strongly related to the work object of this thesis (it is important to precise however that, differently from most deobfuscation approaches using SE, we do not use SMT solving in the context of our work).

Triton: *Triton* [11] is a Dynamic Binary Analysis (DBA) framework featuring a concolic execution engine with dynamic taint analysis capabilities. It works by emulating the semantic of each CPU instruction while simultaneously maintaining a symbolic representation of the program as well as a concrete state of memory and register values, used as default in case of missing symbolic representation. In practice, *Triton* is very effective in following tainted and symbolic values during emulated execution.

Salwan et al. [51] have demonstrated how *Triton* can be employed to completely deobfuscate programs which have been obfuscated using virtualization-based software protections. Their approach consist in the use of taint analysis to distinguish between the instruction of the original non-obfuscated program and those belonging to the VM.

In our proposed approach we use *Triton* (through its Python3 API) to build an AST representation of part of the obfuscated program (Section 5.2).

Backward-bounded DSE: As shown by Kruegel et al. [52] disassembling obfuscated binaries is an arduous task. Several obfuscation methods introduce artifacts which make it difficult to determine which code will be actually executed by the binary (e.g. opaque predicates, dead code, self-modifying code, etc.).

Bardin et al. [53] proposed a method based on Backward Dynamic Symbolic Execution (BDSE) to "clean" obfuscated binaries from unreachable code. Their method consists in first recovering a partial CFG using dynamic analysis, and then use the partial CFG to asses whether a given predicate is satisfiable based on the constraints obtained backward reasoning on the binary code up to a certain level of depth.

We believe that methods such as the one proposed by Bardin et al., could be used in combination to our approach in order to deobfuscate larger portions of a program which are not bind to a particular execution. In Section 7.1 we discuss more in details about this potential application of Bardin et al. work.

Chapter 4

Our approach

By considering the obfuscated program as a black-box, synthesis-based approaches such as *Drill and Join* or *Component-Based Synthesis* (Section 3.1) assume the best case theoretical obfuscation scenario. Employing the black-box assumption for deobfuscation purposes is very handy since, by design, it can be generalized to virtually any program. However, by providing only oracle access to the obfuscated program, black-box based approaches try to solve a harder problem than practical deobfuscation. Not only instances of perfect black-box obfuscation have been demonstrated to be impossible, but even weaker theoretical forms of obfuscation are currently far from being applicable to real world software [18].

We believe that better results may be reached by combining inductive program synthesis with other "white-box" approaches, thus taking advantage of additional information extracted directly from the obfuscated binary. In other words, we are confident that removing the black-box assumption may speed-up synthesis-based deobfuscation.

In this chapter, we illustrate our work on the topic, suggesting a "hybrid" deobfuscation approach based on Abstract Syntax Tree (AST) simplification, mixing program synthesis and symbolic execution.

Figure 4.1 shows an overview of the steps involved in our approach. We start by executing symbolically (part of) the obfuscated program under analysis. Subsequent to symbolic execution we obtain a Dynamic Abstract Syntax Tree (Dynamic AST), which is a representation of the operations performed by the program in the form of an AST. We then proceed to simplify the Dynamic AST. For our simplification routine we employ one or more lookup-tables. Lookup-tables can be generated directly by the user or obtained from a third-party, and can be reused among different deobfuscation tasks.

In Section 4.1 we introduce the notion of Dynamic AST. In Section 4.2 we present the concept of synthesis primitives and illustrate a synthesis primitive based on pre-computed lookup-tables. Finally in Section 4.3 we propose a deobfuscation algorithm based on the combination of program synthesis and symbolic execution.

4.1 Dynamic Abstract Syntax Tree

An Abstract Syntax Tree (AST) is a representation in the form of a tree of a program. Each AST node may contain a variable, a constant, an operator or any other statement which is specified in the program's grammar. Abstract Syntax Trees are, for example, widely employed during the compilation process as an intermediate representation, linking the source code written by the programmer and the binary code generated by the compiler.

In practice any program can be represented in the form of an AST. Let us consider for example Figure 4.2, here the AST on the right hand side represents all operations involved in the computation value **r** just after the execution of the subtraction operation at line 6. This tree contains a conditional statement. If we would however consider only a single execution instance, for example one where **a = 5** (and **r = 2**), then we could remove all conditional statement and represent only the portion of the AST that was actually executed, as shown in Figure 4.3. This last AST version does not contain any control-flow information, but solely describes the data-flow generating variable **r**. We name this typology of AST a *Dynamic AST*, since it is obtained "dynamically" by executing the target program on a given input.

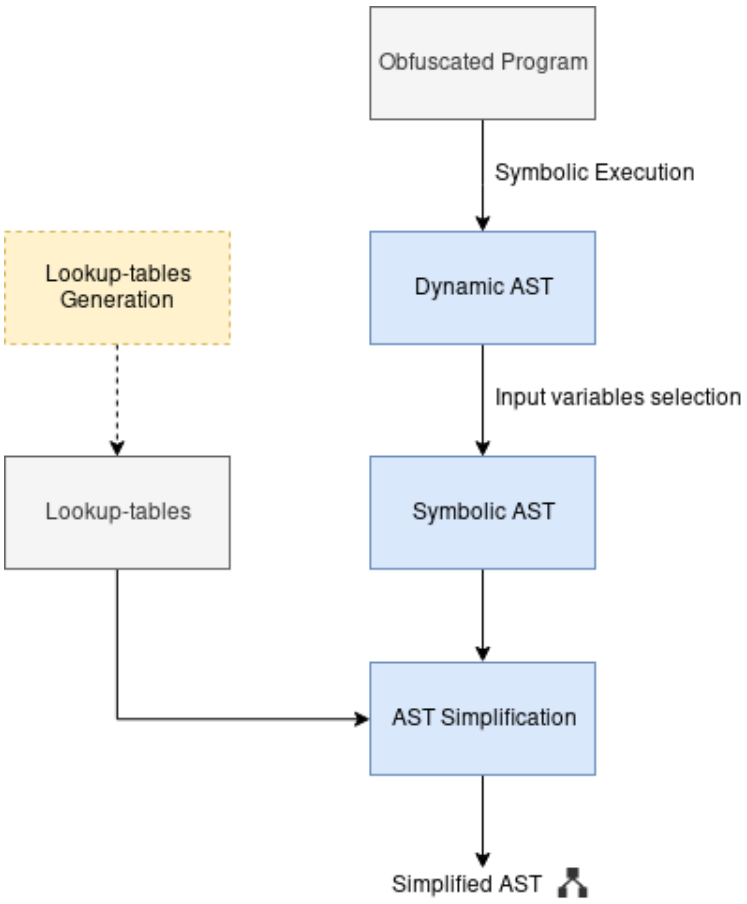


Figure 4.1: Approach overview (dashed steps are optional).

In our work, we focus on Dynamic ASTs of a single target value containing only operations occurred during a given execution e and excluding control-flow information. We define an execution e as an ordered sequence of instructions $\{h_0, h_1, h_2, \dots\}$[1]. Let r indicate a CPU register or memory location and $h_t, h_q \in e$ be two instructions such that $t < q$. We denote $A_e(r, h_t, h_q)$ the Dynamic AST relative to execution e, representing all data and operations directly involved in the computation of the value of r immediately after instruction h_q, starting from instruction h_t.

4.1.1 Symbolic Abstract Syntax Tree

Typically a Dynamic AST $A_e(r, h_t, h_q)$ would contain the concrete values used at run-time to compute the value of r. In the case of the Dynamic AST in Figure 4.3, for example, variable **a** has been substituted by the concrete value 5 and the initial value of **r** has been set to two. We can invert this process and transform any selection of concrete values of the Dynamic AST in symbolic variables. Any symbolic variable can later be evaluated to a concrete value of choice.

Given a Dynamic AST $\alpha = A_e(r, h_t, h_q)$ and given τ a non-empty set of leaves nodes of α corresponding to some given concrete values, we denote a Symbolic AST $\sigma = SA_\alpha(\tau)$ a modified version of α where each concrete values in τ has been replaced by a symbolic variable of the same size. Here σ encodes a symbolic expression similar to those discussed in Section 2.2.2.

Given a Symbolic AST σ, we can evaluate it by assigning to each symbolic variable a concrete value. To this end we denote M_σ the set of mapping functions of σ, taking as input any symbolic variable s_j and returning a concrete integer value of the same size.

[1]For simplicity we do not take into account concurrent programs. However the same reasoning can be easily applied to multi-threaded programs by taking into account topological ordering.

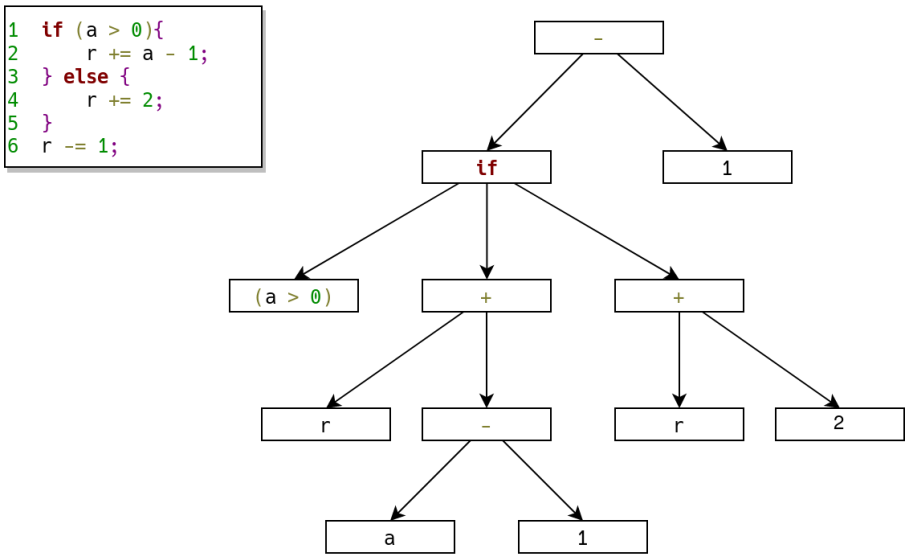


Figure 4.2: Example of Abstract Syntax Tree. The AST represented here describes the operations used to compute the value of variable **r** after the execution of line 6.

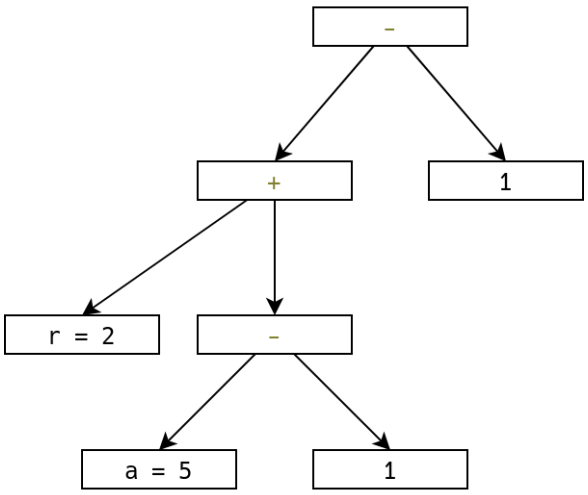


Figure 4.3: Example of Dynamic Abstract Syntax Tree. As in Figure 4.2 this AST describes the operations used to compute the value of variable **r**. Here however only an execution where **r** = 2 and **a** = 5 is taken into account.

Finally, we call $\mathcal{O}_\sigma$ the oracle function of the Symbolic AST σ, such that $\mathcal{O}_\sigma(m)$ for $m \in M_\sigma$, is the concrete value computed by the AST after that each symbolic variable has been concretized according to the mapping function m.

4.2 Synthesis primitives

We use the term *synthesis primitive* for any program *SP* taking as input parameters a black-box oracle $\mathcal{O}_\sigma$, as defined in Section 4.1, together with a finite set of inputs values $I_\sigma = \{m_1, m_2, \ldots, m_n\}$ where $m_i \in M_\sigma$, and returning as output either a program p such that for any $m_i \in I_\sigma$ then $p(m_i) = \mathcal{O}_\sigma(m_i)$, or an error code indicating that *SP* was not able to find such program. More informally, we call synthesis primitive any algorithm which can be used to synthesize a black-box program on a finite set of inputs. In this sense, a synthesis primitive is not sound and may produce a program which behaves differently than the original target, except for the given set of inputs I_σ. It should be possible however to tune synthesis precision by providing a larger set of inputs.

In practice any of the black-box synthesis techniques presented in Section 3.1 could be used as synthesis primitive for our approach. For instance, an example of synthesis primitive is the Monte Carlo Tree Search proposed by Blazytko et al. [17] as part of the synthesis-based deobfuscation tool *Syntia*. For our work, however, we decided to employ a different synthesis approach than those illustrated in Section 3.1. We present our synthesis primitive in Section 4.2.1.

4.2.1 Lookup-tables based synthesis primitive

In the context of our work we employ a synthesis primitive based on pre-computed lookup-tables. For our synthesis approach we generate one or multiple lookup tables (LUT) l which, for a given fixed vector of inputs $I_l = \{i_0, i_1, \ldots, i_n\}$, associate a set of outputs values $O = \{o_0, o_1, \ldots, o_n\}$ to a program p producing that same set of outputs. In other words, our LUT l will take as input a set of values O and, in case of success, return a program p such that for every $o_j \in O$ then $p(i_j) = o_j$.

To build a LUT we first define a fixed set of inputs I_l where each $i \in I_l$ is a set of k values. We then generate the programs contained in our LUT using a context-free grammar $G = \{N, \Sigma, R, S\}$ where N is the set of non terminal symbols, Σ is the set of terminal symbols, R is the set of production rules and S the set of starting symbols. We define our grammar G such that the set of terminal symbols Σ contains any number of constant values as well as k input variables of variable sizes.

Table 4.1 illustrates a minimal example of grammar, containing all terminal symbols and derivation rules necessary to construct the program represented by the Dynamic AST in Figure 4.3 (considering **a** and **r** as 8-bits variables). The necessary derivation steps used to obtain the program in Figure 4.3 from the grammar in Table 4.1 are shown in Table 4.2.

N	$=$	$\{u_8\}$
Σ	$=$	$\{a_8, r_8, 1\}$
R	$=$	$\{u_8 \rightarrow u_8 + u_8,\ u_8 \rightarrow u_8 - u_8,$ $u_8 \rightarrow a_8,\ u_8 \rightarrow r_8,\ u_8 \rightarrow 1\}$
S	$=$	$\{u_8\}$

Table 4.1: Example of grammar G with a non-terminal symbol u_8 (unsigned integer of 8 bits), two operations modulo 2^8 ($+$ and $-$) and two terminal variables of 8-bits a_8 and r_8 (i.e. $k = 2$).

Following, we exhaustively generate all terminal (i.e. not containing any non-terminal symbol) expressions p in G up to an *a priori* defined number of derivations. We then evaluate each expression for all $i \in I_l$ by assigning each value in i to one and only one variable defined in Σ. The result is an ordered set of outputs $O = \{p(i_j) \mid \forall i_j \in I_l\} = \{o_1, o_2, \ldots, o_n\}$. Finally, we add a new entry O to the LUT l such that $l(O) = p$. In case there exist another entry O in l corresponding to a different expression, we keep the old entry and omit

Derivation rule	Expression
	u_8
$u_8 \rightarrow u_8 - u_8$	$u_8 - u_8$
$u_8 \rightarrow u_8 + u_8$	$(u_8 + u_8) - u_8$
$u_8 \rightarrow u_8 - u_8$	$(u_8 + (u_8 - u_8)) - u_8$
$u_8 \rightarrow r_8$	$(r_8 + (u_8 - u_8)) - u_8$
$u_8 \rightarrow a_8$	$(r_8 + (a_8 - u_8)) - u_8$
$u_8 \rightarrow 1$	$(r_8 + (a_8 - 1)) - u_8$
$u_8 \rightarrow 1$	$(r_8 + (a_8 - 1)) - 1$

Table 4.2: Derivation steps used to obtain the program in Figure 4.3 using the grammar in Table 4.1.

the new one. Provided that the exploration order used to generate the expressions is sound, synthesis by LUT guarantees the optimal solution since only the first expression corresponding to the given output is hold in the LUT.

Table 4.3 shows an example of LUT generated from the grammar in Table 4.1. Here each $i \in I_l$ contains two values (i.e. $k = 2$), the first one is always mapped to variable a_8 while the second is mapped to r_8.

LUTs generation can be an expensive operation, especially for large grammar and if a considerable number of derivations is needed. For this reason, when working with a very large base grammar, we suggest generating multiple smaller LUTs using random subsets of the original grammar rather than a comprehensive big LUT using directly the original grammar. Furthermore LUT generation can be easily parallelized and/or can be sped up using memoization.

4.3 AST simplification

In Sections 4.1 and 4.2 we have introduced the concepts of respectively Symbolic AST and synthesis primitive. In this section we propose an AST simplification procedure employing a synthesis primitive *SP*.

Our deobfuscation approach consists in first representing a target program in the form of a Symbolic AST (which we obtain using symbolic execution) and then simplify this last using program synthesis. Indeed the key intuition behind our approach is that, by using symbolic execution to build a structured and easy to manipulate representation of a program (such as an AST), we can tackle separately simpler part of the obfuscated program rather than the whole program at once.

Furthermore, this approach is superior if compared to other *divide and conquer* approaches based on the synthesis of separate chunks of the binary (extracted randomly or using some heuristics), because, contrarily from the said approaches, by exploiting the knowledge of the instruction's semantic provided by a symbolic execution engine we can always guarantee a "meaningful" partitioning of the program.

Algorithm 1 illustrates our simplification approach, this can also be visualized in Figure 4.4. Our algorithm consist in continuously reducing the size of the AST until the whole AST has been synthesized. Our simplification routine (line 1) takes as input the Symbolic AST σ to simplify and a synthesis primitive *SP*. It then proceeds to create an hash map which will be used to keep track of the synthesized portions of the AST.

To simplify the AST, at every iteration of the loop we perform a synthesis step: we traverse all its sub-trees using randomized breadth-first order (lines 27,28) until we find a sub-tree which we are able to synthesize (line 29-32). Our randomized breadth-first search works by processing all nodes of a layer of the tree in a random order before descending to the successive layer (i.e. none of the nodes of a lower layer are processed before all nodes in the upper layer have been processed).

We use a randomized breadth-first order to introduce diversity among different executions[2].

Every time we successfully synthesize a sub-tree, we build the AST correspondent to our synthesized program (line 9) and replace the synthesized portion of the AST with a new symbolic variable (lines 10,11).

[2]The success of our simplification routine also depends on the order at which the nodes are processed, therefore by introducing order diversity we hope that a program which was not successfully simplified at first may be simplified in a later execution using different nodes order.

$\boldsymbol{I}_l = \{(42,10),\ (6,2),\ (75,231)\}$				
Derivations	$\boldsymbol{O}$			$\boldsymbol{p}$
1	42,	6,	75	a_8
1	10,	2,	231	r_8
1	1,	1,	1	1
3	84,	12,	150	$a_8 + a_8$
3	52,	8,	51	$a_8 + r_8$
3	43,	7,	76	$a_8 + 1$
3	0,	0,	0	$a_8 - a_8$
3	32,	4,	99	$a_8 - r_8$
3	41,	5,	74	$a_8 - 1$
3	52,	8,	51	$r_8 + a_8$
3	20,	4,	207	$r_8 + r_8$
3	11,	3,	232	$r_8 + 1$
3	223,	251,	156	$r_8 - a_8$
3	0,	0,	0	$r_8 - r_8$
3	9,	1,	230	$r_8 - 1$
3	43,	7,	76	$1 + a_8$
3	11,	3,	232	$1 + r_8$
3	2,	2,	2	$1 + 1$
3	214,	250,	181	$1 - a_8$
3	246,	254,	25	$1 - r_8$
3	0,	0,	0	$1 - 1$

Table 4.3: Example of LUT generated using grammar G in Table 4.1 up to a maximum of three derivations (notice that the table does not contain any expression of only two derivations, this is because it is not possible to create a terminal expression with two derivations using the given grammar). Expressions are generated from top to bottom, grayed expressions are ignored.

The association between symbolic variable and synthesized program is stored in the hash-table (line 12). This simplification loop (lines 3-13) continues until the whole AST has been synthesized.

Finally we unroll the AST by recursively substituting all symbolic variables in the hash map with the corresponding AST (lines 14, 16-24).

It is important to notice that, in case the synthesis primitive *SP* was not able to synthesize any of the subtrees of the AST during a synthesis step, the AST simplification routine will not complete successfully and will instead return an error. In other words, Algorithm 1 is not guaranteed to always converge.

The number of steps needed for synthesis is directly proportional the size of the AST, and every synthesis step of our algorithm traverses the AST from top to bottom until a suitable node has been found.

Therefore the worst case time-complexity of our algorithm is $O(n^2q)$ where n is the number of nodes in the AST and q the worst case complexity of the synthesis primitive in use[3].

In average, however, our algorithm performs much better than that, since the quadratic effect is limited by the two following aspects:

- The size of the AST is reduced at every iterations;
- Each synthesis step terminates as soon as a suitable node has been found (i.e. it will likely stop before traversing all nodes).

[3] In the case of a LUTs-based synthesis primitive $q = 1$.

Algorithm 1: AST simplification algorithm (starting at function *SimplifyAST*)

```
1  Function SimplifyAST(σ: a Symbolic AST, SP: a synthesis primitive):
2      symVarsMap ← NewHashMap()
3      while σ's root edge is not a symbolic variable do
4          r ← SynthesisStep(σ, SP)
5          if r is an ERROR then
6              return ERROR
7          end
8          (p, σ_ε) ← r
9          σ_p ← ProgramToAST(p)
10         v ←NewSymbolicVariable()
11         σ ←Substitute(σ, σ_ε, v)
12         symVarsMap[v] ← σ_p
13     end
14     return UnrollAST(σ, symVarsMap)
15
16 Function UnrollAST(σ: a Symbolic AST, symVarsMap: an hashtable mapping symbolic variables to
       ASTs):
17     foreach symbolic variable v in σ do
18         if v is in symVarsMap then
19             σ_p ← symVarsMap[v]
20             σ'_p ← UnrollAST(σ_p, symVarsMap)
21             σ ← Substitute(σ, v, σ'_p)
22         end
23     end
24     return σ
25
26 Function SynthesisStep(σ: a Symbolic AST, SP: a synthesis primitive):
27     foreach edge ε in σ using randomized breadth-first order do
28         σ_ε ← sub-tree of σ with root ε
29         p ← SP(σ_ε)
30         if p is not an error code then
31             return (p, σ_ε)
32         end
33     end
34     return ERROR
35
36 Function Substitute(σ: a Symbolic AST, t: an AST or a single edge, s: an AST or a single edge):
37     Substitute all instances of t in σ with s
38     return σ
```

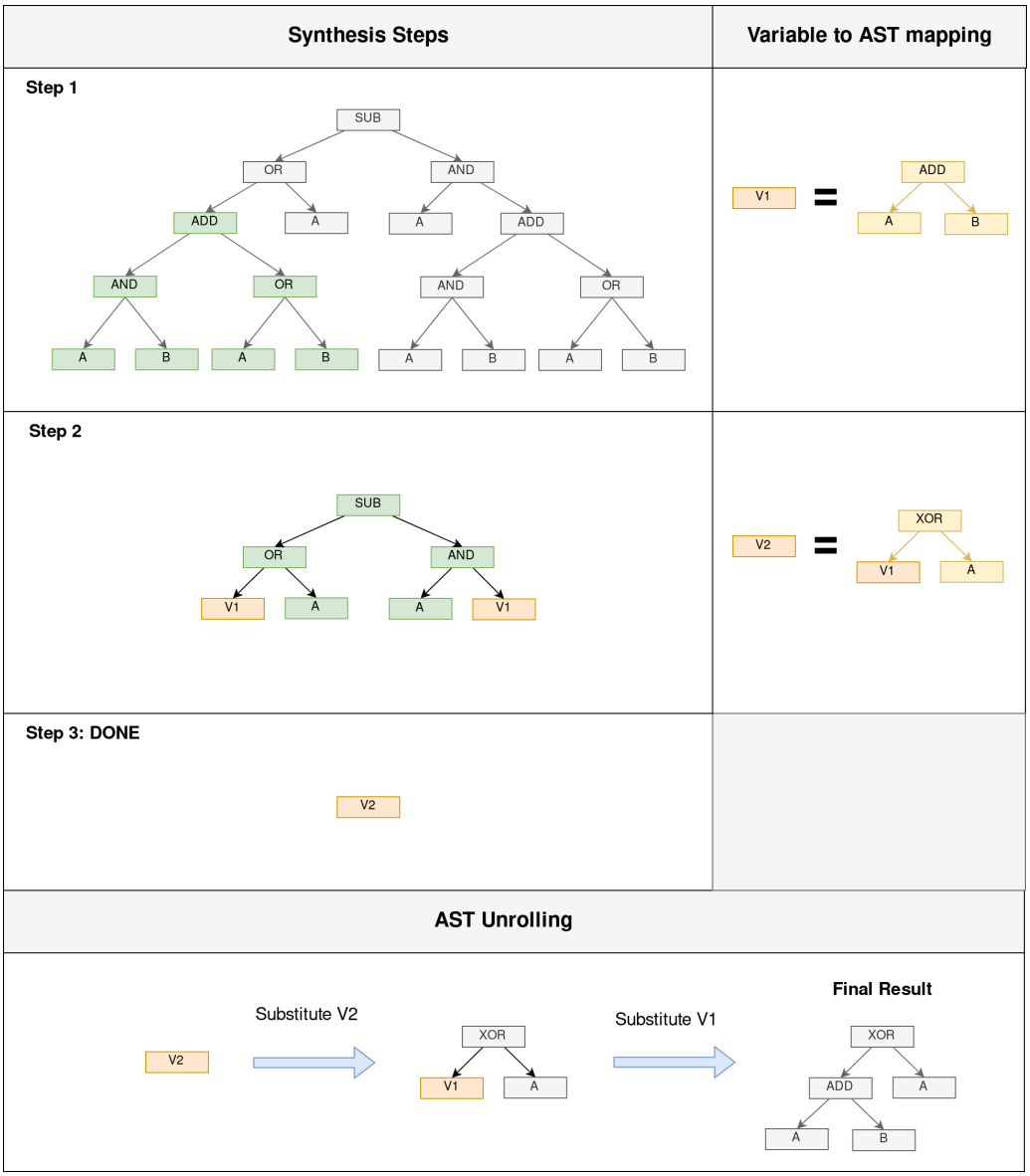


Figure 4.4: AST simplification example. In green are the portion of the AST for which synthesis was successful. The new symbolic variables added to the AST are always kept in orange. Finally the synthesized program's AST are in yellow.

Chapter 5

Implementation

Our implementation of the approach described in Chapter 4 consists of three parts: program tracing, AST extraction and AST simplification.

Figure 5.1 shows the workflow of our implementation. We first trace the obfuscated program (or a portion of it) using Dynamic Binary Instrumentation (DBI). We collect information such as the instructions executed and the values of general purpose registers. Later, we process a selected portion of the trace using a symbolic execution engine to build an AST representation of the program. We translate the AST in a Symbolic AST by identifying inputs and outputs values in the trace. Finally we simplify the AST using the approach presented in Section 4.3.

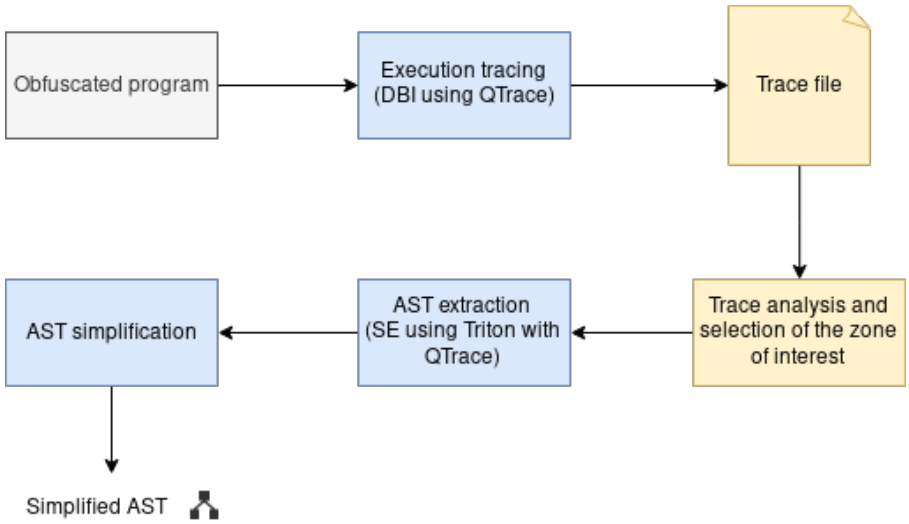


Figure 5.1: Deobfuscation workflow overview.

To trace the obfuscated binaries we used a tracing framework called *QTrace*. For symbolic execution we used *Triton* [11] a dynamic analysis framework. Both *QTrace* and *Triton* have been developed by *Quarkslab*.

In Section 5.1 we briefly present the tracing framework *QTrace* while in Section 5.2 we illustrate our AST extraction approach. Finally, in Section 5.4 we discuss some details of our implemented synthesis approach.

5.1 Program tracing using QTrace

QTrace is a program tracing framework developed in the context of the internship at *Quarkslab* (Section 1.1). It has been particularly designed for complex trace analysis tasks and served as a basis for our research on deobfuscation. It is composed of three components:

- **QTracer**: a Python API and command-line tools for trace collection;
- **QTrace-DB**: a Python API and command-line tools to load and query traces using an SQL backend;

- **QTrace-Analysis**: a Python API implementing various trace analysis passes.

The trace collection module *QTracer* is designed to utilize a set of different tracing backend. At the time of writing only a single tracing backend based on the DBI framework *QBDI* [14] has been implemented, this is the one used for our experimentation.

QTracer can be configured to collect various kind of information while tracing: instructions executed, memory accesses, register values, memory snapshots etc. After execution the trace is stored in a file using a serialization library. The *QTrace-DB* module is then used to load trace files into an SQL database and offers a transparent Python API based on *SQLAlchemy* [54] to query and modify the content of a trace. Using an SQL server as backend makes it possible to issue complex queries cross-referencing different components of a trace, in addition to have all the advantages of modern Database Management Systems in terms of efficiency.

Finally *QTrace-Analysis* offers a collection of analysis routines including the hybrid synthesis-symbolic execution approach object of this work. Figure 5.2 offers a complete overview of *QTrace*'s architecture.

5.2 AST Extraction

For the AST extraction part of our workflow we used the dynamic analysis framework *Triton*. In this step, a sub-set of instructions from a given execution trace is symbolically executed using *Triton*. During symbolic execution the concrete state of *Triton* is constantly synchronized with the state of the obfuscated program recorded in the trace (using *QTrace-DB* as support). The trace used in our implementation contains instructions and registers information [1].

The instructions we need to symbolically execute in order to extract the AST of a target register or memory value, are those corresponding to the *dynamic backward slice* of that target, i.e. all instructions which contributed in computing that target value. However in practice, in our implementation, we execute with *Triton* also all intermediate instructions present in the trace (and not only those in the *dynamic backward slice*). This in order to keep *Triton*'s concrete state consistent with that of the original execution.

The first step toward the extraction of an AST is therefore the selection of a *slicing criterion*, i.e. a register or memory address of interest at a moment during execution. Of course the trace in use must include such criterion: it is important to provide the right parameters/inputs to the program during tracing.[2]. Let us consider the following example:

```
1  unsigned int obfuscated_function(unsigned int a, unsigned int b){
2      return (a | b) + b - ( ~ a & b);
3  }
4
5  int main(int argc, char* argv[]){
6      ...
7      unsigned int a = get_var_a();
8      unsigned int b = get_var_b();
9      c = obfuscated_function(a, b);
10     ...
11 }
```

The above program contains an obfuscated function which we assume difficult to reverse engineer. This last takes as inputs two integers and returns a result which has been computed in an unintelligible way. In this case the value returned by *obfuscated_function* and then stored in variable **c** is of interest to deobfuscate our binary (since it is generated by the obfuscated routine). We can then define our *slicing criterion* as the content of variable **c** just after line 9.

Once the *slicing criterion* has been selected, the second step consists in deciding which portion of the trace has to be taken into account to construct the AST.

[1] *Triton* is able to recover automatically memory access information, otherwise we would have needed to include records of memory accesses in our trace using *QTracer*.

[2] In case of difficulty those inputs can likely be obtained using an approach such as fuzzing or symbolic execution in combination with an SMT solver.

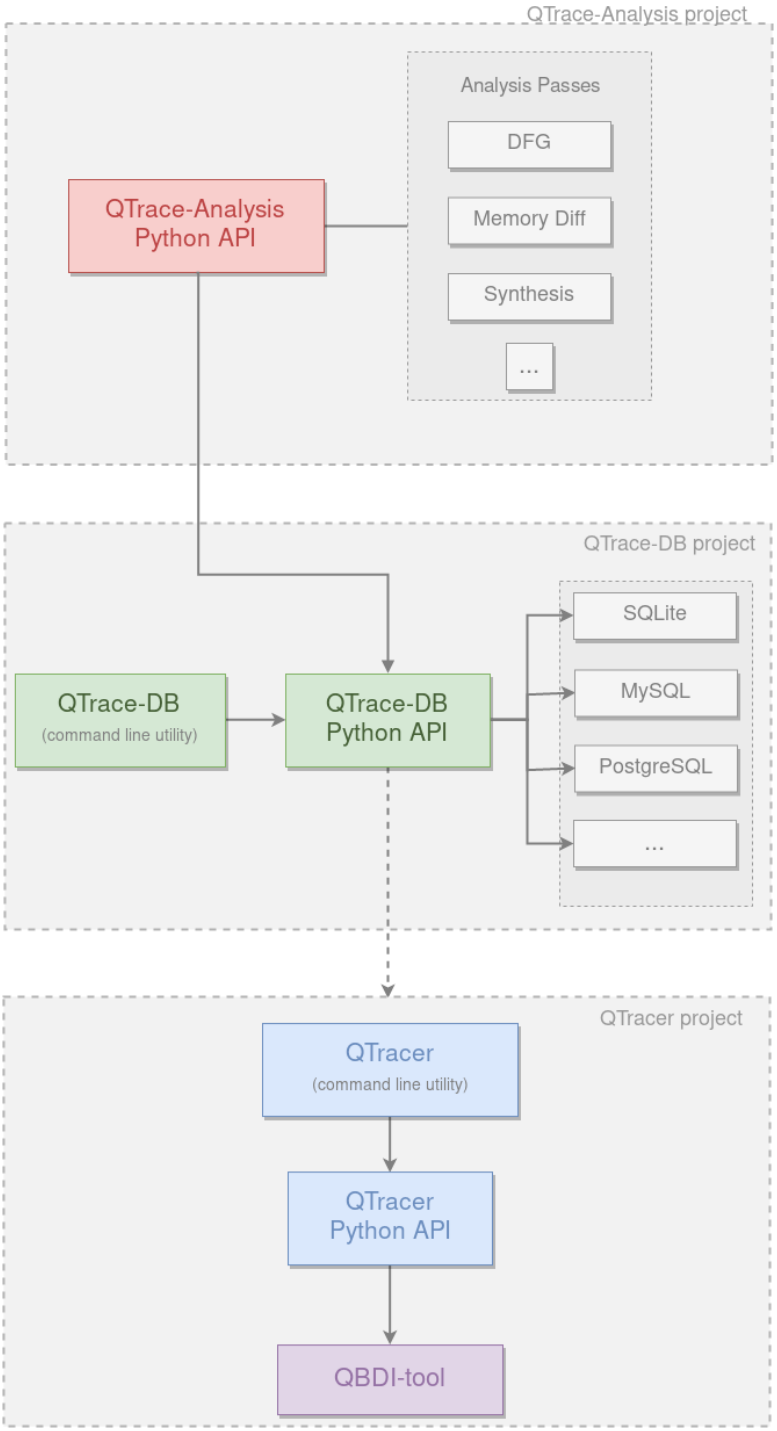


Figure 5.2: Architecture of QTrace (full arrows represent hard dependencies while dotted arrows indicate optional dependencies).

Let us go back to our example, considering variable **c** after the assignment at line 9 as our *slicing criterion*. If we are interested in deobfuscating exclusively the instructions executed inside *obfuscated_function*, then we can limit the portion of the program trace corresponding to lines 1,2,3. One could also include the instructions corresponding to lines 7,8,9 in this case the resulting AST will also encode the operations used in *get_var_a* and *get_var_b* to generate variables **a** and **b**. In practice, the smaller the original AST, the faster and likely more successful will be the synthesis.

While processing the given instructions *Triton* builds up a symbolic state which encodes all registers and memory values in the form of an AST. The Dynamic AST generated *Triton* is then transformed in a Symbolic AST by symbolizing (i.e. translate in symbolic variable) a selection of the AST's leaves.

5.3 LUTs Generation

In this section we present various aspects of our implemented solution for generating LUTs. In particular we describe the approach used to select LUTs' grammar, the order used for generating the expressions in the LUTs and an expression filtering technique to speed up LUTs generation.

Grammars selection: We generate our lookup-tables using one or more target grammars. Those grammars can be generated randomly or crafted manually, and contain only a subset of all derivation rules potentially used by the deobfuscated program. We do that for the two following reason:

- Using a smaller set of derivation rules let us reduce significantly the size of the LUTs and makes it possible to explore higher derivations layers;
- In a real world scenario the operators used by the original binary are conceivably hidden by one or more obfuscation layers. We consider therefore very difficult crafting a grammar matching exactly the grammar used by the original binary.

Expressions generation: In our implementation to generate the expression in the LUT we put a limit only on the number of non-terminal derivations instead of limiting the total number of derivations used.

We call non-terminal all derivations done by using a non-terminal rule where we denote as non-terminal all rules which do not introduce any terminal symbol, to give an example let us consider the non-terminal symbol u_{64}, the derivation rule $u_{64} \rightarrow u_{64} \oplus u_{64}$ is non-terminal while the rule $u_{64} \rightarrow b_{64}$ is terminal since it introduces the terminal symbol b_{64}.

We used this approach of limiting non-terminal derivations, instead of simply counting all types of derivations, to reduce the size of the LUTs while still including expression up to a good derivation depth. Let us consider for example the two following expressions:

1. $(a_{64} + b_{64} + c_{64}) \mid \sim (a_{64} \oplus b_{64})$
2. $-(\sim a_{64} + b_{64}) \mid \sim - \sim c_{64}$

Both expressions can be obtained using 10 derivations, however while expression 1 contains only five operations and thus can be obtained using only five non-terminal derivations, expression 2 needs seven non-terminal derivations to be written. In the context of our work, considering a LUT based on the grammar in Table 6.2, expression 1 would be included in the LUT whereas expression 2 would not.

Expressions filtering: To improve LUT generation time, during the computation of the expressions in our LUT we filter redundant expressions. Let us consider the following example:

1. $(u_{64} + u_{64}) - u_{64}$
2. $(u_{64} - u_{64}) + u_{64}$

The two non-terminal expressions in the above example are equivalent, therefore one could be ignored in subsequent derivations.

To detect redundant expressions we use a simple heuristic: we flag a terminal expression as redundant if, for every LUT's input, the output produced by the expression is already present in the LUT. In turn we flag a non-terminal expression as redundant if every terminal expression directly derived from this last (i.e. without using any additional non-terminal derivation rule) has been flagged as redundant.

5.4 Notes on Synthesis

We implemented the LUTs-based synthesis primitive described in Section 4.2.1 and the AST simplification algorithm described in Section 4.3. Our implementation of the synthesis algorithm is single threaded and entirely written in Python3. Following we describe some details of our implementation introduced in order to enhance synthesis performance.

AST evaluation caching: Evaluating big ASTs based on some input values is an expensive operation. For this reason we use a cache containing previous ASTs evaluations. Every time we use an AST as oracle and evaluate it on a given set of inputs, we store an entry in the cache relative to this evaluation. If later an equivalent AST will need to be evaluated on that same set of inputs we use instead the cached result.

Multiple LUTs: Given the size of the grammar obtained by considering all possible arithmetic and boolean operation, creating a big comprehensive LUT is very costly. Instead, we prefer using multiple smaller LUTs based on different subsets of the original grammar.

Same LUTs inputs: The inputs used to construct the LUTs are the same used to evaluate the AST. Therefore we construct our LUTs using always the same set of inputs in order to maximize AST evaluation caching.

Select shortest expression: During synthesis we iteratively query all LUTs. If multiple LUTs were able to synthesize the AST under analysis, we select the shortest expression provided.

Constant ASTs: Whenever an AST always evaluate to the same result, we do not use the LUTs and directly synthesize the AST to a constant.

Chapter 6

Experimental Validation

We empirically evaluated our approach on a large number of samples, obfuscated using *Tigress* [3] obfuscator. This chapter illustrates our validation process.

In Section 6.1 we describe our experimental setup, including the datasets and LUTs used for synthesis. In Section 6.2 we illustrate and discuss the results of our experimentation.

6.1 Experimental Setup

We tested our approach on three different datasets using three different kind of obfuscations techniques. We run our test on a notebook with an Intel Core i5-3230M @ 2.60GHz CPU and 8 GB of RAM.

6.1.1 Datasets

For our experimentation we use three different datasets each consisting of 500 obfuscated functions. We tested our approach on the same *Tigress* MBA dataset employed by the authors of *Syntia* [17], the most related approach, as well as two additional datasets of higher complexity. We didn't use any compiler optimization during the compilation of our datasets.

Dataset 1: The first dataset, created by the authors of *Syntia*, includes 500 randomly generated functions. Each function takes five arguments as input and returns the result of an expression based on the grammar illustrated in Table 6.1. The expressions in this dataset are generated using from 3 to 5 derivations (i.e. they are expressions of layer 3 to 5). To give an example, the expression $a_{64} \oplus d_{64}$ is an expression of layer 3 while $(a_{64} \times c_{64}) - c_{64}$ is an expression of layer 5.

We obfuscated this dataset using the same exact obfuscation techniques used by the authors of *Syntia* [17]: *Tigress*'s *EncodeArithmetic* and *EncodeData* transforms. The first obfuscation technique hides the original expression using complex MBA equivalences, while the second transformation encodes all integer variables. The average layer of the obfuscated expressions is 151.

Datasets 2: For our second, we crafted 500 random functions taking as parameters five arguments and returning the result of an expression based on the grammar in Table 6.1, the same used by the authors of *Syntia* to generate dataset 1. However in this case we generated expressions of an higher order of complexity, using from a minimum of 6 derivations up to 21 derivations. For instance the following expression, obtained using 21 derivations, is an example of expression included in our dataset:

$$(((b-(b\ \&\ b))\ \&\ b)\times(b\oplus e))+(((b\oplus e)\oplus b)\times(e\ \&\ e))$$

We have obfuscated the expressions in dataset 2 using *Tigress*'s *EncodeArithmetic* transform and obtained a set of obfuscated expression of average layer 244.

N	$=$	$\{u_{64}\}$
Σ	$=$	$\{a_{64}, b_{64}, c_{64}, d_{64}, e_{64}\}$
R	$=$	$\{u_{64} \to u_{64} + u_{64},\ u_{64} \to u_{64} - u_{64},$ $u_{64} \to u_{64} \times u_{64},\ u_{64} \to u_{64} \oplus u_{64},$ $u_{64} \to u_{64} \mid u_{64},\ u_{64} \to u_{64}\ \&\ u_{64},$ $u_{64} \to \sim u_{64},\ u_{64} \to -u_{64},$ $u_{64} \to a_{64},\ u_{64} \to b_{64},\ u_{64} \to c_{64}$ $u_{64} \to d_{64},\ u_{64} \to e_{64}\}$
S	$=$	$\{u_{64}\}$

Table 6.1: Grammar used to generate the expressions in *Syntia* [17] MBA dataset as well as datasets 2 and 3. Here the notation x_n indicates a variable of n bits[1]. The '$\oplus$' symbol indicate a bitwise XOR operation, '|' is the bitwise OR, '&' indicates a bitwise AND operation and '$\sim$' a bitwise NOT.

Datasets 3: For our third dataset we used the same 500 random functions encoding expressions from 6 up to 21 derivations employed in dataset 2. However, in dataset 3, we have obfuscated the expressions using both *Tigress*'s *Virtualize* and *EncodeArithmetic* transforms.

6.1.2 Lookup-tables

For our experimental evaluation we utilized 20 different LUTs. Each LUT was generated taking into account a random subset of the grammar in Table 6.1 (for the reasons explained in Section 5.3). We limited the size of each subset to include only five non-terminal derivation rules and three operands.

Table 6.2 shows an example of subset of grammar in Table 6.1 used for LUT generation.

We generated the expression in each LUT using up to five non-terminal derivations. Our 20 LUTs contain in average 703981 (terminal) expressions.

Each expression in our LUTs is evaluated on five different inputs where each input variable has been selected randomly in a space of 64 bits. That is, the look-up key to access an expression in a LUT is composed of five output values. Moreover, as discussed in Section 5.4, we employ the same set of inputs for every LUT in order to maximize the efficiency of our caching mechanism.

We implemented our LUTs using Python3 dictionaries and stored for later use using Python's serialization module *pickle* [55]. The average size of the stored LUTs is 50 MB, with the biggest LUT measuring 204 MB. After zip compression our LUTs measure in average 37 MB with a peak of 101 MB.

N	$=$	$\{u_{64}\}$
Σ	$=$	$\{a_{64}, b_{64}, c_{64}\}$
R	$=$	$\{u_{64} \to u_{64} + u_{64},\ u_{64} \to u_{64} \times u_{64},$ $u_{64} \to -u_{64},\ u_{64} \to u_{64} \mid u_{64},$ $u_{64} \to \sim u_{64},\ u_{64} \to a_{64},\ u_{64} \to b_{64},\ u_{64} \to c_{64}\}$
S	$=$	$\{u_{64}\}$

Table 6.2: Example of grammar used for LUT generation.

6.2 Results

In this section we illustrate the results of our experimental evaluation. In Section 6.2.1 we discuss the success rate of our algorithm or, in other words, in how many cases our algorithm (Section 4.3) was able to converge and retrieve a result. In Section 6.2.2 we analyze the correctness of the results retrieved by our approach. In Section 6.2.3 we illustrate the execution time needed for our implementation to deobfuscate the expressions in

[1] While the original dataset employed 32-bits variables, we use 64-bits variables instead.

our datasets. In Section 6.2.4 we evaluate the deobfuscated programs generated by our approach in terms of "understandability". Finally in Section 6.3 we discuss some important points and limitations of our approach.

6.2.1 Success Rate

The simplification algorithm illustrated in Section 4.3 does not guarantee a successful completion. In case the synthesis primitive used for AST simplification was not able to successfully synthesize any of the sub-trees of the AST, the simplification routine *SimplifyAST* would stop and return an error instead of the expected simplified expression.

Despite this, during our experimental evaluation we were able to always construct an expression out of the obfuscated samples in our datasets. In other words, our LUT-based synthesis primitive did not fail any synthesis step and always found a suitable sub-tree candidate for synthesis.

6.2.2 Correctness

Similarly to other deobfuscation approaches based on inductive program synthesis, our approach is unsound. The output expression is constructed in such a way that it does not necessarily behave identically to the target obfuscated program for all possible inputs. For this reason we decide to verify the overall correctness of the expressions generated by our approach.

To validate our result we evaluate both the obfuscated expressions and the simplified expression on 100 different random inputs. We consider as "correct" the simplified expressions that always provide the same output as the corresponding obfuscated expression.

Table 6.3 shows the result of our validation.

	Number of correct expressions
Dataset 1	500/500
Dataset 2	479/500
Dataset 3	486/500

Table 6.3: Expressions correctly deobfuscated using LUTs encoding 5 random inputs.

By simply using LUTs based on an input vector of only five values our algorithm was able to correctly deobfuscate, on the first run, most of the samples in our datasets, respectively 100%, 95% and 97%.

It is interesting to notice how dataset 1, which contained very simple expressions if compared to the other datasets, was deobfuscated without errors. We believe this good results should be partially attributed to the small size of the expressions in this dataset. Indeed, given their small size, most of the expression in dataset 1 were already present in our LUTs and, as a result, directly deobfuscated at the first synthesis step. Those expressions in dataset 1 which were not present in our LUTs were, nevertheless, easily decomposed by our simplification algorithm and reconstructed using the combination of smaller expressions in different LUTs.

Our result constitutes a significant improvement over *Syntia* [17] where only 448 expressions out of 500 were deobfuscated on the first run, and only 495 out of 500 expressions after nine runs of Monte Carlo Search Tree black-box synthesis.

The deobfuscated expressions which did not pass our correctness test (using LUTs of 5 random inputs) have a correctness rate of 59.23% for dataset 2, and 53.71% for dataset 3, meaning that even those expression which we have classified as incorrect are nonetheless semantically close to the obfuscated expression[2].

The precision of our approach can be tuned constructing the LUTs on an higher/lower number of input parameters.

[2]Notice that in some context this closeness might be sufficient, e.g. when the obfuscated expression is in practice evaluated only on a reduced set of possible input values.

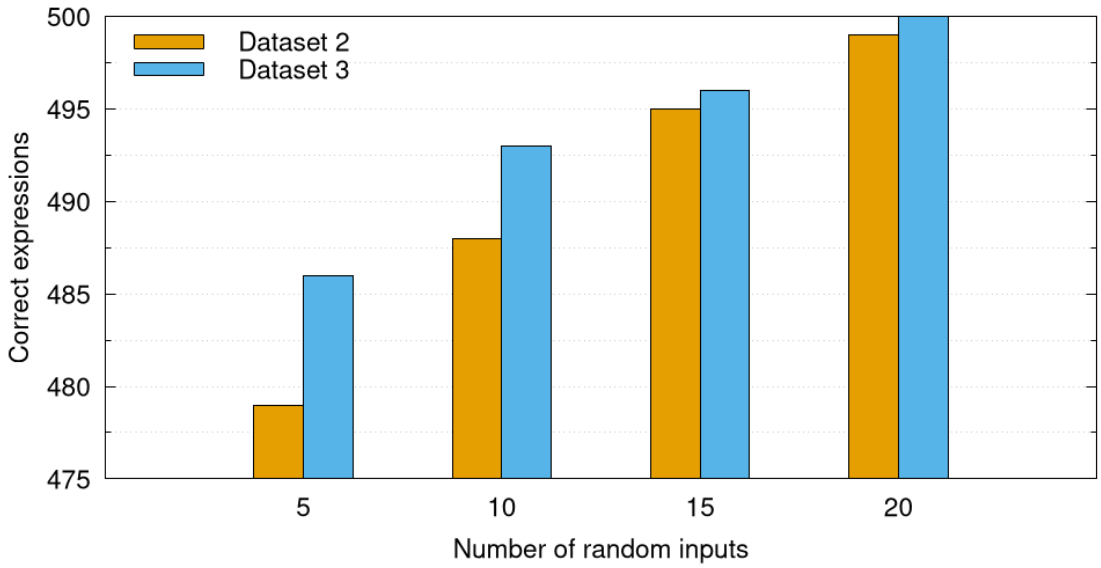


Figure 6.1: Results correctness using LUTs with different number of uniformly inputs.

We also tested the correctness of the results yield by our approach, on dataset 2 and 3, using LUTs based on a larger sets of uniformly random inputs. Figure 6.1 shows the evolution in correctness as the number of inputs parameters in the LUT changes. It can be observed that the correctness of our approach is directly proportional to the number of inputs encoded in the LUT. According to our results, 20 seems to be the appropriate number of random inputs.

The results we have presented so far were obtained using LUTs based on uniformly random sets of inputs. For completeness we have also tested expression correctness using a manually crafted set of input, including a mix of random inputs and "magic numbers" likely to trigger edge results (e.g. 0, 1, 1024, etc.). Figure 6.2 shows the results of our tests. It can be observed that, in our tests, for the same number of inputs, the LUTs using manually crafted sets of inputs performed better than those using only random inputs. In particular, using our selection of inputs, we were able to successfully deobfuscate 500 expressions out of 500 for both datasets 2 and 3 using only 15 inputs.

15 LUTs having the same attributes described in Section 6.1.2 but 20 inputs instead of 5. Using the aforementioned LUTs

6.2.3 Deobfuscation Time

In order assess the practical applicability of our simplification algorithm, we measured the execution time of the simplification routine on every obfuscated sample. The measurements of execution time for our implementation of the simplification algorithm in Section 4.3 were very promising, with most samples being deobfuscated almost instantly.

The box-plot in Figure 6.3 shows a detailed overview of the execution times over the three datasets. All samples in dataset 1 are simplified in less than half second. Once again our approach improves on *Syntia* [17] where the samples from this same dataset where synthesized with an average time of 4.0 seconds per run, whereas the longest time taken by our algorithm to synthesize an expression in dataset 1 was 0.45 seconds.

Our approach has also recorder much shorter execution times than those of other state-of-the-art program synthesis techniques presented in Section 3.1.

Despite the increase in AST size and semantic complexity of the underlying expression, most expressions in datasets 2 and 3 are simplified in less than 0.2 seconds, with the longest simplification time taking respectively 1.69 and 5.65 seconds.

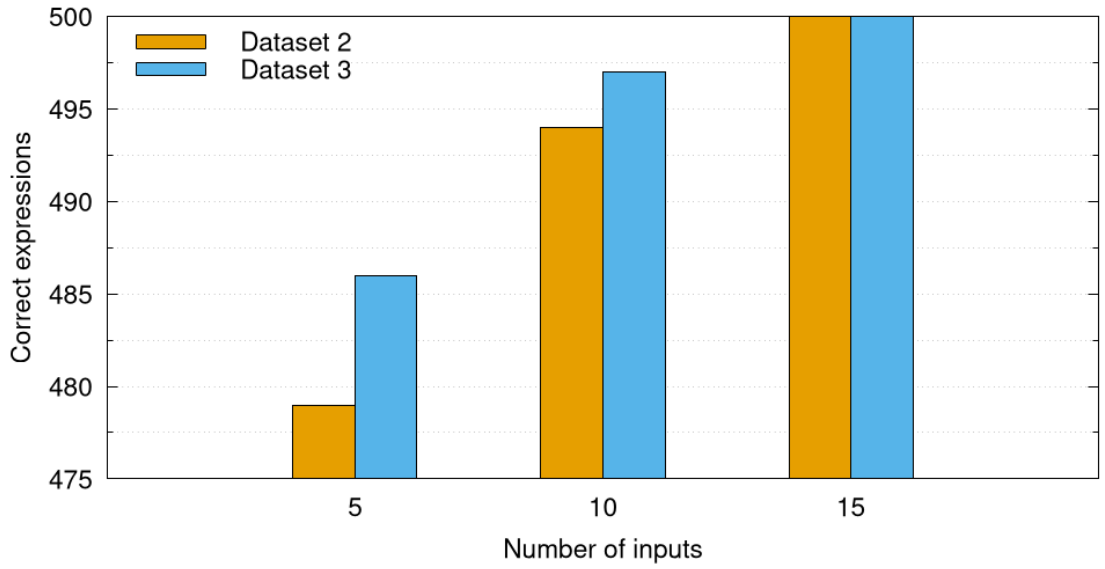


Figure 6.2: Results correctness using LUTs with different sizes of manually selected sets of inputs.

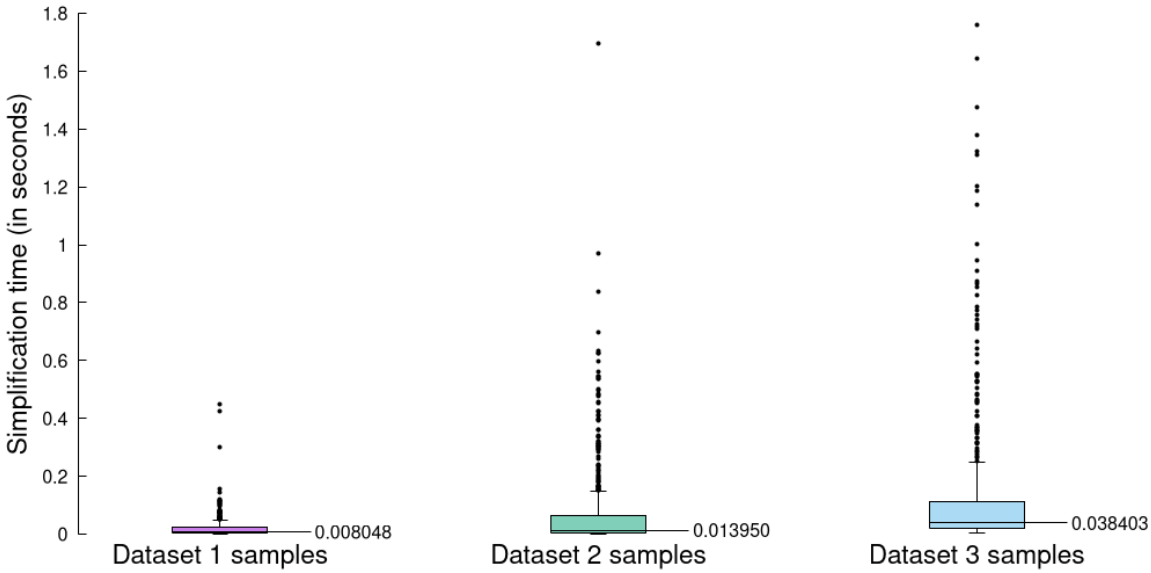


Figure 6.3: Execution time of the simplification routine. The label in each box indicates the median deobfuscation time.

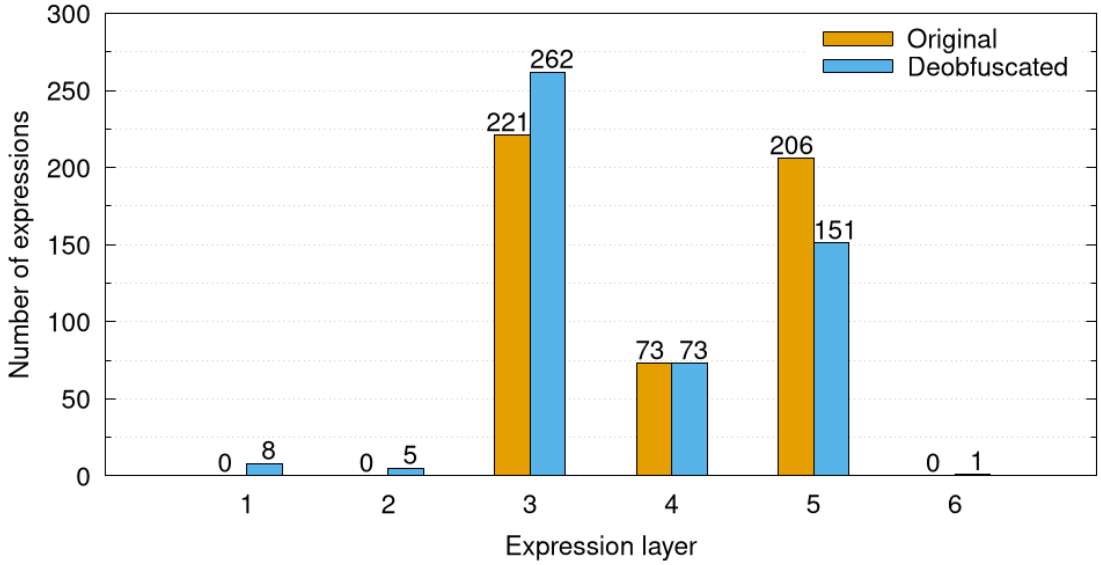


Figure 6.4: Size comparison between deobfuscated and original expressions of dataset 1.

6.2.4 Understandability

To assess the effectiveness of our approach we also evaluate it in terms of "understandability" of the obtained expressions. To do that we compare the size, with regard to the number derivations needed (i.e. layer), of the deobfuscated expressions with the size of the original expressions in the non-obfuscated version of the program.

We believe that an effective deobfuscation technique should retrieve results which complexity is not very far from that of the original non-obfuscated program.

Figure 6.4 compares the size of the deobfuscated expression in dataset 1 with the original expressions. We can observe that, except for one, all the deobfuscated expressions are of smaller or equal in size if compared to the original expressions. In other words, our tool was able to easily identify and simplify all redundant operations which were casually generated during the creation of the dataset. To give an example, an expression such as $(a \mid a) + b$ is deobfuscated to the equivalent and shorter expression $a + b$. Deobfuscated expressions of size one indicate constants and single-variables.

Figures 6.5 and 6.6 show the same comparison for respectively dataset 2 and 3. Here most of the deobfuscated expressions have similar or smaller size than the original expression, however in the case of this two more complex datasets, a non-negligible number of deobfuscated expression has a bigger size than their original counterpart.

In Figure 6.7 we compare the number of expressions which we were able to simplify to less than 21 terms with the number of expressions that required more than 21 terms. We observe that, depending on the complexity of the dataset, we were able to deobfuscate to a relatively good simplicity level from 70% to 100% of the expressions. While the expressions marked in red were not as simple as the corresponding original expressions, they were in general much simpler than their obfuscated counterpart.

6.3 Discussion

In the previous sections we have illustrated the results yield by our evaluation. We have found that, in terms of correctness of the generated results and execution time, our approach performs considerably better than *Syntia* [17].

In this section we further discuss some points and limitations of our approach.

LUTs generation

The effectiveness of our approach is heavily influenced by the quality of the LUTs used during simplification, one of this quality being the size of the LUTs. The bigger the LUT the more likely it will be to find an

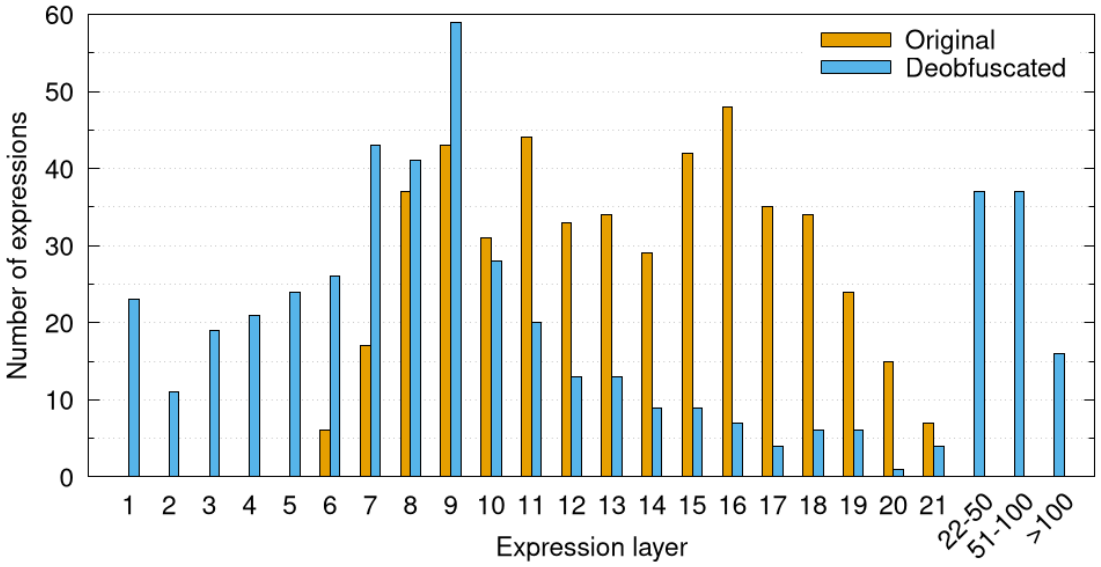


Figure 6.5: Size comparison between deobfuscated and original expressions of dataset 2.

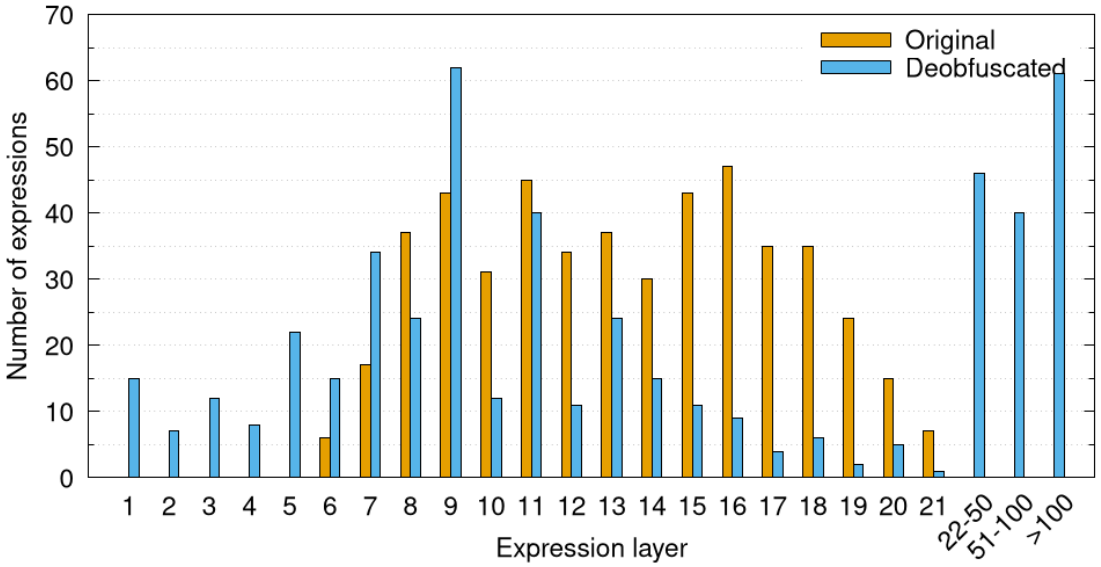


Figure 6.6: Size comparison between deobfuscated and original expressions of dataset 3.

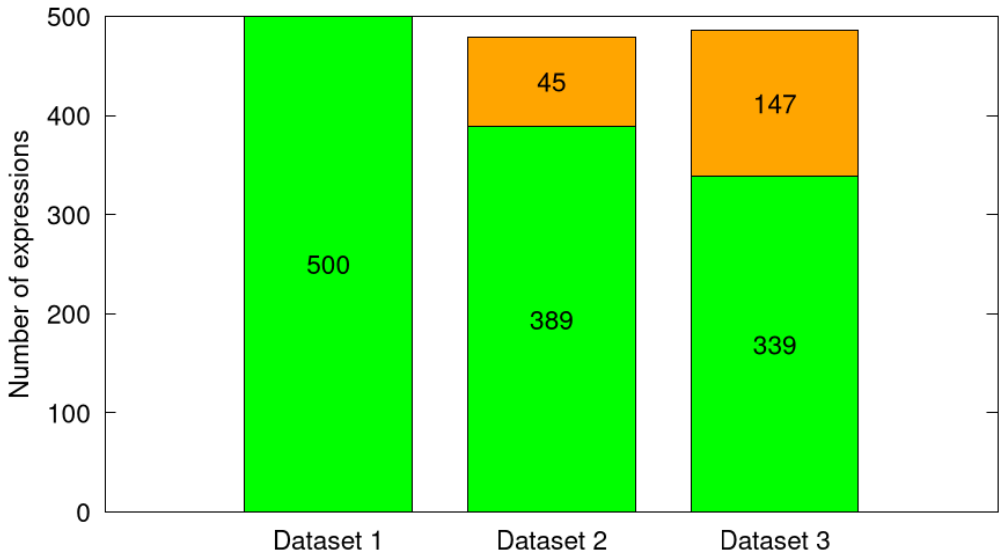


Figure 6.7: Number of deobfuscated expressions of layer 21 or less (in green) compared with the number of deobfuscated expressions of layer 22 or more (in orange).

expression matching a given sub-tree during simplification. However generating big LUTs can be an expensive task.

Fortunately there are several techniques which can be used to optimize LUTs generation. In Section 5.3 we introduced a simple filtering technique to avoid generating redundant expression. We also consider dynamic programming, parallelization and other optimization techniques such as those coming from the literature on superoptimization (such as those described in the study by Embecosm [50]) as suitable approaches to improve LUTs generation.

Furthermore LUTs can be stored and used for different deobfuscation tasks, and can be generated directly by the end user or distributed by a third-party (similarly to pre-computed rainbow table used to reverse cryptographic hash functions).

LUTs generation parameters selection

According to our approach, generating one or multiple LUTs involves the selection of a number of parameters: input arguments of the LUTs, grammars and the maximum derivation depth used to construct the LUTs. In case the LUTs are directly generated by the end user, this last will responsible for choosing a good set of parameters. In case of LUTs distributed by a third party, the end user will be responsible for selecting a set of LUTs which characteristics are closer to the desired set of parameters. In both cases it is reasonable to ask how those parameters should be chosen.

In their work, Jha et al. [45] delegate to the end user, who should be guided by the application domain, the choice of base components for synthesis. While, in principle, we agree with Jha et al., we also believe that their approach can not be always applied to synthesis-based deobfuscation in general and to our approach in particular, for the two following reasons:

- Often it is very difficult to determine the application domain of an obfuscated routine;
- Given the cost of LUTs creation (especially for big LUTs) we encourage LUTs reuse and consequently discourage the creation of highly specialized LUTs (i.e. using a grammar targeting a specific program).

Following we provide some guide lines for parameters choice

- **Grammar and derivation depth:** as a general rule, the bigger is the grammar and the number derivations used, the richer and more effective will be the correspondent LUT. Notice however that the generation time and size of the LUT grow exponentially with the size of the grammar and/or the number of derivations.

When working with particularly large grammars it is possible to find a balance between LUTs' precision and generation time by selecting n (random) subsets of size s of the given grammar, and generating n corresponding LUTs based on the selected subsets[3]. Here n and s can be used to adjust LUT's precision as desired.

- **Inputs:** to maximize LUTs' precision inputs should be uniformly distributed along the range of possible values, and should include magic numbers likely to trigger edge cases (e.g. 0, 1, 128, -128, etc.). Since size of the LUT grows only linearly with the number of inputs, we encourage the use of big set of inputs (> 20) so to maximize correctness of the deobfuscated expression.

Unsoundness

Our approach, as well as many other synthesis-based approaches is unsound or, in other words, it does not guarantee the correctness of the result. The obtained result strongly depends on the input used to generate the LUTs and later evaluate the AST.

In Section 6.2 we have shown how the precision of our approach can be tuned increasing/decreasing the number of inputs used to generate the LUTs. Nonetheless the correctness of the generated expression can not be assured.

In particular the "representativity" of the inputs provided is critical for the soundness of our approach. For this reason our approach will likely not succeed in deobfuscating correctly Dirac functions such as the one presented in Section 2.1.

A possibility in order to make the approach sound is to employ a SMT solver to verify whether the generated expression is equivalent to the obfuscated one and, in case of dis-equivalence, generate a counter example. The provided counter example can be used as additional input to the LUTs.

Performance

Performance of synthesis-based approaches is usually only influenced by the complexity of the underlying code's semantic. This is however not totally true in the case of our hybrid approach, which performances are strongly influenced by the structure of the AST. The results illustrated in Section 6.2 for example, highlights how an additional layer of virtualization-based obfuscation was able to slow down our simplification routine by almost a factor of $\times 3$ (Figure 6.3). Nonetheless the execution time measured during our experimental evaluation is by an order of magnitude inferior if compared to other synthesis-based deobfuscation approaches.

Encodings

In its current formulation our approach is not effective whenever operations inside the AST are encoded using an homomorphic encoding which modifies the semantic of the underlying operation. This limitation applies for example when the data is encoded using homomorphic encryption schemes. However we believe that our approach can be adapted, on a case by case basis, to apply also to this kind of obfuscation.

Mixed operations

The algorithm presented in Section 4.3 relies on the assumption that there exist some (unknown) sub-tree of the obfuscated AST encoding a particular portion of the original AST (i.e. prior to obfuscation). In other words we assume that it is always possible to find meaningful portion of the AST under analysis which can be isolated in order to be synthesized separately from the rest of the AST.

This assumption however does not always apply. Let us take as example the case of MBA expressions. As illustrated in Section 2.1.2, MBA-obfuscated expressions are obtained by iteratively matching and replacing parts of the original formula with equivalent but more complex expressions. While current literature and open-source obfuscation software on MBA-based obfuscation only present rewriting rules targeting single operations

[3] Given that the selected subsets may intersect, it is important to keep track of the generated expression in order to avoid LUTs overlapping.

(Table 2.2) or expressions of relatively low complexity [3, 2, 21, 29], it is always possible to generate new and more complex MBA transformations mixing together multiple operations, thus making the underlying single operations difficult to separate from the rest.

This result can be already obtained using simple MBA rewriting rules and exploiting the commutative and/or distributive property of certain operators. Let us consider the follow rewriting rule:

$$a+b \rightarrow (a \vee b)+(a \wedge b)$$

By taking advantage of the commutative property of the addition we can apply the above rule twice in such a way to transform the expression $a+b+c$ in an equivalent expression "fusing" together all three variables:

$$a+b+c \rightarrow (a \vee b)+(a \wedge b)+c \rightarrow (a \vee b)+(a \wedge b \vee c)+(a \wedge b \wedge c)$$

Considering this last example, our approach would be able to deobfuscate it only provided a LUT having the $+$ operator in its grammar and containing expressions derived at least up to layer five.

Such constructs can be counter attacked using bigger LUTs or, as in the case of the last example, crafting a special set of LUTs including only a selection of commutative or distributive operators and expanding it to a larger number of derivations.

Constants unfolding

In the context of our approach the presence of constants in an obfuscated expression can be handled by substituting constants with symbolic variables of the same size. However the presented approach does offer only limited support for obfuscation by constants unfolding.

Let us consider the following expression:

$$(x \oplus 92) = (x \oplus (45+12+42-7))$$

Here the value 92 has been unfolded to $45+12+42-7$. Support for this simple kind of constant unfolding is simply added by checking whether a portion of the AST does not contain any symbolic variable, prior to constants symbolization, and substituting the non-symbolic portions of the AST by a symbolic variable of the same size:

$$(x \oplus (45+12+42-7)) \rightarrow (x \oplus \text{symvar})$$

However this approach does not cover for example the case where the sub-tree containing the unfolded value also contain other symbolic variables. Let us consider the following expression extracted from [21]:

$$(x \oplus 92) = ((((((x \oplus 127) \oplus 127) \oplus 92) \oplus 122) \oplus 107) \oplus 17)$$

Here the integers 127, 127, 92, 122, 107 and 17 represent a "xor unfolded" version of the value 92. They can not be isolated, or in other words, every portion of the AST containing the unfolded integer contains also the variable x. Furthermore, considering each of those integers as a separate symbolic expression would not help producing the original expression but would instead cause our algorithm to return the obfuscated version.

While in some instances this limitation could be partially solved by adding constants to the LUTs or using the LUT-based synthesis primitive in combination with another primitive, further work is needed to explore more in depth possible improvements to our approach in order to make it resilient against this kind of obfuscation.

Symbolic execution

As our approach relies on symbolic execution to build an AST representation of the program, later used for synthesis, it also suffers from counter measures targeting symbolic execution such as those described in [56] and anti-analysis transformations already implemented in tools such as *Tigress* [3].

This limitation could be overcome using static analysis AST extraction techniques in place of symbolic execution, this approach risks however to worsen the quality of the results by including instructions which are not part of the original program (e.g. in case of a VM-obfuscated program, an AST obtained from the static disassembly of the program would also include the instruction encoding the VM semantic). This alternative approach should at least be further investigated.

Chapter 7

Conclusion

We have presented a novel approach for execution traces deobfuscation. Our approach takes advantage of symbolic execution to build an AST representation of the program which is later used to synthesize the program following a *divide and conquer* strategy to iteratively reduce the size of the AST.

We have also illustrated a constant time synthesis primitive where one or more LUT are used to encode expression output, as well as some techniques to optimize LUT generation.

Our experimental validation shows that our approach is viable, in terms of execution time and size of the generated LUTs. We were also able to show the effectiveness of our approach by simplifying obfuscated expressions to a size comparable to the one prior to obfuscation.

Some of the possible applications of our work include enhancing code comprehension for reverse engineering tasks as well as improving concolic execution performance by simplifying queries to the solver (particularly useful in case of obfuscated conditionals). Differently to other simplification approaches, the technique presented in this work can be applied directly to compiled binaries without any pre-processing.

7.1 Future work

In this section we present some additional ideas for future research directions based on our work.

LUT size reduction

In our current implementation we used associative arrays based on hash tables as data structure to implement the LUTs. Since the effectiveness of our approach strongly depend on the size of the LUTs, we believe that our approach could benefit from any enhancement in terms of memory size used for the LUTs.

Among the possible candidates we are studying the possibility of employing, for example, rainbow table-like structures to encode LUTs.

LUT grammar selection

Currently we generated our LUTs based on random subsets of the grammar used by the obfuscated program. A future research direction could investigate better grammar selection techniques based on further analysis of the obfuscated program.

Synthesis primitive

The same AST simplification techniques proposed in this work could be used to implement other hybrid synthesis approaches using different synthesis primitives. Therefore a possible future research direction could investigate on the effectiveness of different synthesis techniques or various combinations of them.

Superoptimization

We believe that our AST simplification approach could also be applied in the context of compilation optimization as a superoptimization pass. As discussed in Section 6.3 a theorem prover could be employed to ensure soundness.

User study

It would be of interest verifying, by the mean of a user study, to what degree the result of our tool are actually useful for practical reverse engineering.

Testing commercial obfuscators

During our experimental validation we have tested our approach against *Tigress* [3] a freely available C source code obfuscator. We believe that further work is needed to verify the effectiveness of our approach against commercial obfuscators, often implement stronger obfuscation mechanism than what those of public domain obfuscators.

Investigate variants of our algorithm

We shall evaluate the effectiveness of variant of our AST simplification algorithm (Section 4.3): for example, giving priority to bigger substitutions while taking into account also the substitution of repeated sub-trees, instead of prioritizing a single bigger sub-tree which is the first encountered from top to bottom. An research effort could target the construction of an algorithm which guarantees the shortest possible AST give a set of possible sub-trees substitutions.

Test larger grammars

In our experimental evaluation we have tested our approach using a relatively small base grammar (Table 6.1). To further validate our approach we shall test it on larger grammars, including a wider set of operators (e.g. division, modulo operation, bit rotation, etc.). In order to extensively cover this bigger grammar we shall also consider using a larger number of LUTs generated on random subsets of the grammar.

Apply to programs of higher cyclomatic complexity

Currently our approach targets obfuscated dynamic program slices that do not contain conditional branches or loops. This is because the *Dynamic AST* (DAST) presented in Section 4.1 is based on a single execution and does not encode control information.

However we believe that our approach can also be used to deobfuscate programs including control structures, thus presenting higher cyclomatic complexity than the programs used for our tests.

In order to do that we shall first extract an AST representation of the program contain control information. This can be done, for example, by merging together multiple DASTs or using static analysis (better if combined with an approach such as the one described by Bardin et al. [53] to clean the program from unused code). Of course this will require the base grammar to be also expanded in order to include conditional statements.

Combine with expression simplification

The authors of *Syntia* [17] use a final expression simplification step (i.e. Z3's *simplify*) to clean the result produced by their synthesis technique.

We shall also explore the possibility of using our approach in combination with expression simplification approaches, such as those presented in Section 3.2. In particular we can simplify the AST (or parts of it) using one or more expression simplification techniques prior to applying our algorithm in Section 4.3, to reduce the size of the AST beforehand, or after applying it, to clean the result yield by our approach.

Bibliography

[1] Shohreh Hosseinzadeh, Sampsa Rauti, Samuel Laurén, Jari-Matti Mäkelä, Johannes Holvitie, Sami Hyrynsalmi, and Ville Leppänen. Diversification and obfuscation techniques for software security: A systematic literature review. *Information and Software Technology*, 104:72 – 93, 2018.

[2] P. Junod, J. Rinaldini, J. Wehrli, and J. Michielin. Obfuscator-llvm – software protection for the masses. In *2015 IEEE/ACM 1st International Workshop on Software Protection*, pages 3–9, May 2015.

[3] C. S. Collberg, S. Martin, J. Myers, and B. Zimmerman. The tigress C Diversifier/Obfuscator. `http://tigress.cs.arizona.edu/`.

[4] VMProtect Software. `https://vmpsoft.com/`.

[5] Andrew Appel. Deobfuscation is in np. *Princeton University, Aug*, 21:2, 2002.

[6] Jasvir Nagra and Christian Collberg. *Surreptitious software: obfuscation, watermarking, and tamperproofing for software protection*. Pearson Education, 2009.

[7] Sharath K Udupa, Saumya K Debray, and Matias Madou. Deobfuscation: Reverse engineering obfuscated code. In *12th Working Conference on Reverse Engineering (WCRE'05)*, pages 10–pp. IEEE, 2005.

[8] RIPSEC. LLVM-Deobfuscator. `https://github.com/RPISEC/llvm-deobfuscator`.

[9] GraxCode. Zelixkiller. `https://github.com/GraxCode/zelixkiller`.

[10] HoLLy-HaCKeR. Eazfixer. `https://github.com/HoLLy-HaCKeR/EazFixer`.

[11] 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, France, Rennes*, pages 31–54, 2015.

[12] Quarkslab - Irma. `https://irma.quarkslab.com/`.

[13] Quarkslab - Epona. `https://epona.quarkslab.com/`.

[14] QBDI - QuarkslaB Dynamic binary Instrumentation. `https://qbdi.quarkslab.com/`.

[15] Adrien Guinet, Ninon Eyrolles, and Marion Videau. Arybo: Manipulation, canonicalization and identification of mixed boolean-arithmetic symbolic expressions, 2016.

[16] Ninon Eyrolles, Louis Goubin, and Marion Videau. Defeating mba-based obfuscation. In *Proceedings of the 2016 ACM Workshop on Software PROtection*, pages 27–38. ACM, 2016.

[17] Tim Blazytko, Moritz Contag, Cornelius Aschermann, and Thorsten Holz. Syntia: Synthesizing the semantics of obfuscated code. In *26th {USENIX} Security Symposium ({USENIX} Security 17)*, pages 643–659, 2017.

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

[19] Shafi Goldwasser and Guy N Rothblum. On best-possible obfuscation. In *Theory of Cryptography Conference*, pages 194–213. Springer, 2007.

[20] Lukas Zobernig, Steven D. Galbraith, and Giovanni Russello. When are opaque predicates useful? Cryptology ePrint Archive, Report 2017/787, 2017. https://eprint.iacr.org/2017/787.

[21] Ninon Eyrolles. *Obfuscation with Mixed Boolean-Arithmetic Expressions: reconstruction, analysis and simplification tools*. PhD thesis, Université Paris-Saclay, 2017.

[22] Arybo documentation. https://pythonhosted.org/arybo/index.html.

[23] Xortool - a tool to analyze multi-byte xor cipher. https://github.com/hellman/xortool.

[24] Didier Stevens. Xorsearch and xorstring. https://blog.didierstevens.com/programs/xorsearch/.

[25] Bypassing Antivirus With Ten Lines of Code or (Yet Again) Why Antivirus is Largely Useless. https://hackingandsecurity.blogspot.com/2017/07/bypassing-antivirus-with-ten-lines-of.html.

[26] Igor Aronov. An Example of Common String and Payload Obfuscation Techniques in Malware. https://securityintelligence.com/an-example-of-common-string-and-payload-obfuscation-techniques-in-malware/.

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

[28] Ninon Eyrolles. What theoretical tools are needed to simplify mba expressions? https://blog.quarkslab.com/what-theoretical-tools-are-needed-to-simplify-mba-expressions.html, 2015.

[29] Yongxin Zhou and Alec Main. Diversity via code transformations: A solution for ngna renewable security. *NCTA-The National Show*, 2006.

[30] Yongxin Zhou, Alec Main, Yuan X Gu, and Harold Johnson. Information hiding in software with mixed boolean-arithmetic transforms. In *International Workshop on Information Security Applications*, pages 61–75. Springer, 2007.

[31] Chris Domas. Movfuscator - the single instruction c compiler. https://github.com/xoreaxeaxeax/movfuscator.

[32] Stephen Dolan. mov is turing-complete. *Cl. Cam. Ac. Uk*, pages 1–4, 2013.

[33] Jacob Baines. *Programming Linux Anti-Reversing Techniques*. Leanpub, 2016.

[34] Dave Barry. A whirlwind tutorial on creating really teensy elf executables for linux. https://www.muppetlabs.com/~breadbox/software/tiny/teensy.html.

[35] Nikos Mavrogiannopoulos, Nessim Kisserli, and Bart Preneel. A taxonomy of self-modifying code for obfuscation. *Computers & Security*, 30(8):679–691, 2011.

[36] PTrace - Linux API. https://linux.die.net/man/2/ptrace.

[37] Strace, Linux syscall tracer. https://strace.io/.

[38] Ltrace, Debugging program to track runtime library calls in dynamically linked programs . https://gitlab.com/cespedes/ltrace.

[39] Dang Hoang Vu Nguyen Anh Quynh. Unicorn – The Ultimate CPU Emulator. http://www.unicorn-engine.org.

[40] Microsoft Research. The Z3 Theorem Prover. https://github.com/Z3Prover/z3.

[41] Wikipedia. Return-oriented programming. https://en.wikipedia.org/wiki/Return-oriented_programming.

[42] Fabrizio Biondi, Sébastien Josse, and Axel Legay. Bypassing malware obfuscation with dynamic synthesis, 2016.

[43] Fabrizio Biondi, Sébastien Josse, Axel Legay, and Thomas Sirvent. Effectiveness of synthesis in concolic deobfuscation. *Computers & Security*, 70:500–515, 2017.

[44] Remis Balaniuk. Drill and join: A method for exact inductive program synthesis. In *International Symposium on Logic-Based Program Synthesis and Transformation*, pages 219–237. Springer, 2014.

[45] Susmit Jha, Sumit Gulwani, Sanjit A Seshia, and Ashish Tiwari. Oracle-guided component-based program synthesis. In *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*, pages 215–224. ACM, 2010.

[46] Henry S. Warren. *Hacker's Delight*. Addison-Wesley Longman Pub-lishing Co., Inc., Boston, MA, USA, 2002.

[47] Henry Massalin. Superoptimizer: a look at the smallest program. In *ACM SIGARCH Computer Architecture News*, volume 15, pages 122–126. IEEE Computer Society Press, 1987.

[48] Torbjo rn Granlund and Richard Kenner. Eliminating branches using a superoptimizer and the gnu c compiler. *cal*, 5(1):r4, 1992.

[49] Eric Schkufza, Rahul Sharma, and Alex Aiken. Stochastic superoptimization. In *ACM SIGPLAN Notices*, volume 48, pages 305–316. ACM, 2013.

[50] Superoptimization Feasibility Study by Embecosm Limited, supported by Innovate UK. http://superoptimization.org/wiki/Superoptimizing_Compilers.

[51] Jonathan Salwan, Sébastien Bardin, and Marie-Laure Potet. Symbolic deobfuscation: From virtualized code back to the original. In Cristiano Giuffrida, Sébastien Bardin, and Gregory Blanc, editors, *Detection of Intrusions and Malware, and Vulnerability Assessment*, pages 372–392, Cham, 2018. Springer International Publishing.

[52] Christopher Kruegel, William Robertson, Fredrik Valeur, and Giovanni Vigna. Static disassembly of obfuscated binaries. In *USENIX security Symposium*, volume 13, pages 18–18, 2004.

[53] Sébastien Bardin, Robin David, and Jean-Yves Marion. Backward-bounded DSE: targeting infeasibility questions on obfuscated codes. In *2017 IEEE Symposium on Security and Privacy (SP)*, pages 633–651. IEEE, 2017.

[54] SQLAlchemy - The Python SQL Toolkit and Object Relational Mapper. https://www.sqlalchemy.org/.

[55] Pickle - Python object serialization. https://docs.python.org/3/library/pickle.html.

[56] Mistreating Triton. https://blog.quarkslab.com/mistreating-triton.html.