QSynth - A Program Synthesis based Approach for Binary Code Deobfuscation

Robin David
Quarkslab
rdavid@quarkslab.com

Luigi Coniglio
University of Trento
luigi.coniglio@studenti.unitn.it

Mariano Ceccato
University of Verona
mariano.ceccato@univr.it

***Abstract*—Assessing software robustness became arduous given the broad adoption of obfuscation in the industry and especially in mobile applications and embedded systems. As such, deobfuscation is becoming crucially important. Obfuscation usually concerns either target the control-flow or the data-flow of the program. While standard static and dynamic analyses suffer some shortcomings, Dynamic Symbolic Execution (DSE) turns out to be very effective on control-flow obfuscation. Yet, fewer approaches address issues raised by data-flow obfuscation. Program synthesis appears to be a promising approach to target such obfuscation. We present a generic approach leveraging both DSE and program synthesis to successfully synthesize programs obfuscated with Mixed-Boolean-Arithmetic, Data-Encoding or Virtualization. The synthesis algorithm proposed is an *offline enumerate synthesis primitive guided by top-down breath-first search*. We shows its effectiveness against a state-of-the-art obfuscator and its scalability as it supersedes other similar approaches based on synthesis. We also show its effectiveness in presence of composite obfuscation (combination of various techniques). This ongoing work enlightens the effectiveness of synthesis to target certain kinds of obfuscations and opens the way to more robust algorithms and simplification strategies.**

I. INTRODUCTION

Context: Over the last two decades software obfuscation has gained in popularity as an approach to protect programs against reverse engineering and tampering attempts. Software obfuscation consists in a semantic-preserving transformation of the original instance of a program P, into a "unintelligible" version $\mathcal{O}(P)$, that is harder to analyse and to understand. A large body of obfuscation techniques have been proposed [12] that can roughly be classified into (i) targeting the control flow (conditional branching and loop logic of the code) and (ii) targeting the data-flow (namely variable values of the program). Ultimately obfuscation strongly alters the syntactic aspect of the control-flow logic or the data-flow logic while preserving its semantic aspect. With the goal of measuring the strength of obfuscation solutions, or to facilitate the analysis of obfuscated malware, in parallel, the research community has grown an interest in elaborating novel approaches to defeat obfuscation. In particular, semantic-based techniques have been proposed, like abstract interpretation [15], taint analysis [32] or symbolic execution [14], [5].

Problem: Techniques based on Dynamic Symbolic Execution (DSE) is effective at solving boolean queries like branching condition [27], thus promising to address control-flow related obfuscation. However obfuscation like data encoding [12] or Mixed-Boolean-Arithmetic (MBA) [34] targets the data-flow and produce overly complex expressions from simple constant values or expressions for which DSE hardly applies. Indeed, checking the satisfiability of a boolean condition is easier than encoding a query to find an alternative and semantically equivalent expression.

To overcome this limitation, alternative approaches have been proposed based on program synthesis [7], [6], [21], [26]. The advantage in using program synthesis over other deobfuscation techniques is that the former is less hindered by the semantic complexity of the program under analysis as many of them solely consider the input/output behavior. Moreover it can in theory be applied regardless of the obfuscation technique in use, while other deobfuscation techniques are challenged by heavily obfuscated expressions and usually only applicable to a subset of obfuscation techniques. Deobfuscation approaches based on program synthesis are usually either enumerative [1], [25], SMT-based [21] or stochastic-based [7], [30]. All of them have to deal with an extremely large search space and hardly scale on the program size. Thus the first research question raised is **RQ1**: How to perform deobfuscation at scale by optimizing the search on a large space?

Obfuscation hides the semantic of a program by increasing its syntactic complexity. A whole variety of techniques like opaque predicates, virtualization and CFG flattening can be used to hinder the analysis. While black-box synthesis techniques are not influenced by the syntactic complexity of the program, a number of synthesis-based approaches to deobfuscation involve some analysis of the program [7] to reduce the search space problem (cf. **RQ1**). Although such analysis steps help reducing the synthesis search space, they also violate the black-box property and expose synthesis to the syntactic complexity of the program. Thus the second research question raised is **RQ2**: How to make synthesis resilient to syntactic complexity and especially composite obfuscation?

Goal and Challenge: In this paper we are interested in automatically synthesize expressions obfuscated with MBA, data encoding and virtualization, to recover the original version or a close semantically equivalent one. Yet, the end goal is to produce, as much as possible, an equivalent program that is more readable for a reverse engineer or a software security analyst. As such, the designed analysis should scale and should be resilient to various kinds of obfuscations encountered.

Workshop on Binary Analysis Research (BAR) 2020
23 February 2020, San Diego, CA, USA
ISBN 1-891562-62-2
https://dx.doi.org/10.14722/bar.2020.23009
www.ndss-symposium.org

Our approach: Program synthesis methods work by automatically deriving a program from a given high-level specification, such as its Inputs/Outputs (I/O) behaviour. Most synthesis-based deobfuscation methods consider the obfuscated program as a virtual black-box, thus trying to solve the problem of black-box deobfuscation, which is a harder problem than practical deobfuscation [4]. The biggest limitation of such approaches consists in the search space size, often too big to be exhaustively explored in an efficient manner.

This work is based on the following intuitions:

- Analysis techniques, such as symbolic execution, can be used to reduce dramatically the size of the search space limiting the aforementioned issue by modeling accurately the semantic of instructions;
- Existing synthesis methods and especially stochastic ones spend time deriving the same expressions again and again while they can be computed once and for all;
- By design, DSE thwarts multiple obfuscations like dead-code, self-modification or packing thanks to its dynamic nature. Thus leveraging a synthesis algorithm on this basis allows addressing program obfuscated with composite techniques while focusing for instance, the synthesis on data obfuscation.

Contributions: Based on these intuitions and the results obtained implementing our approach, this paper makes the following contributions:

- We introduce a novel approach combining DSE, data-flow graph extraction and program synthesis leveraging their strength to address certain obfuscation techniques;
- We propose a black-box synthesis method based on an offline enumerative search. This method uses pre-computed lookup tables allowing to encode a subset of the search space and consequently to synthesize expressions in near constant time;
- We propose an expression simplification algorithm, QSynth, applying black-box synthesis, iteratively simplifying complex obfuscated expressions;
- We performed an empirical validation involving comparison with the deobfuscation tool Syntia [7] for which we shows that we supersede it both in accuracy and time (x20 faster).

II. Motivating example

Let's discuss a motivating example which exhibits the difficulty to address MBA obfuscation. Figure 1 shows the original expression and Figure 2 its obfuscated form using the *Encoded-Arithmetic* (EA) [8] of *Tigress* [11]. The obfuscated expression introduces various bit-wise expressions and purposely makes it very hard to deduce the original expression from. Yet, the semantic is preserved.

$$a - (\neg((b \times a) \times b))$$

Figure 1: Original expression.

$$(a \land \neg(-((((b \land a) \times (b \lor a) + (b \land \neg a) \times (\neg b \land a)) \land b) \times (((b \land a) \times (b \lor a) + (b \land \neg a) \times (\neg b \land a)) \lor b) + (((b \land a) \times (b \lor a) + (b \land \neg a) \times (\neg b \land a)) \land \neg b) \times (\neg((b \land a) \times (b \lor a) + (b \land \neg a) \times (\neg b \land a)) \land b)) - 1)) - (\neg a \land (-((((b \land a) \times (b \lor a) + (b \land \neg a) \times (\neg b \land a)) \land b) \times (((b \land a) \times (b \lor a) + (b \land \neg a) \times (\neg b \land a)) \lor b) + (((b \land a) \times (b \lor a) + (b \land \neg a) \times (\neg b \land a)) \land \neg b) \times (\neg((b \land a) \times (b \lor a) + (b \land \neg a) \times (\neg b \land a)) \land b)) - 1))$$

Figure 2: Obfuscated expression, arithmetic encoding.

As the computation does not involve branching condition everything would be performed in a single basic-block making the obfuscation recognizable. Figure 3 shows the obfuscated expression after applying the *Tigress* Virtualization pass [10] (abbreviated VR), further intertwining control-flow and data-flow obfuscation. After applying this virtualization pass it is even harder to determine the function's original semantic by examining it manually.

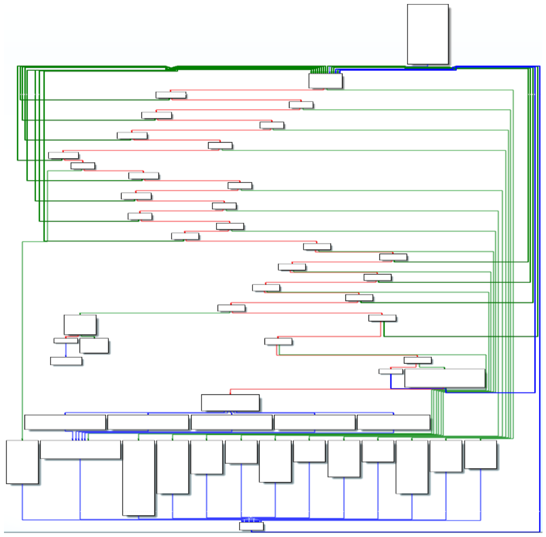


Figure 3: Function CFG computing expression obfuscated using arithmetic encoding and virtualization.

Current approaches based on symbolic execution are able to reconstruct the obfuscated expression after virtualization by carefully keeping track of the dataflow, however they do not offer any way of simplifying the obtained expression. Other approaches based on plain arithmetic simplification (e.g. Z3's `simplify`) works on simple arithmetic properties and mangle the expression to be more easily solved, but not necessarily more readable. On expressions with a complex semantic, `simplify` is then ineffective and it cannot be used for synthesis purposes.

This motivating example is leading the study and we show in this paper how retrieving the original expression is made possible by carefully combining DSE and program synthesis.

III. Background

A. Obfuscation

1) Mixed Boolean-Arithmetic expressions: MBA expressions mix arithmetic operators (ADD, SUB, MUL, etc.) with bit-wise operators (AND, OR, XOR, ROL, ROR etc.). MBA expressions are well-known in the literature, especially in the 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* networks (ARX) [23]. 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". There is little work regarding simplification of MBA expressions [17]. The usage of complex MBA for obfuscation purposes was first formalized by Zhou et al. [33], [34]. In practice any expression can be transformed in an equivalent, as 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 I 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 $f^{-1}(f(p))$.

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

Table I: Example of MBA rewriting rules.

This method can be used to obfuscate statements in a program by replacing them with longer equivalent statements.

2) Virtualization: This technique is used as a way to hide the control-flow of the original program under an additional level of abstraction. It transforms a target function or program in an interpreter of a custom *virtual instruction set* [12], often referred as *Virtual Machine* (VM). The instructions of the original program are translated into the corresponding VM instructions, which are then embedded in the obfuscated program together with the interpreter. Upon execution the interpreter applies a fetch, decode and execute scheme on VM instructions using the appropriate instruction handlers. Depending on the implementation, the VM may make use of a *virtual stack* (such as in the case of *Tigress*) and a number of *virtual registers*. To strengthen obfuscation some VM implementation make use of duplicated instruction codes, repeated instructions handlers running different instructions or multiple nested levels of virtualization.

B. Dynamic Symbolic Execution

DSE, also known as concolic execution [31], is a dynamic data-flow path-based analysis technique. It consists in considering the symbolic values of some or all program inputs, and tracking them during the execution of the program. This allows reasoning on the values that symbolic inputs may take on an execution path in order to solve constraints. Formalized by King [24], it found a renewal of interest the last decade with breakthrough and performances improvements of SMT solvers [16]. During the execution, a symbolic state gets updated according to the semantic of instructions. Working on the semantic makes the DSE sound for the path considered. However, to be complete, DSE would require to cover all paths in a program, which is infeasible in practice.

IV. Synthesis Approach

Our deobfuscation approach involves various steps for which a global overview is shown in Figure 4; we start by tracing an execution of the obfuscated program to obtain an execution trace on which DSE is then performed. For every value of interest, a backward slice is computed on the DSE path predicate to extract the value expression as an Abstract Syntax Tree (AST). The AST of each expression is then forwarded to the synthesis simplification algorithm itself interacting with the Offline Enumerative Synthesis Oracle (denoted $\mathcal{SO}$) that will act as a black-box synthesis primitive. These different steps are defined hereafter.

A. Program tracing

The execution is performed via Dynamic Binary Instrumentation (DBI) that will collect all instructions and their concrete side-effects on registers and memory. All these data are consolidated in an execution trace. More formally, a trace $Tr \triangleq < ins_0, ins_1, \ldots ins_n >$ is defined as a sequence of instructions ins that produces side effects on registers and memory. We denote $\mathbb{C}$ all the concrete states and $C \in \mathbb{C}$ a concrete state of the CPU (registers and memory). Then, $\rightsquigarrow$ is the concrete evaluation operator of an instruction on the concrete state such as evaluating ins_0 on C_0 is denoted $C_0 \overset{ins_0}{\rightsquigarrow} C_1$ with C_1 the concrete state updated. The approach considers a single execution trace of the program. Thus, any obfuscated expression located outside the executed instructions subset will be ignored. Performing program coverage in order to deobfuscate the whole program is beyond the scope of this paper.

B. Dynamic Symbolic Execution

The considered DSE is offline as it is performed as a separate step performed after program execution. In our approach no SMT solver is being used as part of the DSE process (SMT solvers are only used for semantic equivalence checking c.f. VII). The whole synthesis computation is performed on symbolic expressions.

Let $\mathbb{V}ar$ be the set of free variables representing either a register or a memory cell at a given address. Let also $\mathbb{V}al$ be the set of all integer constant values represented as bit-vectors. We can define $\mathbb{C} : \mathbb{V}ar \mapsto \mathbb{V}al$ a concrete memory state mapping variables to constant values. Similarly

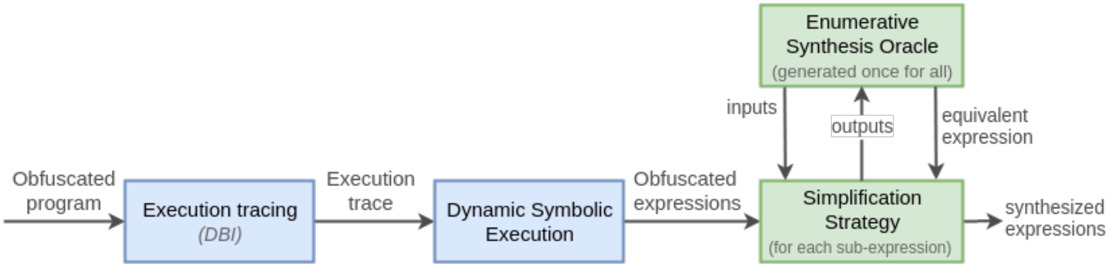


Figure 4: Overview of our deobfuscation approach.

we can define $\mathbb{S} : \mathbb{V}ar \mapsto \Phi$ to be a symbolic state mapping variables to their logical (i.e. symbolic) counterpart. The DSE works on a semantic of instructions which upon execution updates a symbolic state denoted $S \in \mathbb{S}$ that keeps symbolic values for both registers and memory. We similarly denote the symbolic execution as the successive application of the current instruction semantic on the symbolic state denoted $S_n \overset{ins_n*}{\rightsquigarrow} S_{n+1}$ with $\overset{*}{\rightsquigarrow}$ the symbolic evaluation function and n the instruction index in the trace. Symbolic values are defined on the bit-vector theory.

We denote π a symbolic execulogicaltion result, such that $\pi \in \Pi$ with Π the set of all possible symbolic execution paths. We define $\pi \triangleq < S_0, S_1 \cdots S_n >$ the sequence of symbolic memory state updates with $S_{0..n} \in \mathbb{S}$[1].

C. Expression abstract syntax tree computation

We now denote $\rho \triangleq (v, n)$ a slicing criterion defined by $(\mathbb{V}ar \times \mathbb{N})$ indicating to retrieve a logical value expression of $v \in \mathbb{V}ar$ at an offset $n \in \mathbb{N}$ in the trace. Therefore we introduce a backward slicing function $get\_expr : (\Pi \times \rho) \rightarrow \Phi$ that performs a backward dependency lookup to starts from the memory state S_n and recursively finds all logical variables expressions used by v up S_0. The output of $get\_expr(\pi, \rho)$ is an expression formula on bit-vectors noted φ_v^n with the symbolic value of the variable $v \in \mathbb{V}ar$ at trace offset n. Figure 5 depicts a high level view of what get_expr is performing underneath. Starting from the executing path taken (5a) it backtracks on logical dependencies (control and data) (5b) and then only keeps the data flow to create the final AST representation of the expression[2]. Note that, automatically locating and selecting these criteria in the execution trace is out-of-scope of this paper (criteria used for experiments are discussed in VI-B). This logical expression φ_v^n is structured as an AST whose nodes are operators and leaves are constants or variables. These free variables are the inputs of the expression.

We now define $assignment : \langle \mathbb{V}ar \rangle \rightarrow \langle \mathbb{V}al \rangle$ a function that given a vector of variables returns an assignment of values acting as test inputs noted i. This assignment can be deterministic or random. We now can define: $\mathcal{O}_\varphi : \langle \mathbb{V}al \rangle \rightarrow \mathbb{V}al$ the I/O oracle associated with φ_v^n that given a test input returns the associated output constant value after evaluation. A similar oracle can be defined for any sub-expressions (namely sub-AST) of φ_v^n; only the number of variables might vary.

D. Synthesis Oracle

We define $\mathcal{SO} : \Phi \rightarrow \{\Phi \cup \varnothing\}$ a general synthesis oracle function, taking an expression formula as parameter and returning either a new expression formula satisfying some high-level specification (Section IV-E) or no formula if the synthesis failed ($\varnothing$).

Let $I \triangleq < i_0, i_1, \ldots i_k >$ be a test suite of inputs where each i provides an assignment $\mathbb{V}ar \mapsto \mathbb{V}al$. Let also $O \triangleq < o_0, o_1, \cdots o_k >$ be the vector of output values obtained by evaluating an expression φ_v^n on I. $\mathcal{SO}$ is composed of two sub-oracles components:

- $\mathcal{O}_\varphi$: the expression I/O oracle defined hereabove
- $\mathcal{O}_S : \{O\} \mapsto \Phi$ a function mapping a vector of outputs to expressions that by means of evaluation on a test input I produces O.

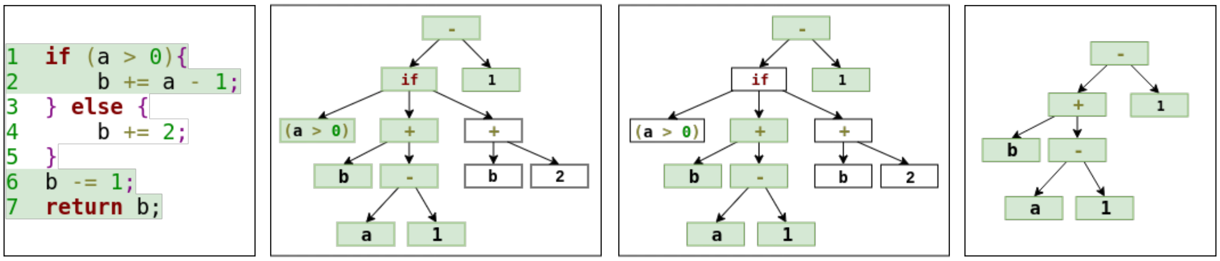
As an example, applying each test inputs of I on the I/O oracle $\mathcal{O}_\varphi$ associated with φ_v^n produces an output vector O_φ. If O_φ belongs to $\mathcal{O}_S$ then $\mathcal{O}_S$ provides φ' that exhibit the exact same I/O behavior (with respect to I) than φ_v^n. The test suite of inputs being finite, the $\mathcal{SO}$ is not sound and may produce expressions which behave differently than the original target (except for the given set of inputs I). The larger the set of inputs, the more successful the synthesis will be. In practice during our experimentation we have verified that even small tests input vectors (∼10-20) appears to be sufficient (c.f VII-C). Section VI-D further illustrates our input values selection strategy In practice any black-box synthesis technique could be used as the $\mathcal{O}_S$ synthesis oracle. We present hereafter our approach based on an *offline enumerative search.*

1) Offline Enumerative Search: The function $\mathcal{O}_S$ mapping output vectors to expression is implemented as an exhaustive search on a context-free grammar $G = \{N, \Sigma, R, S\}$ and making usage of a fixed set of inputs I_k where each k is the size of the input test suite. In G, N is the set of non terminal symbols, Σ is the set of terminal symbols, R is the set of production rules (also known as derivation 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 x input variables. As such, each $i \in I$ should provide a mapping of at least x values so that each terminal variable can get a valuation in i.

Table II illustrates a minimal example of grammar, containing a non-terminal symbol u_8, three terminal symbols (among which two variables a and b) and derivation rules necessary to construct the program represented by the AST in Figure 5d.

[1] The path predicate is implicitly included in the sequence as it is the conjunction of the symbolic values of the instruction pointer register.

[2] For the rest of the paper we use AST and expression interchangeably



```
1  if (a > 0){
2       b += a - 1;
3  } else {
4       b += 2;
5  }
6  b -= 1;
7  return b;
```

(a) Program with path taken.

(b) Dynamic Slicing with $C = (b, 7)$.

(c) Remove control dependencies.

(d) Final expression AST of b.

Figure 5: Expression AST extraction process.

Table III shows the derivation steps needed to obtain the program in Figure 5d.

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 II: Example of grammar G.

Derivation rule	Expression
	u_8
$u_8 \rightarrow u_8 - u_8$	$u_8 - u_8$
$u_8 \rightarrow u_8 + u_8$	$(u_8 + u_8) - u_8$
$u_8 \rightarrow u_8 - u_8$	$(u_8 + (u_8 - u_8)) - u_8$
$u_8 \rightarrow b$	$(b + (u_8 - u_8)) - u_8$
$u_8 \rightarrow a$	$(b + (a - u_8)) - u_8$
$u_8 \rightarrow 1$	$(b + (a - 1)) - u_8$
$u_8 \rightarrow 1$	$(b + (a - 1)) - 1$

Table III: Derivation steps used to obtain the expression 5d using grammar in Table II.

Then, we exhaustively generate all terminal (i.e., containing only terminal symbol) expressions up to a defined number of derivations [3]. Each expression φ is then evaluated for all test suites i yielding a vector of outputs O. Then, a new entry $O \mapsto \varphi$ is added to the $\mathcal{O}_S$ mapping. If the same vector is already in the mapping, we only keep the existing one. Indeed, the enumerative exploration is performed as a breath-first-search, thus, all entries already in the mapping are guaranteed to be more compact than the candidate ones. The mapping ensures an optimal solution since only the first expression corresponding to the given output vector is kept.

Table IV shows an example of the mapping generated from the grammar in Table II with three derivations at most[4]. Here, each $i \in I_k$ contains two values (i.e., $x = 2$), the first one is always mapped to variable a while the second is mapped to b[5]. Expressions are generated from top to bottom and grayed expressions are ignored as they produce the same output than previously computed entries. In practice this mapping function $\mathcal{O}_S$ is implemented as a lookup table (i.e. a dictionary).

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

Table IV: Example mapping table generated with G.

This exhaustive search is costly and its complexity largely depends on the grammar used with the number of terminal symbols and operators. However the insight is that this cost is affordable for the following three reasons:

1) This enumerative search is performed once. Then, all expressions that will be synthesized afterward will take advantage of this pre-computation. This is the “offline” aspect of the search. Other approaches like Syntia [7] have to perform similar derivations for all expressions they intend to synthesize.
2) Various optimizations on the search and derivations allows reducing the complexity. For example, it is possible not to perform some derivations thanks to the commutative property of operators or given the property that applied on the identity, result is known in advance (e.g., $a \oplus a = 0$, $a \wedge a = a$, etc.).
3) The mapping of large grammar can be approximated by using multiple mappings obtained from smaller subsets of the target grammar. This technique is an easy and convenient way to tune synthesis precision.

Finally, the last key intuition is that, it is not mandatory to derive very large expressions, as it all depends on the sim-

[3] The ideal number of derivations depends on the complexity of expression to synthesize, which is however concealed by of obfuscation. As a general rule a higher number of derivations results in higher chances of successful synthesis.

[4] Note that the table does not contain any two-derivations expression as it not possible to create such terminal expression using the considered grammar.

[5] In practice only one mapping is necessary since the derivation step will already generate all possible combinations of variables.

plification strategy that will use this synthesis oracle. It is the role of the simplification strategy to replace sub-expressions such as by recursion the original expression will be recovered. Next section discusses a candidate strategy to achieve such simplification.

E. Expression simplification

This section covers the global simplification strategies leveraging the $\mathcal{SO}$ synthesis primitive to simplify AST of expressions via synthesis. Given an expression φ the naive approach would be to run a direct synthesis of the whole expression. However for expressions encoding complex behaviors it is unlikely that an entry of $\mathcal{O}_S$ would have the exact same output vector O. Thus the general idea is to iterate the expression and use $\mathcal{SO}$ to synthesize and replace parts separately, if not all sub-expressions. In practice this simplification paradigm (e.g. divide-and-conquer etc.) can be implemented using a variety of strategies, differing from the order used to visit sub-expressions to the way successfully synthesized sub-expressions are handled. In this work we present and run benchmark on a simplification strategy with the algorithm name QSynth.

The idea of QSynth is iteratively rewriting the expression AST with placeholder variables, until reaching a fix-point when no more substitution takes place (Figure 6). Algorithm 1 in annex describes the QSynth simplification strategy. QSynth takes as input an expression φ to simplify and a synthesis oracle $\mathcal{SO}$ (line 1). It then proceeds to create a mapping holding the associations between synthesized sub-expressions of φ and placeholder variables. At every loop iteration it performs a synthesis step (l.4), namely it traverses all sub-trees using a randomized breadth-first-search (BFS) order (l.16,17) until finding a synthesizable sub-tree (l.18-21). The randomized BFS works by processing all nodes of a given AST depth before getting deeper. Using BFS order ensures that bigger sub-expressions, closer to the root node, are addressed first. Randomization aims at introducing diversity among different executions[6].

Every time a sub-tree of φ is successfully synthesized, its root node is replaced with a new placeholder variable (l.9,10) and stored in the mapping associating placeholder variables to synthesized expressions (l.11). This simplification loop (l.3-12) continues until the whole AST has been reduced to a single placeholder variable. At the end, the final AST is reconstructed by interating placeholder variables in reverse (addition) order and recursively substituting all of them with the associated AST (l.13, 25-33). The process is illustrated in Figure 7.

It is important to notice how this simplification by substitution with placeholder variables makes simplification possible even if the target expression contains a bigger number of input parameters than what $\mathcal{O}_S$ can handle. This because Algorithm 1 will summarize sub-expressions containing multiple variables into a single placeholder variable.

Algorithm 1 is guaranteed to terminate, but not to successfully simplify any sub-expressions. In this later case, $\mathcal{SO}$ returns $\varnothing$, indicating that it has not been able to fully synthesize the AST. The number of steps needed for synthesis grows with the size of the expression counted as AST nodes, noted $\|\varphi\|$. As every synthesis step traverses the expression from top to bottom until a suitable node has been found, the worst case time-complexity of our algorithm is $O(n^2)$ where n is the number of nodes in the expression AST. However, QSynth 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).

V. Implementation

The synthesis engine is implemented as part of bigger dynamic analysis framework called *QTrace* [13] developed at Quarkslab. This framework is architectured on the following two main components:

- QBDI [20] (version 0.7.0), a Dynamic Binary Instrumentation (DBI) framework which allows configuring the instrumentation and working on multiple architectures x86, x86_64, ARM and ARMv8 (internally) and also multiple platforms Linux, Windows, Mac OS, Android (internally);
- *Triton* [29] (version 0.7), a DSE engine written in C++ performing the dynamic symbolic execution for various architectures x86, x86_64, ARMv8 and ARMv7 (on-going).

Both of these core components are open-source. Based on these, *QTrace* works by collecting execution traces with QBDI which gather instructions executed along with their concrete state $\mathbb{C}$ namely register and memory values. The trace is then stored in an SQL database for which *QTrace* exposes a Python API. This API allows implementing all the subsequent analyses.

In the context of this work, an execution trace is processed by *Triton* which performs the DSE using trace data stored in the database. At any point of the execution it is then able to compute the backward slice/data dependency of a register or memory cell in order to retrieve its AST. The synthesis engine processes the generated expression AST and performs the synthesis as presented in Section IV-E.

VI. Experimental Setup

The goal of this section is to evaluate how our approach compares to similar synthesis-based deobfuscation approaches, namely Syntia, and also how does the two strategies compares to one-another. Similarly to Syntia, benchmarks have been performed on Tigress 2.2. All the tests were performed on a laptop with an Intel Core i7-6700HQ CPU running at 2.60GHz along with 16GB of RAM.

[6] The success of the simplification also depends on the node processing order. Therefore introducing order diversity might help synthesizing sub-expressions that were not simplified at a first iteration but that can be simplified using a different order.

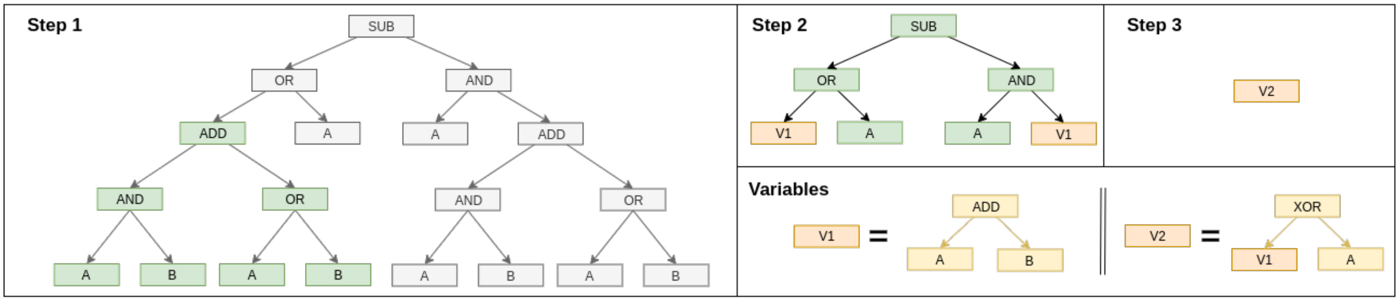


Figure 6: AST iterative synthesis and rewriting.

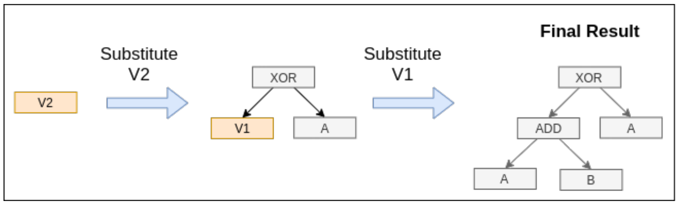


Figure 7: AST's placeholder variables unrolling.

A. Datasets

For the experiments, we use four different datasets consisting in 500 obfuscated functions each. The first one is the same data set used by Syntia authors. However, we hardly managed to get Syntia working, thus results are not experimentally replicated by us, but those reported in their paper. On other datasets, Syntia never gave interesting results or did not terminate in a reasonable amount of time. Thus, only results on the first dataset appeared to be conclusive and fair (cf. Section VII-A). The three other datasets are expressions obfuscated with combination of different obfuscations and aims at comparing the simplification strategy. Original expressions to obfuscate were generated automatically with the same grammar than Syntia that includes up to 3 variables and a variety of 8 operators including $(+, -, \times, \oplus, \vee, \wedge, \sim, -_{(unary)})$. The datasets are built in such a way that expression variables are sent through function parameters and the result of the evaluation is the return value.

a) Dataset 1: created by Syntia authors, the 500 randomly generated functions were obfuscated with the *EncodeArithmetic* [8] and *EncodeData* [9] transformation. The first obfuscation technique hides the original expression using MBA equivalent expressions, while the second transformation encodes all integer parameters before calling the function and decode them on the return site. Original expressions were generated deriving from 3 to 5 times the expression. Thus, average expression size is 3.97 nodes while its obfuscated version is approximately 140 nodes.

b) Dataset 2: is a custom dataset only obfuscated with the *EncodeArithmetic* transformation, but the original expressions were derived from 6 to 21 times (Section IV-D), yielding in average expression of 13.4 nodes. Obfuscated expressions were in average 246 nodes, thus almost twice more complex than Dataset 1.

c) Dataset 3: also uses expression of dataset 2, but obfuscated with *Virtualize* and *EncodeArithmetic* transforms. As shown with the motivating example in Figure 3, virtualization strongly alters both control and data flow.

d) Dataset 4: Use the same original expressions than Dataset 2 but obfuscated with *EncodeArithmetic* and *EncodeData* passes. Due to the complexity induced by this obfuscation, only the first 239 functions of the program were considered in this dataset. Expressions are significantly bigger than other datasets by pushing their complexity to some extremes.

	Trace	Mean	**Mean size** φ *(in node)*		
	len	fun.len*	Orig	Obf$_S$	Obf$_B$
#1: Syntia	77K	148	3.97	139.08	203.19
#2: EA	48K	90	13.5	245.81	131.56
#3: VR-EA	582K	1156	13.5	/	443.64
#4: EA-ED	823K	3369	13.5	6176.89	9223.46

* Mean instruction length of function symbolically executed

Table V: Dataset statistics information.

Table V summarizes the global statistics of the four datasets. The first two columns give the complete execution trace length and the average instruction size of functions symbolically executed as the symbolic execution is performed on a per function basis. Remaining columns shows the mean size of expressions for the original one, the obfuscated one at source-level and the one obtained after compilation and symbolic execution. The three last datasets uses the same original functions but obfuscated with different schemes. For VR-EA it is not possible to establish the obfuscated expression size, as it is splitted in various basic-blocks. At this obfuscation, trace length significantly increases as many control-flow overhead is introduced by the VR obfuscation.

B. Slicing criterion

Similarly to Syntia dataset, obfuscation hides the function return value computation. For each function, the slice is then automatically performed on the `rax` register at the `ret` instruction with an unbounded backward lookup $\rho \triangleq (\texttt{rax}, loc_{ret})$. In practice, the implementation uses *Triton* functionalities to do it and iterates the path predicate in SSA (Static Single Assignment) starting from the register or memory cell of interest. Besides benchmarks, the criterion highly depends on the kind of code being examined, thus, as a general approach, the choice has to be made manually.

	Mean expr. size			Simplification			Mean scale factor			Sem.	Time			
	Orig	Obf$_B$	Synt	∅	Partial	Full	Obf$_S$/Orig	Synt/Obf$_B$	Synt/Orig		Sym.Ex	Synthesis	Total	per fun.
Syntia	/	/	/	52	0	448	/	/	/	/	/	/	34 min	4.08s
QSynth	3.97	203.19	3.71	0	500	500	x35.03	x0.02	x0.94	500	1m20s	15s	1m35s	0.19s

Orig, Obf$_S$, Obf$_B$, Synt are respectively original, obfuscated (source, binary level) and synthesized expressions

Table VI: Benchmark 1 results (Syntia benchmark).

C. Benchmarks

The four dataset are grouped in two benchmarks. The first aims at comparing Syntia against our approach. The second benchmark groups dataset 2, dataset 3 and dataset 4 in order to compare the simplification strategy. A last benchmark experiment aims at evaluating the influence of the lookup table $\mathcal{O}_S$ generation parameters on the synthesis accuracy. The sizes of execution traces are ranging from a few instructions to more than 2000 for virtualized functions.

D. Synthesis lookup tables presetting

For our experimental evaluation we used 14 lookup tables, each generated using a random, yet not repeating, sub-grammar of five operators and three variables. Expressions are exhaustively derived up to five non-terminal derivations, followed by all possible combination of terminal derivations. Each table contains in average 344,933 expressions.

Each expression is evaluated on an input vector of 15, 64-bit integer inputs for the each of the 3 variables. Indeed, larger the inputs vectors, more likely the output vector will be unique. The choice of input values plays an important role during synthesis. For our experiment inputs values are generated uniformly randomly, but favoring values (1, 0, -1) which reveal the idiosyncratic behaviors of expressions and help distinguishing otherwise similar expressions.

The choice of these settings is discussed in Section VII-C. Tables generated *(as Python pickle objects)* use in average 70 MB on disk and were generated in a matter of few minutes. The bottleneck is not the generation time but the memory, as expressions are growing exponentially in memory. Tables are generated in a matters of minutes reaching easily depth of 5 (∼11 nodes expressions).

E. Metrics

a) Success rate: This metric evaluates whether some simplification took place or not. Among simplified expression denoted "partial synthesis", it measures the one that are equal or smaller than the original expression "full synthesis". The AST node count is used as unit. Results does not take in account the correctness of the returned expression.

b) Correctness: Our approach is unsound as output expressions are constructed in such a way that it does not necessarily behave identically to the target obfuscated expression for all possible inputs (but only for the I/O pairs considered). However it is possible to formally check the semantic equivalence of the synthesized expression by the mean of an SMT solver.

c) Execution time: In order to assess the scalability and the practical applicability of the simplification algorithm, benchmarks includes both the DSE time and the synthesis time of the simplification routine on every obfuscated functions.

d) Understandability: The main goal of deobfuscation is to make the code more comprehensible for a reverse engineer or an analyst. There is no predefined metrics encompassing this aspect. However, for benchmark we evaluates the understandability by evaluating the size reduction factor of the obfuscated expression against the synthesized one. This metric is based on the insight that, even if not completely simplified, the more the expression is reduced the more it is understandable.

VII. Benchmark results

A. Benchmark 1: Syntia benchmark

This benchmark aims at assessing our synthesis simplification approach (QSynth) against the state-of-the-art tool in this domain, Syntia. Table VI shows the results. Our algorithm manages to synthesis all the expressions with 100% semantic accuracy in 1m35s. Syntia takes 34 minutes to terminate and it misses the synthesis of 52 functions. QSynth spends in average 0.19s per functions which makes it 20 times faster on this benchmark. The mean expression size after synthesis is 3.71 thus even more compact than original expression which mean size was 3.97. This is explained by the fact that some expressions can be reduced to constant values. The mean reduction scale with the original expression is then 0.94 and 0.02 with obfuscated expressions. This means synthesized expressions are more than 50 times smaller than their obfuscated counterpart. This constitutes a significant speed-up and improvement over Syntia [7] where 448 expressions out of 500 were deobfuscated on the first run. They managed to reach 495 out of 500 expressions after nine runs of Monte Carlo Search Tree black-box synthesis[7].

B. Benchmark 2: Scale benchmark

This benchmark aims at assessing the scale of the QSynth on larger expressions and in presence of composite obfuscations. Thus, functions are significantly bigger. The first impact is on the execution trace ranging from 48,838 instructions to 823,078 for EA-ED. Table VII shows the results on the three datasets EA, VR-EA and EA-ED. The two first successfully simplify all expressions and achieve a full synthesis for 3/4 of them. Synthesized expressions are less than two times bigger than to original expressions, which is a considerable improvement considering the obfuscated size. To illustrate that, Figure 8 shows for EA the node size *(in y)* of obfuscated expressions, the expressions after symbolic execution, the

[7] this first run took ∼34 minutes, thus extrapolating to ∼5h for the 9 runs

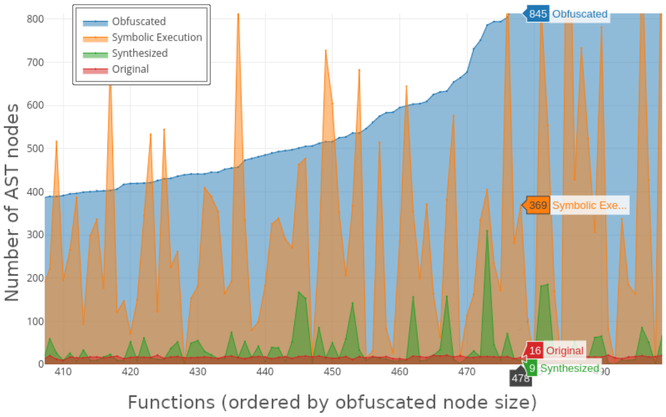


Figure 8: Expressions sizes comparison of each functions on EA benchmark (with functions on x, and their expression sizes in y. Functions are ordered by size of their obfuscated expression).

synthesized and the original expression sizes. Expressions *(in x)* are sorted by obfuscated sizes[8]. The area below the curve represents the gain in nodes number that have been reduced by synthesis. Thus even though an exact synthesis is not obtained for each function, the overall complexity of expressions is strongly reduced. As shown by the scale factor, synthesized expressions are 0.17 and 0.06 times the size of the obfuscated expressions.

For the VR-EA, an appreciable side-effect of the DSE is that it thwarts the control-flow obfuscation added virtualization. Indeed, by symbolizing only function parameters all internal states used by the virtual machine will be ignored in the synthesized expression. However, it has a noticeable impact on the symbolic execution time (17m10s), which is significantly higher than the synthesis time. The two other datasets are spending more time synthesizing.

For EA-ED, given the blow-up of obfuscated expressions, the algorithm still manages to divide by almost two the size of synthesized expressions, lowering the scale factor ratio to x234.44. Also, more than half of functions managed to be fully synthesized.

C. Benchmark 3: Lookup table presetting

This benchmark aims at assessing the tables parameters in order to maximize the synthesis correctness while minimizing input vectors. Figure 9 shows the soundness evolution as the number of input parameters increases. It has been tested on the 500 functions of the dataset 2. For each input vector size, 5 tables were generated with 5 operators. On the figure, semantically correct expressions are shown in green, incorrect expression in red. No timeout was set on the query solving. Results show that the correctness of our approach is directly proportional to the number of inputs used. A vector of 15 inputs gives a reasonable trade-off between input vector size and correctness.

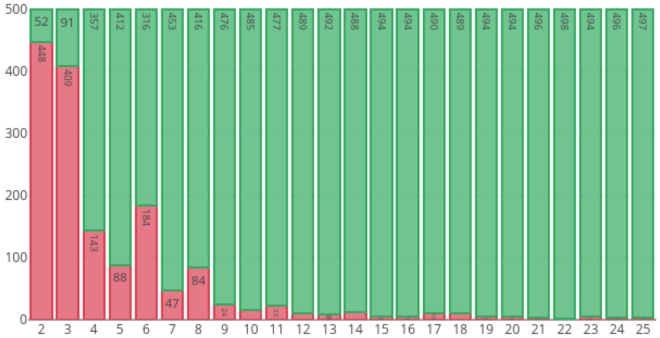


Figure 9: synthesis semantic accuracy of all 500 functions in y, w.r.t. input size in x. (semantic equivalence verified in green and respectively invalid in red)

VIII. Related Work

Program Synthesis is a general concept. Thus, it can be put in practice in a variety of manners. Three approaches have been favored in the litterature: enumerative search, stochastic search and SMT-based approaches. In all cases the synthesis faces the problem of intractability of the program space and the diversity of possible program inputs. The approach of this paper is enumerative. As such, Alur et al. [1] propose an enumerative approach aiming at finding expressions that behave correctly on a subset of I/O pairs as well as *distinguishing predicates* for those subsets. The expressions are then organized in a decision tree. The key limitation of this approach is the assumption that distinguishing predicates exists, which is not always the case for large and complex arithmetic operations.

Synthesis applied to program optimization is often called *superoptimization*. This approach aims at finding a better performing version of a program f given its original definition. This approach is similar to deobfuscation only the *cost function* varies. Superoptimization considers running performance while deobfuscation consider compactness and understandability. The concept of *superoptimization* was first introduced by H. Massalin [25], who proposes a superoptimizer based on exhaustive search over a selected subset of machine's instructions. In their work, two optimization methods are used to reduce search time. The first 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 consists in skipping redundant instructions patterns during the search. Other superoptimization approaches discuss the generation of Peephole superoptimizers [3] or to use SMT-based approach like Souper [28]. Schkufza et al. [30] introduced *STOKE* stochastic approach based on Monte Carlo Markov Chain method, capable of exploring the space of possible programs faster than previous approaches, but without guarantees of optimality.

Program synthesis for deobfuscation. Applied on deobfuscation matter, the Jha et al. technique [21] involves transforming the synthesis problem into a satisfiability problem by selecting a list of base operations, called *components*, and encoding in a formula the space of all possibly generated programs as well as "well-formedness" and behavioural con-

[8] only a slice of all functions is shown on the figure

	Mean size	Simplification			Mean Scale factor			Semantic	Time			
	Synt	∅	Partial	Full	Obf$_S$/Orig	Synt/Obf$_B$	Synt/Orig		Sym.Ex	Synthesis	Total	per fun.
Dataset 2 **EA**	21.92	0	500	354 (70.80%)	x18.34	x0.17	x1.64	OK: 413 KO: 4	1m7s	1m42s	2m49s	0.34s
Dataset 3 **VR-EA**	25.42	0	500	375 (75.00%)	-	x0.06	x1.90	OK: 401 KO: 43	17m10s	2m46s	19m56s	2.39s
Dataset 4 **EA-ED**	3812.84	5	234	133 (55.65%)	x405.25	x0.41	x234.44	-	13m18s	2h7m	2h21m	35.47s

Orig, Obf$_S$, Obf$_B$, Synt are respectively original, obfuscated (source, binary level) and synthesized expressions, OK: semantic equivalence verified (KO if not)

Table VII: Benchmark 2 results (Scale benchmark).

straints. An SMT solver is then used to solve the formula and generate a candidate program. This approach focus on automatic generation of programs performing non-obvious bit manipulations and assume only black box oracle access to the target.

Rolles [26] also uses an enumerative program synthesis approach using I/O pairs as oracle backed by an SMT for correctness. The approach inspired by the peephole superoptimizer [3] is called *peephole superdeobfuscation*. He also suggested a template-based program synthesis to address metamorphic code. Biondi et al. [6] propose a deobfuscation approach based on the *drill and join* method for inductive program synthesis first introduced by Balaniuk [2] to deobfuscate obfuscated conditionals using black-box synthesis. This technique iteratively decomposes each bit-component of the target function in its own sub-space and re-enconding the component's function in a simpler form. Their approach has shown to be effective in deobfuscating MBA-obfuscated expressions, however its complexity grows exponentially with the size of the input. Lately, Blazytko et al. [7] proposed Syntia a stochastic search using Monte-Carlo-Tree-Search (MCTS) trying to find programs with equivalent I/O behaviour given set of I/O pairs.

Other deobfuscation approaches. Many other approaches address obfuscation such as virtualization or MBA with other techniques. Salwan et al. [27] demonstrated how *Triton* [29], a Dynamic Symbolic Execution framework, can be used to deobfuscate virtualization obfuscation. The approach works by carefully symbolizing user inputs to distinguish between the instruction of the original non-obfuscated program and those belonging to the VM. Our Synthesis approach leverages this mechanism for the VR-EA dataset.

Multiple deobfuscation approaches are focusing on analytical simplification of MBA expressions from the obfuscated program. These approaches are complementary to our work, in that they can be used as an additional simplification step prior and/or after our deobfuscation strategy (e.g., to refine our result). Biondi et al. [6] suggest an algebraic simplification approach to polynomial MBA expressions. Their approach is aimed at reducing the complexity of polynomial MBA to MBA of degree one, in order to ease deobfuscation or the satisfiability check. However, it only works on a subset of MBA expression having a specific construct. The approach of Eyrolles et al. [18] simplifies expressions via pattern matching and rewriting. The associated tool `SSPAM` works similarly to the *simplify* API of Z3[16] but with a specific focus on MBA. It was able to reduce the obfuscated expressions tested by the authors to 50% of their original size. However the effectiveness of the approach heavily depends on the richness of its database of known patterns.

Another approach based on boolean algebra 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. *Arybo* by Guinet et al. [19] is on of the only simplification tool supporting boolean and arithmetic operations tracking on bit-vectors. It works by constructing a bit-level symbolic representation of a given expression where each bit is normalized using the Algebraic Normal Form (ANF). Results shows the effectiveness of the approach on expressions with a low number of bits but struggles to scale to the size of inputs.

IX. Discussion

Results obtained in this work are preliminary results, yet promising. Larger experiments are planned on other obfuscators like Obfuscator-LLVM [22] and other commercial obfuscators. Some real-world cases are also considered to assess the efficiency of the approach.

The algorithm still needs to be tuned and improved. It is still rather ineffective on encoded-data and is weak on common synthesis issues like synthesizing constants. The latter is the main limitation as MBA makes heavy usage of constants. Combining the synthesis with other algorithm like SSPAM [18] may be beneficial.

The paper does not discuss the problem of locating the obfuscated expression. This is left as future work in order being able to apply a global automated deobfuscation on a whole program.

While *QTrace* and the QSynth algorithm are not open-source, the two main components, namely QBDI [20] and *Triton* [29] are open-source. Moreover, all the datasets and artifacts are available on Github[9].

X. Conclusion

This paper presents an offline enumerative search synthesis primitive combined with a simplification strategy geared for deobfuscation. Experimental results demonstrate the scalability of the approach and suggest that it overcomes cutting-edge approaches like Syntia (cf. **RQ1**). Results also suggest

[9] https://github.com/werew/qsynth-artifacts

resilience in presence of composite obfuscation (combinations) leveraging for instance, the power of DSE to simplify control dependencies introduced by virtualization obfuscation (cf. **RQ2**). These preliminary results are very encouraging to address data obfuscation such as MBA and might lead to more robust algorithms targeting other kind of data obfuscation.

References

[1] R. Alur, A. Radhakrishna, and A. Udupa, "Scaling enumerative program synthesis via divide and conquer," in *Tools and Algorithms for the Construction and Analysis of Systems - 23rd International Conference, TACAS 2017, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS*, 2017, pp. 319–336.

[2] R. Balaniuk, "Drill and join: A method for exact inductive program synthesis," in *International Symposium on Logic-Based Program Synthesis and Transformation*. Springer, 2014, pp. 219–237.

[3] S. Bansal and A. Aiken, "Automatic generation of peephole superoptimizers," in *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems, ASPLOS 2006, San Jose, CA, USA, October 21-25, 2006*, 2006, pp. 394–403.

[4] B. Barak, O. Goldreich, R. Impagliazzo, S. Rudich, A. Sahai, S. P. Vadhan, and K. Yang, "On the (im)possibility of obfuscating programs," *J. ACM*, vol. 59, no. 2, pp. 6:1–6:48, 2012.

[5] S. Bardin, R. David, and J. Marion, "Backward-bounded DSE: targeting infeasibility questions on obfuscated codes," in *2017 IEEE Symposium on Security and Privacy, SP 2017, San Jose, CA, USA, May 22-26, 2017*, 2017, pp. 633–651.

[6] F. Biondi, S. Josse, A. Legay, and T. Sirvent, "Effectiveness of synthesis in concolic deobfuscation," *Computers & Security*, vol. 70, pp. 500–515, 2017.

[7] T. Blazytko, M. Contag, C. Aschermann, and T. Holz, "Syntia: Synthesizing the semantics of obfuscated code," in *26th USENIX Security Symposium, USENIX Security 2017, Vancouver, BC, Canada, August 16-18, 2017*, 2017, pp. 643–659. [Online]. Available: https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/blazytko

[8] C. S. Collberg, S. Martin, J. Myers, and B. Zimmerman, "Documentation for arithmetic encodings in tigress." [Online]. Available: http://tigress.cs.arizona.edu/transformPage/docs/encodeArithmetic

[9] ——, "Documentation for data encodings in tigress." [Online]. Available: http://tigress.cs.arizona.edu/transformPage/docs/encodeData

[10] ——, "Documentation for virtualization transformation in tigress." [Online]. Available: http://tigress.cs.arizona.edu/transformPage/docs/virtualize

[11] ——, "The tigress C Diversifier/Obfuscator," http://tigress.cs.arizona.edu/.

[12] C. S. Collberg and J. Nagra, *Surreptitious Software - Obfuscation, Watermarking, and Tamperproofing for Software Protection*, ser. Addison-Wesley Software Security Series. Addison-Wesley, 2010.

[13] L. Coniglio, "Exploring Execution Trace Analysis," October 2019. [Online]. Available: https://blog.quarkslab.com/exploring-execution-trace-analysis.html[DEANONYMIZING]

[14] K. Coogan, G. Lu, and S. K. Debray, "Deobfuscation of virtualization-obfuscated software: a semantics-based approach," in *Proceedings of the 18th ACM Conference on Computer and Communications Security, CCS 2011, Chicago, Illinois, USA, October 17-21, 2011*, 2011, pp. 275–284.

[15] M. Dalla Preda and R. Giacobazzi, "Semantic-based code obfuscation by abstract interpretation," in *International Colloquium on Automata, Languages, and Programming*. Springer, 2005, pp. 1325–1336.

[16] L. M. de Moura and N. Bjørner, "Z3: an efficient SMT solver," in *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, 2008, pp. 337–340.

[17] N. Eyrolles, "Obfuscation with mixed boolean-arithmetic expressions : reconstruction, analysis and simplification tools," Ph.D. dissertation, University of Paris-Saclay, France, 2017.

[18] N. Eyrolles, L. Goubin, and M. Videau, "Defeating mba-based obfuscation," in *Proceedings of the 2016 ACM Workshop on Software PROtection, SPRO@CCS 2016, Vienna, Austria, October 24-28, 2016*, 2016, pp. 27–38.

[19] A. Guinet, N. Eyrolles, and M. Videau, "Arybo: Manipulation, canonicalization and identification of mixed boolean-arithmetic symbolic expressions," 2016.

[20] C. Hubain and C. Tessier, "Implementing an llvm based dynamic binary instrumentation framework, *in 34th chaos communication congress, lipzeig, germany, december 27-30, 2019*," December 2017. [Online]. Available: https://qbdi.quarkslab.com/QBDI_34c3.pdf

[21] S. Jha, S. Gulwani, S. A. Seshia, and A. Tiwari, "Oracle-guided component-based program synthesis," in *Proceedings of the 32nd ACM/IEEE International Conference on Software Engineering-Volume 1*. ACM, 2010, pp. 215–224.

[22] P. Junod, J. Rinaldini, J. Wehrli, and J. Michielin, "Obfuscator-LLVM – software protection for the masses," in *Proceedings of the IEEE/ACM 1st International Workshop on Software Protection, SPRO'15, Firenze, Italy, May 19th, 2015*, B. Wyseur, Ed. IEEE, 2015, pp. 3–9.

[23] D. Khovratovich and I. Nikolić, "Rotational cryptanalysis of arx," in *International Workshop on Fast Software Encryption*. Springer, 2010, pp. 333–346.

[24] J. C. King, "Symbolic execution and program testing," *Communications of the ACM*, vol. 19, no. 7, pp. 385–394, 1976.

[25] H. Massalin, "Superoptimizer: a look at the smallest program," in *ACM SIGARCH Computer Architecture News*, vol. 15. IEEE Computer Society Press, 1987, pp. 122–126.

[26] R. Rolles, "Program synthesis in reverse engineering, *in no such conference, paris, france, november 19th, 2014*," November 2014.

[27] J. Salwan, S. Bardin, and M. Potet, "Symbolic deobfuscation: From virtualized code back to the original," in *Detection of Intrusions and Malware, and Vulnerability Assessment - 15th International Conference, DIMVA 2018, Saclay, France, June 28-29, 2018, Proceedings*, 2018, pp. 372–392.

[28] R. Sasnauskas, Y. Chen, P. Collingbourne, J. Ketema, J. Taneja, and J. Regehr, "Souper: A synthesizing superoptimizer," *CoRR*, vol. abs/1711.04422, 2017. [Online]. Available: http://arxiv.org/abs/1711.04422

[29] F. Saudel and J. Salwan, "Triton: A dynamic symbolic execution framework," in *Symposium sur la sécurité des technologies de l'information et des communications, SSTIC, France, Rennes, June 3-5 2015*. SSTIC, 2015, pp. 31–54.

[30] E. Schkufza, R. Sharma, and A. Aiken, "Stochastic superoptimization," in *ACM SIGPLAN Notices*, vol. 48. ACM, 2013, pp. 305–316.

[31] K. Sen, "Concolic testing," in *22nd IEEE/ACM International Conference on Automated Software Engineering (ASE 2007), November 5-9, 2007, Atlanta, Georgia, USA*, 2007, pp. 571–572. [Online]. Available: https://doi.org/10.1145/1321631.1321746

[32] B. Yadegari, B. Johannesmeyer, B. Whitely, and S. Debray, "A generic approach to automatic deobfuscation of executable code," in *2015 IEEE Symposium on Security and Privacy*. IEEE, 2015, pp. 674–691.

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

[34] Y. Zhou, A. Main, Y. X. Gu, and H. Johnson, "Information hiding in software with mixed boolean-arithmetic transforms," in *International Workshop on Information Security Applications*. Springer, 2007, pp. 61–75.

Appendix

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

```
Function Simplify($\varphi$: an expression, $\mathcal{SO}$: a synthesis oracle):
    varsMap ← NewMapping()
    while $\varphi$'s root edge is not a placeholder variable do
        $r \leftarrow$ SynthesisStep($\varphi$, $\mathcal{SO}$)
        if $r$ is $\varnothing$ then
            return $\varnothing$
        end
        $(\varphi_\epsilon,\ \varphi'_\epsilon) \leftarrow r$
        $v$ ←NewPlaceholderVariable()
        $\varphi$ ←Substitute($\varphi$, $\varphi_\epsilon$, $v$)
        varsMap$[v]$ ← $\varphi'_\epsilon$
    end
    return UnrollAST($\varphi$, $varsMap$)

Function SynthesisStep($\varphi$: an expression, $\mathcal{SO}$: a synthesis oracle):
    foreach edge $\epsilon$ in $\varphi$ using randomized breadth-first order do
        $\varphi_\epsilon \leftarrow$ sub-tree of $\varphi$ with root $\epsilon$
        $\varphi'_\epsilon \leftarrow \mathcal{SO}(\varphi_\epsilon)$
        if $\varphi'_\epsilon$ is not $\varnothing$ then
            return $(\varphi_\epsilon,\ \varphi'_\epsilon)$
        end
    end
    return $\varnothing$

Function UnrollAST($\varphi$: an expression, varsMap: a mapping of variables to expressions):
    foreach variable $v$ in $\varphi$ do
        if $v$ is in varsMap then
            $\varphi'_\epsilon \leftarrow$ varsMap$[v]$
            $\varphi_\epsilon \leftarrow$ UnrollAST($\varphi'_\epsilon$, varsMap)
            $\varphi \leftarrow$ Substitute($\varphi$, $v$, $\varphi'_p$)
        end
    end
    return $\varphi$

Function Substitute($\varphi$: an expression, $\varphi_t$: a target sub-expression, $\varphi_s$: a substitute expression):
    Substitute all instances of $\varphi_t$ in $\varphi$ with $\varphi_s$
    return $\varphi$
```