

Redefining Grey-Box Deobfuscation through Context-Aware Program Synthesis, and Application to MBA

Vidal Attias
Université Paris-Saclay, CEA, List
Palaiseau, France
vidal.attias@cea.fr

Grégoire Menguy
Université Paris-Saclay, CEA, List
Palaiseau, France
gregoire.menguy@cea.fr

Nicolas Bellec
Université Paris-Saclay, CEA, List
Palaiseau, France
nicolas.bellec@cea.fr

Sébastien Bardin
Université Paris-Saclay, CEA, List
Palaiseau, France
sebastien.bardin@cea.fr

Jean-Yves Marion
Université de Lorraine, CNRS, LORIA
Nancy, France
jean-yves.marion@loria.fr

Abstract

Code obfuscation is widely used for software protection, but it also plays a central role in concealing malware, making automated deobfuscation fundamentally challenging due to the combinatorial explosion of syntactic and semantic complexity. While *black-box* deobfuscation via program synthesis elegantly bypasses syntactic complexity by reasoning purely over input/output semantics, it suffers from fundamental scalability limitations when the underlying semantic complexity grows. On the other hand, *white-box* approaches struggle to cope with the increase of syntactic complexity. The few existing *grey-box* approaches attempt to bridge this gap, but suffer from fundamental limitations that hinder their effectiveness on large and complex obfuscated programs. We argue that these limitations stem from a missing abstraction layer between syntax and semantics. In this paper, we precisely revisit grey-box deobfuscation as the missing abstraction layer between syntax and semantics, enabling scalable deobfuscation by combining structural guidance with observational semantics. We introduce GREYSYNTH, a grey-box deobfuscation framework that automatically lifts any black-box synthesizer into an efficient context-aware simplification procedure. GREYSYNTH relies on two key mechanisms: (i) observational equivalence-driven simplification, which compresses the search space by collapsing semantically redundant sub-expressions, and (ii) context-aware local synthesis, which recovers high-level semantics from truncated program structures while preserving global dependencies. Comprehensive evaluation against state-of-the-art black-box tools, grey-box tools and MBA-specialized deobfuscators shows that GREYSYNTH outperforms them both in terms of compression and speed on most benchmarks, and scales much better across obfuscation levels. Overall, GREYSYNTH demonstrates that combining structural awareness with black-box synthesis enables synthesizer-agnostic, scalable and context-aware deobfuscation, opening a new direction beyond purely black-box or syntax-driven approaches.

CCS Concepts

• Security and privacy → Software security engineering; Malware and its mitigation; • Software and its engineering → Software reverse engineering.

Keywords

Deobfuscation; reverse engineering; mixed boolean-arithmetic; grey-box; program synthesis

ACM Reference Format:

Vidal Attias, Grégoire Menguy, Nicolas Bellec, Sébastien Bardin, and Jean-Yves Marion. 2026. Redefining Grey-Box Deobfuscation through Context-Aware Program Synthesis, and Application to MBA. In *Proceedings of the 2026 ACM SIGSAC Conference on Computer and Communications Security (CCS '26)*, November 15–19, 2026, The Hague, Netherlands. ACM, New York, NY, USA, 15 pages. <https://doi.org/10.1145/3830454.3846695>

1 Introduction

Obfuscations are protection mechanisms that transform a program P into a semantically equivalent program P' , considerably more expensive to reverse engineer [16, 17]. Since cryptography has not yet provided a usable mathematically-proven obfuscation technique, heuristically "hard-to-revert" program transformations are being used [6, 17, 18, 37, 38, 47, 50], opening the door to deobfuscation attacks [4, 12, 28, 34, 42, 45, 49]. Although obfuscation can be used to lawfully protect *Intellectual Properties* as with *Digital Rights Management* (DRM) protections, it is also used by malware to evade analysis [12, 46]. Thus, *deobfuscation* methods have been devised to counter protections [4, 29, 34, 43, 49]. Given an obfuscated version P' of P , the goal is to find an equivalent program P_0 easier to analyze.

White and black-box deobfuscation. Traditionally, obfuscation is attacked using *white-box techniques* [9, 12, 28, 29, 40, 43, 48, 49] that syntactically analyze the code and deduce applicable simplification passes from it. Such methods are particularly powerful as they often target specific obfuscation schemes [5, 9, 12, 19, 35, 43]. However, they also suffered severe setbacks such as *path-oriented protections* [7, 37, 38], *covert channels* [47], or *Mixed Boolean-Arithmetic* (MBA) [30, 50]. These protections exploit the inherent sensitivity of white-box methods to syntactic complexity, which can be increased arbitrarily. To tackle the *white-box* limitations, *black-box techniques* have emerged [4, 11, 34]. They approximate obfuscated code semantics through input-output observations and leverage *Programming*

ACM acknowledges that this contribution was authored or co-authored by an employee, contractor or affiliate of a national government. As such, the Government retains a nonexclusive, royalty-free right to publish or reproduce this article, or to allow others to do so, for Government purposes only. Request permissions from owner/author(s). CCS '26, The Hague, Netherlands

© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-2871-6/2026/11
<https://doi.org/10.1145/3830454.3846695>

by *Example* (PbE) [3, 25, 26] to generate simpler versions of the code. By focusing on input-output observations only, black-box methods are immune to syntactic complexity. Recently, black-box deobfuscation [4] has been successfully used to simplify obfuscated expressions which were beyond the simplification capabilities of specialized *white-box* tools like GAMBA [40] and PROMBA [29].

Problem. In this paper, we highlight that using a pure *white-box* or *black-box* technique is a radical move that suffers inherent limitations. On the one hand, white-box methods struggle to cope with the increase of syntactic complexity. Even dedicated deobfuscation methods using algebraic attacks are impacted [29, 40] as they face head-on the syntactic size explosion induced by obfuscation. On the other hand, black-box methods [4, 11, 34] radically consider that the code syntax is not available and only rely on input-output observations. Hence, they are inherently limited by synthesizer capabilities and suffer from fundamental scalability limitations when the underlying semantics become intricate. Ultimately, black-box synthesis operates as an all-or-nothing mechanism: it either recovers a solution near-instantaneously or stalls indefinitely.

Goal and challenges. We aim to build a new deobfuscation framework, both resilient to syntactic and semantic complexity, by smartly combining *white-* and *black-box* reasoning. However, designing such a *grey-box* approach is highly challenging. A first attempt has recently been proposed [22], but it suffers from fundamental research gaps and is not living up to expectations. Especially, we advocate that the current definition of *grey-box* is too limiting, focusing on the semantics of complete subtrees as targets of *code synthesis*. Since current MBA transformations [15, 50] operate at a local level, we advocate that a new definition of *grey-box* is needed, that encompasses the semantics of local subtrees. More generally, we identify 4 main challenges that appear when combining *white-* and *black-box* methods, namely *Syntactic Complexity Explosion*, *Polluted Truncated Trees*, *Semantic Context Blindness* and *Rigidity in Representation*.

Our Solution. We re-define *grey-box* synthesis as a *context-aware deobfuscation paradigm* that leverages the *global syntactic structure* and *observational behaviors* of a program to *partition complex expressions* into *tractable local synthesis problems*. Furthermore, we propose **GREYSYNTH**, a *grey-box* deobfuscation tool designed to overcome current *grey-box* limitations through a “*synthesis-in-context*” approach. Our solution is synthesizer agnostic and relies on the following two key algorithms:

- **Observational Equality Simplification (OES)** leverages observed input-outputs to identify redundant and constant-valued sub-expressions and replace them with minimal equivalent forms. OES aggressively collapses the AST since the number of unique semantics grows significantly slower than the total syntactic size. It also allows to process each unique semantics only once.
- **Contextual Horizon Simplification (CHS)** simplifies local truncated subtrees of the obfuscated expression with black-box synthesis. CHS uses sampled I/O from the actual program context to identify functional dependencies between nodes. By using these I/O relationships, the synthesizer can “detect for free” dependencies between variables and recover the smallest possible high-level term. Furthermore, CHS overcomes the traditional black-box

struggle with constant values by dynamically seeding its synthesis grammar with constants identified by OES.

Contributions. Our contributions are the following:

- (1) **Conceptual Framework:** We redefine in Section 3 the shift from isolated black-box synthesis to a contextual grey-box strategy that uses AST structural properties to guide local synthesis. Our new definition extends the tool-centric view proposed in prior work to provide a general framework for grey-box deobfuscation. Moreover, we highlight 4 main challenges faced when building grey-box deobfuscation. We argue that solving these challenges is key to get efficient and robust grey-box methods;
- (2) **GREYSYNTH Algorithm:** We present in Section 4 our dual-engine architecture, composed of OES and CHS to efficiently recover heavily obfuscated expressions. This leads to the GREYSYNTH framework: the first principled method that tackles the core challenges of grey-box deobfuscation;
- (3) **Real-world dataset with ground truth expressions:** Current evaluations of deobfuscation often rely on datasets containing only synthetic expressions. This is a major limitation, especially when evaluating grey-box approaches which are affected by both the syntax and the semantics of the code. Hence, we build a new dataset of real-world expressions, extracted from a large corpus of well-known C libraries, and obfuscated with the obfuscators Tigress [15], OMVLL [1] and MBA Obfuscator [36].
- (4) **Experimental Evaluation:** Section 5 extensively evaluates GREYSYNTH against the state-of-the-art black-, white- and grey-box deobfuscators. Experiments show that it highly outperforms all competitors. Especially, it achieves near-perfect simplification on general obfuscated expressions while competitors return expressions 4 to 135 times larger. Moreover, we show that GREYSYNTH is the only solution that scales to extremely complex obfuscations, whereas competitors’ performance breaks down (GREYSYNTH retrieves expressions 2 times larger than the ground truth in the worst cases, whereas competitors’ results are 494 times larger at best).

Overall, we highlight important insights and challenges faced by grey-box methods. We built on them to provide GREYSYNTH, a novel and powerful tool for grey-box deobfuscation. GREYSYNTH shows that combining structural awareness with black-box deobfuscation enables scalable and context-aware deobfuscation, opening a new direction beyond purely black-box or syntax-driven approaches.

2 Background

2.1 Syntax Guided Program Synthesis

Syntax guided program synthesis (SyGUS) [2, 13, 26] aims to automatically generate code from both a syntactic and a semantic specification. The syntactic specification, often called the *domain-specific language (DSL)*, restricts the syntax of the candidate solutions, reducing the search space. On the other hand, the semantic specification expresses the expected behavior of the result code.

Formal synthesis and programming-by-example (PbE). The semantic specification can be given in different format. Formal synthesis [32] considers that the user can provide the specification

as a formula. It can then deduce a solution by reasoning on the given formula. However, providing such a formal specification is often too hard for non-expert users. Hence, *programming-by-example* (PbE) considers that the semantic specification is given as input-output (I/O) examples. These I/Os approximate the expected semantics but are a lot easier to provide for a user. Formally, given an unknown function $t : In \rightarrow Out$, the user is asked to give a set of I/O examples $S \subset In \times Out$, such that, for all $i, o \in S$, $t(i) = o$. A programming-by-example synthesizer then aims to find a function $f : In \rightarrow Out$ with the same behavior, i.e., for all $i, o \in S$, $f(i) = o$.

2.2 Obfuscation and deobfuscation

Obfuscation [16] is a set of methods that transform a program P into a more complex yet equivalent program P_o to slow down the analysis of the program. It is used to lawfully protect intellectual property (e.g., in DRM), but also to prevent the comprehension and detection of malwares [46]. A wide variety of obfuscation methods, based on various program analysis recipes [17], have been proposed [17, 37, 47, 50]. Especially, *Mixed Boolean-Arithmetic* (MBA) [24, 29, 31, 40, 50] has recently particularly drawn the attention of the academic community. MBA is a commonly used obfuscation that rewrites expressions on bitvectors into syntactically more complex yet equivalent ones that combines both arithmetic ($+$, $-$, $\times$) and bitwise ($\neg$, $\wedge$, $\vee$, $\oplus$) operators, e.g., $x + y \rightarrow (x \oplus y) + 2 \times (x \wedge y)$. Usual MBA techniques involve term-rewriting or algebraic transformations based on invertible polynomials or special properties of bitwise operators.

Deobfuscation [4, 33, 34, 43, 48, 49] aims at inverting the obfuscation process, retrieving a simpler equivalent program from an obfuscated one. The study of deobfuscation methods provides new tools for malware analysts and helps evaluate the current obfuscation methods. Different deobfuscation methods exist:

- *White-box deobfuscation* [24, 29, 31, 40, 43, 48] uses program analysis methods (e.g., compiler simplification, symbolic execution) to revert the obfuscations. These methods have been vastly studied and shown to be very efficient in various contexts [9, 12, 28, 29, 40, 43, 48, 49]. Recently, many works emerged to attack MBA obfuscation through specialized white-box deobfuscation [24, 29, 31, 40]. These rely on two main methods: algebraic attacks and formal synthesis. Algebraic attacks such as MBA-Blast [31] and Gamba [40] use knowledge about the obfuscation method to reduce the expression in a known basis. Such algebraic attacks are completed in ProMBA [29] with formal synthesis to infer rewriting rules. While powerful, such methods are inherently sensitive to syntactic complexity, and thus are vulnerable to anti-white-box deobfuscation [37, 47, 50].
- *Black-box deobfuscation* [4, 11, 34] is a recent deobfuscation framework that emerged as an answer to anti-white-box deobfuscation. It relies on the fact that obfuscations only increase the syntactic complexity while providing a semantically equivalent program. For example, $x + y$ can be rewritten as $(x \oplus y) + 2 \times (x \wedge y)$, but both versions share the same I/O behaviors. Black-box deobfuscation exploits this weakness to completely bypass obfuscations. It runs the obfuscated expression e over a bounded set of sampled inputs and collects the resulting outputs. It then builds

a set of input-output (I/O) examples to feed a PbE synthesizer, which returns a simple version of e . Relying on I/O observations only makes black-box deobfuscation immune to the syntactic complexity and anti-white-box methods. However, it cannot deobfuscate semantically complex expressions [34, 44].

- *Grey-box deobfuscation* [20, 22] aims at combining structural traversal with semantic reasoning to mitigate both syntactic and semantic complexity. QSYNTH is the only proposal in this direction. It relies on a precomputed cache that maps input/output behaviors to expressions, built by enumerating a fixed grammar and retaining only semantically distinct candidates. Given a sub-expression e , QSYNTH evaluates it on sampled inputs and tries to match the observed behavior in the cache. If a match is found, e is replaced by the mapped expression. While efficient, QSYNTH suffers from key limitations: the grammar is fixed at precomputation time, the cache size grows exponentially with grammar richness, and simplification is limited to previously enumerated expressions, preventing the discovery of new simplifications.

In practice, deobfuscation methods work on a *reverse window*, i.e., the subset of the code that includes the obfuscation and on which analysis will be performed. It can be provided by hand or automatically [14]. Typical reverse windows include execution traces [9, 22], functions [33], basic blocks [4] or well-delimited expression [29, 31, 34, 40]. Choosing a good reverse window is crucial for efficient deobfuscation. Especially, black-box methods are brittle to small reverse-window mistakes, which may increase semantic complexity. White- and grey-box methods can be more flexible, relying on the syntax to focus on interesting sub-windows. From a well-defined reverse window, deobfuscators return a simplified version of it, which can be used for further analysis.

3 Grey-box Deobfuscation and its Challenges

Existing definitions of grey-box deobfuscation remain informal and tool-centric. In particular, prior work [22] characterizes grey-box as an offline enumerative synthesis guided by AST-based semantic search, as instantiated in QSYNTH. While seminal, this definition is both restrictive and tied to a specific implementation strategy. At the same time, recent approaches such as ProMBA [29], although typically classified as white-box due to their reliance on symbolic reasoning, formal synthesis and algebraic simplifications, implicitly follow a similar intuition: they decompose complex expressions into smaller sub-problems using structural truncation and apply synthesis locally. This suggests that the essence of grey-box reasoning goes beyond existing categorizations and needs a conceptual step ahead to become fully effective.

Conceptually, we propose a new, more general framework for grey-box deobfuscation. It includes the first clear definition of the grey-box setting and the first identification of its fundamental algorithmic challenges. This framework will be particularly important in Section 4 for devising GREYSYNTH, the first grey-box deobfuscator that addresses the aforementioned challenges.

Towards a principled definition. To clarify our perspective, Table 1 compares white-box, black-box, and grey-box deobfuscation along their core principles, use of syntax and semantics, and problem-solving strategies. In this work, we propose to define

grey-box deobfuscation as follows: *Grey-box deobfuscation is a decomposition paradigm that leverages program structure to partition a semantically complex expression into a set of tractable synthesis problems, solved using observational semantics.*

This definition highlights three essential properties:

- (1) **Structural guidance.** Unlike black-box approaches, which ignore syntax, grey-box methods exploit the program's structure to guide the decomposition of the problem.
- (2) **Observational reasoning.** Unlike white-box approaches, grey-box methods do not rely on symbolic reasoning or equivalence checking as a primary mechanism, but instead approximate semantics through input/output observations.
- (3) **Decomposition over synthesis.** Grey-box does not aim to synthesize the full expression at once, but rather reduces the global problem into a collection of local synthesis tasks whose complexity is bounded.

This perspective reframes grey-box not as a simple hybrid between white-box and black-box techniques, but as an *abstraction layer* between syntax and semantics: structure is used to organize the search, while semantics is used to validate and simplify it effectively departing from a mere new tree traversal technique.

Table 1: Comparison of deobfuscation paradigms.

	White-box	Black-box	Grey-box
Core	Symbolic	Global synth.	Decomp. + synth.
Task nature	Simplification	Synthesis	Simplification
Syntax access	Full	None	Full
Semantics access	Exact	I/O	I/O
↳ SMT checks	Allowed	Impossible	Intractable
Strength	Semantics	Syntax-agnostic	Balanced
Limitation	Syntax blowup	Semantic scale	Decomp. limits

Grey-box challenges. We now highlight four main challenges faced when combining white- and black-box methods. We advocate that solving these challenges is central to get the best of both worlds:

- (1) *Syntactic Complexity Explosion*: Obfuscation inflates the AST size by introducing deeply nested and redundant structures. Grey-box approaches, which operate by traversing the AST and applying local synthesis, must therefore process a number of sub-expressions proportional to this inflated size. Since synthesis is computationally expensive, it leads to a combinatorial explosion in the number of synthesis attempts, making the approach intractable on large obfuscated expressions;
- (2) *Polluted Truncated Trees*: A major hurdle occurs when the synthesis horizon captures a "polluted" semantic. Because the horizon is fixed, it often includes fragmented bits of obfuscated sub-expressions that are not part of the underlying rewrite rule (e.g., instead of $A + B$, the synthesizer has to find $A + (x|y)$ because B was truncated mid-expression). Existing tools fail because they attempt to synthesize this fragmented logic directly, leading to a semantic complexity that quickly exceeds the synthesizer's reach.
- (3) *Semantic Context Blindness*: By anonymizing sub-trees using symbolic variables or fresh inputs, prior works fail to capture

contextual bias (e.g., invariants or functional dependencies). Hence, preventing aggressive, context-valid simplifications.

- (4) *Rigidity in Representation*: Approaches relying on static caches lack the flexibility to adapt grammars on-the-fly, failing to recover complex arithmetic like multiplication or handle "notoriously hard" nested constant values.

These challenges are general, affecting known grey-box methods but also white-box methods that leverage synthesis and black-box simplifiers. For instance, *Syntactic Complexity Explosion* is the key reason why QSYNTH [22] does not leverage full-fledged synthesis but a precomputed cache of expressions. Indeed, QSYNTH attempts to synthesize each node of the AST. Thus, it cannot afford the cost of synthesis. On the other hand, white-box and grey-box synthesis-based simplifiers [22, 29] can anonymize sub-expressions to simplify obfuscation rules. Hence, they are both impacted by the *Polluted Truncated Trees* and *Semantic Context Blindness* challenges (refer to Fig. 1 for more details). Finally, black-box approaches [4, 11, 34] cannot adapt synthesis to the obfuscated code. Hence, they are impacted by the *Rigidity in Representation*, and cannot be used on semantically complex expressions. Interestingly, white- and grey-box synthesis-based simplifiers could adapt their grammar to tackle this last challenge. However, none of them noticed this issue and, as a result, remain vulnerable to it.

Key idea. In this work, we propose a new conceptual approach to grey-box deobfuscation that *leverages black-box synthesis as a building block*, while explicitly exploiting the structure of the program. Our approach is based on a simple principle: *use the structure to guide the synthesis, and the semantics to simplify the structure*. This leads to a form of grey-box deobfuscation that is both scalable and robust to complex obfuscation patterns. GREYSYNTH directly addresses the four previously cited challenges through two main components: *Observational Equality Simplification (OES)* and *Contextual Horizon Simplification (CHS)*. The former (i) prunes redundant semantics in the code to limit *Syntactic Complexity Explosion*; and (ii) gathers useful constant values to integrate them in synthesis, tackling *Rigidity in Representation*. The latter performs local black-box simplifications, using the context gathered by OES and the I/O sample dependencies. Thereby, it tackles the *Polluted Truncated Trees* and *Semantic Context Blindness* challenges.

4 Efficient Grey-box Deobfuscation

In this section, we describe GREYSYNTH, a new grey-box deobfuscation that tackles the previously discussed challenges.

4.1 The GREYSYNTH framework

We now detail the GREYSYNTH framework, designed to efficiently simplify highly obfuscated expressions.

Grey-box deobfuscation problem.. We consider an obfuscated expression $e : BV[m_1] \times \dots \times BV[m_r] \rightarrow BV[m]$, where $BV[n] = \{0, 1\}^n$ denotes the set of bitvectors of size n . The expression is represented as an abstract syntax tree (AST), where internal nodes correspond to arithmetic or boolean operators (including division, modulo, and bitwidth conversion operators), and leaves correspond to either input variables $x_1, \dots, x_r$ or constants. The size of an expression e , denoted $|e|$, is defined as the number of nodes in its

Figure 1: Example of Polluted Truncated Trees and Semantic Context Blindness challenges

Obfuscated expression: $(5x \oplus 6y) + 2 \times (5x \wedge 6y) \equiv 5x + 6y$		
Horizon	Anonymized sub-exprs	Synthesis query
$h = 2$	$\alpha_1 : (5x \oplus 6y); \alpha_2 : 2 \times (5x \wedge 6y)$	$\alpha_1 + \alpha_2$
$h = 3$	$\alpha_1 : 5x; \alpha_2 : 6y; \alpha_3 : (5x \wedge 6y)$	$(\alpha_1 \oplus \alpha_2) + 2 \times \alpha_3$
$h = 4$	$\alpha_1 : 5x; \alpha_2 : 6y$	$(5x \oplus 6y) + 2 \times (\alpha_1 \wedge \alpha_2)$
$h = 5$	$\emptyset$	$(5x \oplus 6y) + 2 \times (5x \wedge 6y)$

The expression is an obfuscated version of $5x + 6y$. It exemplifies the *Populated Truncated Trees* and *Semantic Context Blindness* challenges faced by grey-box and synthesis-based white-box simplifiers. These anonymize sub-expressions with fresh variables ($\alpha_1, \dots, \alpha_n$) to simplify obfuscation passes. However, recovering with synthesis $x + y$ from its obfuscated form $(x \oplus y) + 2 \times (x \wedge y)$ is hard. Indeed, it requires anonymizing exactly the correct set of sub-expressions (here all the occurrences of $5x$ and $6y$, which can be themselves obfuscated), and is brittle to any mistake. For instance, the natural solution from [29], consisting of anonymizing subtrees at a given horizon, is not enough. For horizons 2 and 3, synthesis succeeds but returns useless expressions (no simplification). For horizon 4, synthesis cannot be simplified either. Indeed, the truncated tree is polluted with subexpressions irrelevant to the simplification ($5x$ and $6y$). Moreover, semantic context blindness impedes seeing that $5x \leftrightarrow \alpha_1$ and $6y \leftrightarrow \alpha_2$. Finally, at horizon 5, the expression is too complex for synthesis (note: while, as explained, ProMBA cannot infer the simplification, it can simplify it with algebraic attack as the expression is simple enough).

AST (counting operators, variables, and constants). By abuse of notation, we identify a node n of the AST with the sub-expression it represents, and write $|n|$ for the size of this sub-expression. We consider a bounded set of sampled inputs $\mathcal{I} \subseteq BV[m_1] \times \dots \times BV[m_r]$. For each variable x_i and sample index j , we denote by x_i^j the value of x_i in the j -th sample of $\mathcal{I}$. Evaluating e over $\mathcal{I}$ induces, for each node n of the AST, an output vector denoted $\vec{o}(n)$.

Given an input expression e , the goal of grey-box deobfuscation is to recover a simpler expression e' that is observationally equivalent to e over the sample set $\mathcal{I}$, i.e., $\forall x \in \mathcal{I}, e(x) = e'(x)$, while minimizing $|e'|$. Since equivalence is only enforced over $\mathcal{I}$, this objective does not guarantee semantic equivalence over the full input space. Still, our evaluation (Section 5) shows that, grey-box results are predominantly correct.

GREYSYNTH framework. Our GREYSYNTH framework is composed of two core components: *Observational Equality Simplification* (OES) and *Contextual Horizon Simplification* (CHS). OES and CHS are designed as a mutually reinforcing pipeline addressing all four problems from Section 3. Given an obfuscated expression e and a black-box synthesizer PbE-Synth, GREYSYNTH returns a simplified version of e . It proceeds as follows:

- (1) **OES.** First, GREYSYNTH leverages OES to exploit global observational relations and compresses the AST in linear time, collapsing syntactic inflation by identifying redundant semantics and eliminating bloated representations (*Syntactic Complexity Explosion*). As a byproduct, it recovers obfuscated constants and establishes a syntactic–semantic correspondence that CHS relies on.

- (2) **CHS.** Second, GREYSYNTH calls CHS to exploit this prepared representation by combining a *white-box* tree traversal with *black-box* program synthesis. It systematically extracts truncated subtrees up to a growing depth, stubbing its pruned nodes with fresh variables (*Polluted Truncated Trees*), then performs synthesis (with S) using contextually sampled values for the new variables, preserving functional dependencies (*Semantic Context Blindness*), and adapting the synthesis grammar by dynamically injecting constant values to each synthesis target (*Rigidity in Representation*).
- (3) **Final OES pass.** Finally, OES is reapplied to capture any new equivalences created by CHS rewrites, ensuring that the output is fully compacted.

Observational equivalence & Fingerprints. All along the deobfuscation, GREYSYNTH approximates expression equivalence with observational equivalence. It is derived from the black-box setting that considers two expressions e and e' equivalent if they exhibit the same I/O behavior across the samples $\mathcal{I}$. More precisely, two nodes n, n' of the obfuscated AST are deemed observationally equivalent if $\vec{o}(n) = \vec{o}(n')$. Additionally, if all the outputs of a node n over $\mathcal{I}$ are identical, the node is said to be *observationally constant*. In practice, GREYSYNTH computes for each node n a fingerprint $\mathcal{F}(n)$ as the hash of its output vector.

4.2 Observational Equivalence Simplification

Algorithm 1 details our Observational Equivalence Simplification (OES) step. OES is a linear-time preprocessing pass that exploits *observational equivalence* (i.e., equivalence modulo sampled inputs $\mathcal{I}$) to prune redundant sub-expressions and reveal hidden constants. It directly addresses the *Syntactic Complexity Explosion* problem: rather than operating over the raw obfuscated AST, whose size grows (often exponentially) with each obfuscation layer, OES compresses the expression so that subsequent processing scales with the number of *semantically distinct* sub-expressions, which grows much more slowly than the total syntactic size.

At a high level, OES constructs a canonical representation of the expression modulo sampling by ensuring that each class of observationally equivalent sub-expressions is represented by a unique minimal sub-expression. It proceeds in two phases:

- (1) *Phase 1: Minimal representative collection.* A bottom-up traversal of the AST (lines 2-10) evaluates each node n on the N samples of $\mathcal{I}$ to compute its fingerprint $\mathcal{F}(n)$, which captures its input/output behavior over $\mathcal{I}$. For each fingerprint, the cache (lines 1, 8-9) stores the *smallest* node seen so far, where size is measured by AST node count $|n|$ (that is, the number of nodes of the subtree of root n). As a result, the cache maintains, for each observationally equivalent sub-expressions, a minimal syntactic representative. If a node is observationally constant (line 5), i.e., all N evaluations yield the same bitvector c , it is immediately cached as the constant c (line 6). Otherwise, the cache is updated whenever a smaller representative is found (lines 8-9). This process collapses all sub-expressions sharing the same observed semantics into a single minimal representative, e.g. $(x \oplus x) \& (x \mid \neg x)$ is merged with 0, and $(x \& \neg y) + (x \& y)$ is merged with x .

- (2) *Phase 2: Top-down replacement.* A second pass rebuilds the expression using the REPLACE function (lines 12-18). For each node n , the algorithm retrieves its fingerprint and checks whether the cache contains a strictly smaller representative. If so, the entire subtree rooted at n is replaced (lines 13-14), effectively pruning large portions of the AST in one step. Otherwise, the algorithm recursively processes its children (lines 15-17) and reconstructs the node. The top-down order is essential: when a node is replaced, its entire subtree is discarded, preventing redundant processing of nodes that would otherwise be traversed and simplified individually.

PROPERTY 1. *OES returns an expression such that two nodes of the resulting AST are observationally equivalent (modulo I) if and only if they are syntactically identical.*

This follows from the invariant maintained by the cache: each equivalence class of nodes (with respect to observational equivalence) is represented by a unique minimal AST node, and all occurrences of that class are rewritten to this canonical representative. As a consequence, semantic equivalence (modulo sampling) coincides with syntactic equality in the transformed AST.

This property is critical for CHS, as it allows purely syntactic comparisons to act as a proxy for semantic equivalence, enabling efficient identification and reuse of equivalent sub-expressions.

PROPERTY 2. *OES runs in a linear time complexity.*

OES evaluates each node of the AST over each sampled input only once thanks to memoization. As we assume that each operator is computed in constant time, the complexity of OES is then in $O(N \cdot I)$, with N the number of nodes and I the number of inputs. In practice I is fixed. Hence, OES runs in linear time of the number of nodes in the AST.

4.3 Contextual Horizon Simplification

While OES identifies and compresses observationally redundant subterms, it cannot perform simplifications that depend on *relationships between different* sub-expressions. For example, the identity $(x \mid y) + (x \& y) = x + y$ cannot be inferred by simplifying each operand independently: it requires recognizing a joint structural pattern across both operands. Contextual Horizon Simplification (CHS) addresses this limitation by combining a structural traversal of the AST with black-box synthesis on bounded-depth abstractions. For instance, given an expression from Fig. 1, CHS will be able to simplify it back to $5x + 6y$. Its design is driven by the three remaining problems identified in Section 3: *Polluted Truncated Trees*, *Semantic Context Blindness*, and *Rigidity in Representation*. We now detail Algorithm 2, which traverses the AST and invokes a PbE synthesizer to simplify local subtrees.

Top-down traversal. The traversal is deliberately performed in a top-down manner: simplifying a node early may subsume its entire subtree, reducing both the number of synthesis queries and the effective search space. This effect is further amplified by OES and caching: since OES merges semantically equivalent subtrees, CHS encounters fewer distinct synthesis problems, and the per-horizon cache ensures that each such problem is solved at most once.

Formally, given an expression e , CHS performs a top-down traversal of its AST and attempts to simplify sub-expressions using a

Algorithm 1: Observational Equality Simplification (OES)

Input: An expression e as an AST, a set of N samples I

Output: A simplified expression e' observationally equivalent to e

```

// Phase 1: Bottom-up scan to collect
// minimal representatives
1 Cache  $\leftarrow \emptyset$ ;
2 foreach node  $n$  in a bottom-up traversal of  $e$  do
3    $\mathcal{F}(n) \leftarrow \text{Fingerprint}(n)$ ;
4   if IsConstant( $n$ ) then
5      $c \leftarrow \text{Constant}(n)$ ;
6     Cache [ $\mathcal{F}(n)$ ]  $\leftarrow c$ 
7   else
8     if  $\mathcal{F}(n) \notin \text{Cache}$  or  $|n| < |\text{Cache}[\mathcal{F}(n)]|$  then
9       Cache [ $\mathcal{F}(n)$ ]  $\leftarrow n$ ;
10  $e' \leftarrow \text{Replace}(e)$ ;
11 return  $e'$ ;

// Phase 2: Top-down replacement
// with minimal occurrences
12 Function Replace( $n$ )
13   // Check if a smaller equivalent
14   // sub-expression exists in cache
15   if  $|\text{Cache}[\mathcal{F}(n)]| < |n|$  then
16     return Cache [ $\mathcal{F}(n)$ ];
17   // Explicitly traverse and replace each child
18   foreach  $c \in \text{GetChildren}(n)$  do
19      $c' \leftarrow \text{Replace}(c)$ ;
20     Replace  $c$  by  $c'$  in  $n$ ;
21 return  $n$ ;

```

depth horizon h , which progressively increases (lines 1–5 of SIMPLIFY in Algorithm 2).

Each node is processed through the PROCESS procedure. This procedure first extracts a simplification target, noted *tree*, using the TRUNCATE procedure (line 18). The *tree* is a sub-expression of e such that (i) it is rooted at node n , and (ii) all nodes beyond depth h are replaced by fresh stub variables $\alpha_1, \dots, \alpha_k$, yielding a truncated AST of depth h , while nodes within this horizon are preserved and exposed to the synthesizer.

The mapping *map* associates each stub variable with the original subtree it replaces. A key design point is that stub assignment is governed by the syntactic-semantic correspondence established by OES: subtrees with the same fingerprint share the *same* stub. This directly addresses the *Polluted Truncated Trees* problem.

The I/O specification of the truncated *tree* is then computed (lines 19-29), and submitted to a PbE synthesizer (line 30). If a simpler expression *tree'* is found, it replaces *tree* in the full AST after de-anonymizing the stub variables using the *map* (line 32). Once all eligible nodes have been processed at horizon h , the algorithm proceeds to horizon $h + 1$, until h exceeds the AST depth.

Per-horizon caching. To further reduce synthesis cost, CHS maintains a cache for each horizon h , keyed by fingerprints. Before

processing a node, the algorithm checks whether a node with the same fingerprint has already been simplified at this horizon (line 10). If so, the cached result is reused directly.

The cache maps each fingerprint to the result of its simplification at horizon h , i.e., either a rewritten subtree if synthesis succeeds, or the original node otherwise. This ensures that each semantically distinct sub-expression is synthesized at most once per horizon.

This mechanism is particularly effective when obfuscation replicates identical semantics with syntactic noise. For instance, consider the expression: $((x \mid y) + (x \& y) + (z \oplus z)) + ((x \mid y) + (x \& y) + (t \oplus t))$. After OES simplifies null expressions such as $z \oplus z$ and $t \oplus t$ to 0, the expression reduces to: $((x \mid y) + (x \& y)) + ((x \mid y) + (x \& y))$. At horizon $h = 2$, both subtrees $(x \mid y) + (x \& y)$ yield the same truncated form $(A \mid B) + (A \& B)$ and are assigned the same fingerprint. Without caching, each occurrence would trigger an identical synthesis query. With per-horizon caching, the result is computed once and reused across all occurrences.

More generally, in expressions where a pattern is repeated k times, caching reduces the number of synthesis calls from $O(k)$ to $O(1)$ for that pattern. This significantly mitigates the impact of syntactic duplication and contributes to the scalability of CHS.

Black-box synthesis. Once the white-box traversal delivers the truncated AST *tree*, PROCESS invokes the PbE synthesizer (line 30). A key question at this point is what concrete values must be assigned to the stub variables. One could assign random values and run the synthesis. However, it would lead to the *Semantic Context Blindness* problem (cf. Fig. 1). We advocate for *contextual sampling*: rather than assigning fresh random values to each stub α_i , GREYSYNTH evaluates every stub on the real program samples $\mathcal{I}$, i.e., $\vec{o}(\alpha_i)$. Propagating the program’s actual I/O context makes functional dependencies, which extend beyond the local horizon, immediately visible: if two stubs satisfy an invariant such as $\alpha_1 + \alpha_2 = 0$ across all samples of original variables, the synthesizer can exploit it for context-valid simplifications that would be entirely invisible with independent random values. Furthermore, contextual sampling automatically enforces inter-stub dependencies without additional bookkeeping—a natural consequence of the same fingerprint-based stub unification performed by the white-box phase.

Moreover, constant values present in the truncated tree are *dynamically injected as grammar terminals* prior to synthesis. It addresses *Rigidity in Representation*: since Phase 1 of OES has already resolved obfuscated constant expressions to their concrete values, the grammar adapts to each synthesis target without any a priori knowledge of the obfuscation scheme, without the limitations of SMT-based constant recovery.

Crucially, this synthesis step is agnostic of the actual synthesizer—any oracle that operates on concrete input-output examples suffices, making GREYSYNTH a general framework.

5 Evaluation

Our evaluation answers the following four Research Questions:

RQ1 *How does GREYSYNTH perform against state-of-the-art deobfuscators on real-world, non-synthetic expressions?* We evaluate GREYSYNTH against black-box tools SYNTIA, XYNTIA, and XSMIR, the grey-box tool QSYNTH, and white-box tools PROMBA

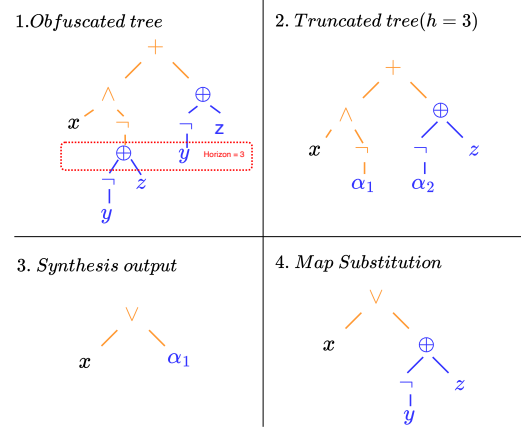


Figure 2: Example of CHS simplification.

We illustrate how CHS synthesizes the rule $(A \wedge (\neg B) + B) \rightarrow A \vee B$: 1. an obfuscated tree where $A = x$ and $B = (\neg y) \oplus z$. We highlight in orange the rewrite-rule pattern and in blue the repeating semantic of B . 2. in order to catch the rule, we need horizon at 3, so we stub $\alpha_1 := \neg y \oplus z$ and $\alpha_2 := y$. 3. since we passed concrete values of α_1 and α_2 such that $\alpha_1 = \neg \alpha_2 \oplus z$, the synthesis engine was able to retrieve $x \vee \alpha_1$. 4. we substitute the subtrees back into α_1 , obtaining $x \vee (\neg y \oplus z)$

and GAMBA, on a new dataset of obfuscated expressions extracted from real-world C libraries;

RQ2 *How does GREYSYNTH perform on MBA-obfuscated expressions compared to the state-of-the-art deobfuscators, and especially MBA-specialized tools?* We evaluate GREYSYNTH against SYNTIA, XYNTIA, XSMIR, QSYNTH, PROMBA, and GAMBA on MBA-focused benchmarks, including both standard synthetic datasets from prior works and a new dataset of MBA expressions extracted from real-world C libraries;

RQ3 *How does GREYSYNTH handle syntactic complexity explosion compared to other white-box and grey-box tools?* We compare GREYSYNTH against PROMBA, GAMBA and QSYNTH on a dataset of increasing obfuscation complexity.

RQ4 *How do internal parameters impact GREYSYNTH performance?* We study the importance of OES and CHS in the algorithm, the injection of constant values in the synthesis grammar and the impact of the black-box synthesizer in GREYSYNTH.

5.1 Methodology

GREYSYNTH implementation and setup.

GREYSYNTH is implemented in OCaml. It takes as input an expression to simplify, either given as a raw string, or extracted from a C file or from a binary basic block (using the BINSEC tool [21]). Unless explicitly mentioned, GREYSYNTH uses OES, CHS and constant values injection. GREYSYNTH internal PbE synthesizers defaults to XYNTIA [34], with a 2s timeout, 100 I/O samples, the XYNTIA FULL grammar (the usual MBA operators plus shift, division, and modulo operators) and other default parameters. The evaluation of GREYSYNTH with other PbE synthesizers (Section 5.5) uses the same setup (100 I/O and FULL grammar) and the tools default configurations, with varying timeouts from 2s to 10s. Our implementation

Algorithm 2: Contextual Horizon Simplification (CHS)**Input:** Expression e , samples $\mathcal{I}$, synthesis timeout T **Output:** Expression e simplified

```

1  $h \leftarrow 2$ ;
2 while  $h \leq \text{depth}(e)$  do
3    $\text{Cache} \leftarrow \emptyset$ ;           // fresh per-horizon cache
4    $e \leftarrow \text{Simplify}(e, h, \text{Cache})$ ;
5    $h \leftarrow h + 1$ ;
6 return  $e$ ;

7 Function  $\text{Simplify}(n, h, \text{Cache})$ 
8   if  $\text{depth}(n) < h$  then
9     // Subtree too shallow: no truncation
10    // at depth  $h$  is possible
11    return  $n$ ;
12   if  $\mathcal{F}(n) \in \text{Cache}$  then return  $\text{Cache}[\mathcal{F}(n)]$ ;
13    $\text{result} \leftarrow \text{Process}(n, h)$ ;
14   foreach  $c \in \text{GetChildren}(\text{result})$  do
15      $c' \leftarrow \text{Simplify}(c, h, \text{Cache})$ ;
16     Replace child  $c$  by  $c'$  in  $\text{result}$ ;
17    $\text{Cache}[\mathcal{F}(n)] \leftarrow \text{result}$ ; // memoize simplified
18   // node (the fingerprint is immutable)
19   return  $\text{result}$ ;

17 Function  $\text{Process}(n, h)$ 
18    $(\text{tree}, \text{map}) \leftarrow \text{Truncate}(n, h, \mathcal{I})$ ;
19   //  $\text{map}$  maps each stub  $\alpha_i$  to the original
20   // subtree it abstracts (subtrees with the
21   // same OES fingerprint share the same stub)
22    $S \leftarrow \emptyset$ ;
23   foreach  $i \in 0..|\mathcal{I}|$  do
24      $\text{inputs} = \emptyset$ ;
25     foreach  $v \in \text{Vars}(\text{tree})$  do
26       if  $v$  is an original variable then
27          $\text{value} \leftarrow i^{\text{th}}$  value of  $v$  in  $\mathcal{I}$ ;
28       else
29         // contextual samples for stub
30          $\text{value} \leftarrow i^{\text{th}}$  value of  $\tilde{o}(\text{map}[v])$ ;
31        $\text{inputs} \leftarrow \text{inputs} \cup \{v : \text{value}\}$ ;
32   // evaluate tree on contextual inputs
33    $\text{out} \leftarrow i^{\text{th}}$  value of  $\tilde{o}(\text{tree})$ ;
34    $S \leftarrow S \cup \{\text{inputs}, \text{out}\}$ ;
35    $\text{tree}' \leftarrow \text{PbE-Synth}(S, T)$ ; //  $T = \text{TimeOut}$ 
36   if  $\text{tree}' \neq \perp$  and  $|\text{tree}'| < |\text{tree}|$  then
37     Replace all stub variables  $\alpha_i$  with  $\text{map}[\alpha_i]$  in  $\text{tree}'$ ;
38     return  $\text{tree}'$ ;
39   return  $n$ ;

```

makes calls to a black-box synthesizer either via an OCaml API or a command line call.

Competitors. GREY_{SYNTH} against six main competitors, all with a 1h timeout (TO) and their default configurations: the three leading black-box deobfuscators SYNTIA [11], XYNTIA [34] and XSMIR [4]; the two latest and leading white-box MBA deobfuscators PROMBA [29] and GAMBA [40]; and the first (and only so far) grey-box deobfuscator QSYNTH [20, 22]. To ensure a fair comparison, we configure each competitor according to its best-known settings or official documentation. SYNTIA is evaluated using 100 I/O samples, as it offers no further tuning parameters. For XYNTIA and XSMIR, we use the FULL grammar with 100 samples. For QSYNTH, we follow the recommended setup from its official repository [41]: we generate offline tables for three variables using 16 I/O samples and a random level of 5 (to avoid corner-case bias) across a set of five million expressions. Notably, we extend the default configuration by generating tables for 16, 32, 64, and 128-bit vectors to accommodate the varied bit-widths in our dataset. Finally, for the white-box tools, we evaluate PROMBA using its default configuration [27], which specifies a maximum horizon height of 30, the Symba enumerative synthesizer, and the Symbad linear MBA solver. GAMBA is executed in its default state (no option choice).

Metrics. The *success rate* is the ratio of cases where the tool returns a successful answer, i.e., a solution that passes the SMT checks.¹ The *false positive rate* measures the fraction of cases a tool returns a solution non-equivalent to the ground-truth (the SMT check returns a counter-example). The *no-result rate* is fraction of cases where no solution is returned (typically, crashes or time-out for black-box synthesizers). *Quality* measures the capacity to return an expression close to the ground truth (original unobfuscated expression). It is computed as the ratio between the size of the result expression and the size of the ground truth expression (the lower the better, 1 denotes perfect simplification). *Compression* measures simplification power and is computed as the ratio between the size of the obfuscated expression and the size of the result expression (the higher the better). The *time* metric records the time for outputting a result. The number of time a method fails to terminate within a 1h time budget is recorded in the *timeout rate*. When the tool timeouts but returns an intermediate result, this one is considered as the output. For fairness, when a tool returns no solution, we consider that it returns the input obfuscated expression.

Benchmarks. We consider two main datasets:

- **PROMBA dataset.** We use the dataset used by Lee and Lee [29] to compare GREY_{SYNTH} with state-of-the-art MBA tools. We split this dataset in two: *Synthetic MBA* and *MBA loki*. The former includes 2,511 synthetic expressions used to evaluate PROMBA, including 1,003 polynomial MBAs (referred as *Poly* in [29]) and 1,008 non-polynomial MBAs (referred as *Non-Poly* in [29]), and 500 expressions from the QSYNTH paper. The latter includes 1,500 MBAs extracted from the LOKI obfuscator, among which 1,000 have a restricted MBA grammar (referred as *Loki* in [29]) and 500 have a more general grammar (referred as *Full-Loki* in [29]). However, we argue that, being synthetic, such datasets are

¹As we consider highly obfuscated code, SMT-based equivalence checks between the simplified version and the ground truth or the obfuscated version may time out. We use the SMT check as a thorough, but approximate, check of correctness of the result, i.e., we consider the check valid when the SMT solver cannot prove it invalid.

inherently limited, especially for evaluating grey-box deobfuscation;

- **In-the-Wild semantics datasets with ground truth: C-MBA, C-BeyondMBA and C-Full.** Existing datasets have clear shortcomings for grey-box deobfuscation evaluation. Indeed, they are either fully synthetic (no real semantics to stress-test synthesis) or no ground-truth is available. Hence, In order to mitigate that, we built a dataset of real expressions extracted from twelve popular C libraries: OpenSSL, BearSSL, Monocypher, fftw, gmp, sleef, libcanard, libevent, libuv, Open62541, Quake and SQLite. All expressions are extracted after expanding all constant value macros, and conserving their bitsize and signedness. We filtered out expressions with a depth lower than 2, and ended up with about 7,000 expressions, of which we randomly extracted 500 MBA expressions (C-MBA dataset) and 500 non MBA expressions (C-BeyondMBA dataset). The C-Full dataset is the concatenation of C-MBA and C-BeyondMBA. Each dataset has been then obfuscated with (i) *Tigress* [15] using its rewrite-rules based MBA engine; (ii) *MBA Obfuscator* [36] with the option *non_polynomial+zero*, which transforms an expression in a non-polynomial one plus a complicated expression that always evaluates to zero, and (iii) *OMVLL*, a tool that obfuscates at the LLVM IR level, with a rewrite rules-based MBA engine. Unfortunately, MBA Obfuscator could not be considered for C-BeyondMBA as it failed over it.
- **Evaluation pipeline** Every MBA expression is placed in a single function and compiled. We retrieve the target expression from the `eax` (or `rax`) register with careful consideration given to input and output bit-widths. How these expressions are compiled depends on the obfuscator. For obfuscators generating high-level code (*Tigress* returns C code, *MBA-Obfuscator* returns string expressions then converted to C), we compile the result with GCC 12.2.0 (with `-O0` flag). We explicitly enforce `-O0` to ensure that we evaluate the deobfuscator and not GCC. For *OMVLL* that returns binary code, we use its default setting, i.e., the Android NDK `r26d` Clang toolchain with no optimization flag.

Setup. We run all experiments on three servers with each the same configuration, an *Intel Xeon 4214* with 48 cores running at 3.02GHz and 400 GB of memory. All simplification tasks are parallelized on the 48 cores.

5.2 RQ1: Comparison on general ground truths

This section studies the efficiency and generality of GREYSYNTH, and compares it against state-of-the-art techniques on the general obfuscated expressions from our C-BeyondMBA dataset (pure MBA expressions are the focus of RQ2). Results are provided in Table 2.

GREYSYNTH vs black-box methods. GREYSYNTH handles $\approx 95\%$ of the datasets while black-box approaches succeed in at most 44% of the cases. This is expected as black-box methods are inherently limited by the semantic complexity. Hence, GREYSYNTH better simplifies the expressions, reaching an average a quality factor of $1.2 - 1.7\times$ against at best $10 - 20\times$ for black-box approaches. Moreover, GREYSYNTH is faster: average simplification time of 1.3-4.0s vs. 60s for XSMIR the most competitive black-box method.

GREYSYNTH vs white- and grey-box methods. GREYSYNTH achieves a consistent $1.2-1.7\times$ quality factor across the two datasets, which is reached in a few seconds on average. This contrasts with PROMBA, which achieves only a quality of $4.0-18\times$ depending on the obfuscator (*OMVLL* or *Tigress*), within minutes. Moreover, QSYNTH and GAMBA often fail to return an expression on this dataset. Indeed, QSYNTH returns non-equivalent results for 87.2% of the dataset obfuscated with *Tigress*. Still, even when it succeeds (*OMVLL* case), QSYNTH cannot simplify expressions as much a GREYSYNTH does (quality: $1.2\times$ vs $7.9\times$). Finally, GAMBA returns no results on 94% of the dataset obfuscated with *OMVLL* and 48% with *Tigress*. The poor results of GAMBA are due to (i) the ground-truths, which contain operations like division, modulo or shift that it cannot handle, and (ii) the obfuscation and the transformation to a binary for *OMVLL* that add operators that GAMBA cannot process like bit shifting.

Conclusion. GREYSYNTH highly outperforms black-, white- and grey-box methods on general obfuscated expressions. It successfully handles 95% of the dataset and achieves fast, near-perfect simplification.

5.3 RQ2: Comparison on MBA ground truths

Table 2 (columns PROMBA dataset and C-MBA dataset) compares GREYSYNTH against prior works over pure MBA expressions.

GREYSYNTH vs black-box methods. As expected, GREYSYNTH highly outperforms black-box deobfuscation. Indeed, black-box methods suffer from their all-or-nothing behavior. Except for the MBA *Loki* case, where SYNTIA, XSYNTIA and XSMIR successfully synthesized $\approx 70\%$ of the dataset, no black-box deobfuscator can handle more than $\approx 39\%$ of the other MBA datasets. Hence, black-box methods often timeout and are on average 1.6 to 55 times slower than GREYSYNTH, and cannot simplify expressions as much.

GREYSYNTH vs white- and grey-box methods. The comparison against white- and grey-box methods is more subtle. We observe that GREYSYNTH performs better as datasets become less synthetic. The worst case arises with the *Synthetic MBA* dataset, where most expressions are synthetic. Especially, GAMBA simplifies expressions better than GREYSYNTH (quality of 1.0 vs 1.4) and is $5.7\times$ faster. QSYNTH also achieves good results, simplifying a bit less than GREYSYNTH but being $500\times$ faster. Such an impressive speed is expected. Indeed, QSYNTH uses a pre-computed (hence fast) cache of expressions, which exactly fit the synthetic scenario.

Still, as soon as expressions become less synthetic, GREYSYNTH improves. Considering the quality metric for the *MBA Loki* case, we observe that GREYSYNTH (i) outperforms GAMBA and QSYNTH; and (ii) is on par with PROMBA. It is also $388\times$ faster than PROMBA, $42\times$ faster than GAMBA, and as fast as QSYNTH. Over the real-world C expressions from the C-MBA dataset, GREYSYNTH improves further. It always outperforms QSYNTH and PROMBA, reaching a quality of $\approx 2\times$ over the *OMVLL* and *Tigress* obfuscators against $\approx 5 - 6\times$ for QSYNTH and up to $34\times$ for PROMBA. For *MBA-Obfuscator*, simplification is harder (quality of $4.6\times$ for GREYSYNTH) but always better than QSYNTH (quality: $6.2\times$) and PROMBA (quality: $133\times$). GREYSYNTH also outperforms GAMBA for the *OMVLL* and *Tigress* obfuscators (quality of $\approx 2\times$ for GREYSYNTH against $\approx 5\times$ for GAMBA) and is 3 to 14 times faster. Still, GAMBA remains slightly better over *MBA-Obfuscator*.

Table 2: Evaluation of GREYSYNTH against white-, black- and grey-box deobfuscators (TO=1h)

		ProMBA dataset		C-MBA dataset			C-BeyondMBA dataset	
		Synthetic MBA	MBA Loki	MBA-Obfuscator	OMVLL	Tigress	OMVLL	Tigress
SYNTIA	Success	22.5%	71.2%	14.2%	14.4%	14.2%	0.2%	0.2%
	False positive	3.4%	21.3%	6.0%	6.1%	6.0%	0.0%	0.0%
	No result	74.2%	7.6%	79.8%	79.5%	79.8%	99.8%	99.8%
	Max compression	467×	5346×	9364×	505×	23269×	1.0×	95×
	Compression	1.9×	25×	1.9×	1.6×	2.6×	1.0×	1.0×
	Quality	5.9×	5.5×	99×	53×	411×	46×	198×
	Time	1360.8s	8.4s	1743.2s	1725.3s	1743.2s	3561.9s	3561.9s
	Timeouts	74.2%	7.6%	79.8%	79.5%	79.8%	99.8%	99.8%
XYNTIA	Success	30.7%	75.7%	23.8%	24.1%	23.8%	28.6%	28.6%
	False positive	20.3%	17.5%	19.8%	19.9%	19.8%	20.4%	20.4%
	No result	49.0%	6.8%	56.4%	56.0%	56.4%	51.0%	51.0%
	Max compression	833×	5346×	23982×	1889741×	26594×	2262×	5160×
	Compression	2.3×	41×	3.1×	2.6×	4.7×	2.4×	4.2×
	Quality	4.9×	3.4×	60×	34×	227×	19×	48×
	Time	214.9s	0.4s	175.3s	167.9s	175.3s	95.2s	95.2s
	Timeouts	48.7%	5.7%	55.4%	55.0%	55.4%	51.0%	51.0%
XSMIR	Success	35.0%	72.5%	39.0%	38.9%	39.0%	44.0%	44.0%
	False positive	9.8%	20.0%	11.6%	11.8%	11.6%	12.8%	12.8%
	No result	55.2%	7.5%	49.4%	49.3%	49.4%	43.2%	43.2%
	Max compression	833×	6736×	125941×	21837012×	41586×	3572×	4749×
	Compression	2.9×	34×	7.9×	4.9×	14×	4.7×	9.8×
	Quality	4.0×	4.0×	24×	18×	79×	10.0×	20×
	Time	600.6s	10.7s	163.7s	162.7s	163.7s	63.2s	63.2s
	Timeouts	55.1%	6.5%	48.4%	48.3%	48.4%	43.2%	43.2%
QSYNTH	Success	99.7%	90.0%	98.0%	88.0%	96.0%	76.0%	12.6%
	False positive	0.3%	10.0%	1.0%	8.7%	3.0%	20.8%	87.2%
	No result	0.0%	0.0%	1.0%	3.2%	1.0%	3.2%	0.2%
	Max compression	467×	5346×	42772×	18279×	27095×	9841×	834×
	Compression	7.0×	73×	15×	14×	217×	5.9×	1.5×
	Quality	1.6×	1.9×	13×	6.2×	4.9×	7.9×	135×
	Time	0.04s	0.1s	1.8s	0.1s	1.8s	0.1s	1.5s
	Timeouts	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%
ProMBA	Success	61.3%	100.0%	98.0%	91.9%	98.4%	90.0%	88.6%
	False positive	38.7%	0.0%	1.0%	4.1%	0.6%	8.8%	11.0%
	No result	0.0%	0.0%	1.0%	4.1%	1.0%	1.2%	0.4%
	Max compression	833×	5346×	127×	4121×	23103×	3796×	3584×
	Compression	1.8×	132×	1.4×	14×	32×	11×	11×
	Quality	6.4×	1.0×	133×	6.3×	34×	4.0×	18×
	Time	113.9s	38.8s	3167.2s	222.3s	762.2s	53.2s	257.0s
	Timeouts	0.0%	0.0%	94.2%	6.1%	12.6%	4.2%	5.8%
GAMBA	Success	99.3%	83.5%	89.8%	74.6%	93.8%	5.4%	37.0%
	False positive	0.7%	8.3%	4.8%	0.8%	0.2%	0.0%	14.2%
	No result	0.0%	8.2%	5.4%	24.5%	6.0%	94.6%	48.8%
	Max compression	833×	5346×	10981×	76773×	26594×	3796×	1395×
	Compression	12×	64×	73×	19×	199×	1.2×	2.7×
	Quality	1.0×	2.2×	2.6×	4.6×	5.3×	37×	73×
	Time	3.5s	4.2s	9.3s	8.7s	66.5s	2.5s	37.8s
	Timeouts	0.0%	0.0%	1.2%	3.7%	5.0%	0.0%	0.2%
GREYSYNTH	Success	98.3%	94.2%	95.4%	90.9%	91.8%	96.2%	95.2%
	False positive	1.5%	0.4%	2.0%	7.7%	6.0%	2.8%	4.2%
	No result	0.2%	5.4%	2.6%	1.4%	2.2%	1.0%	0.6%
	Max compression	833×	6736×	10384×	4913327×	27504×	708907×	4790×
	Compression	8.2×	108×	41×	40×	583×	39×	120×
	Quality	1.4×	1.3×	4.6×	2.2×	1.8×	1.2×	1.7×
	Time	20.2s	0.1s	97.9s	2.9s	4.5s	1.3s	4.0s
	Timeouts	0.0%	0.0%	6.2%	1.0%	0.0%	0.0%	0.0%

Average values of the quality, compression, and time metrics are computed through the geometric mean (the arithmetic mean is provided in parentheses).

Conclusion. As expected, GREYSYNTH performs significantly better on MBA expressions than black-box methods and prior grey-box proposal. More surprisingly, GREYSYNTH performs well against dedicated white-box MBA simplifiers on their domain. When expressions come

from the real-world, GREYSYNTH outperforms GAMBA and PROMBA, being up to 32× faster and achieving better simplification.

5.4 RQ3: Scaling experiment

We now evaluate how GREYSYNTH scales on highly obfuscated expressions compared to existing white- and grey-box methods—black-box tools are discarded since they are immune to syntactic complexity. The evaluation is performed on the *C-MBA* dataset to provide favorable conditions for the MBA deobfuscators GAMBAA and PROMBA, and we consider an increasing number of MBA obfuscation passes using *Tigress*.

Results are provided in Table 3. We observe that the success of GREYSYNTH stays around $\approx 90\%$ across all obfuscation levels while QSYNTH and GAMBAA are highly impacted: success decreases from 98.6% to 14.4% for QSYNTH and from 98.2% to 7.4% for GAMBAA. Only PROMBA achieves a success comparable to GREYSYNTH across all obfuscation levels. Still, GREYSYNTH is the only framework that maintains a steady 1.1-2.6 $\times$ quality factor, showing that it manages to massively retrieve the ground truth. On the other hand, GAMBAA, PROMBA, and QSYNTH quality factors significantly deteriorate with the number of obfuscation passes, respectively from 1.2 $\times$ to 8,118 $\times$, from 2.2 $\times$ to 494 $\times$ and from 1.8 $\times$ to 4,264 $\times$. Regarding the computation time, GREYSYNTH stays fast over all obfuscation levels. Even over the most obfuscated level, GREYSYNTH takes only 18s on average to simplify an expression, against 161s for QSYNTH, 493s for GAMBAA and 1,796s for PROMBA.

Conclusion. *GREYSYNTH demonstrate very good scalability against highly obfuscated code, with almost stable performance in success and quality and linear increase of computation time, while the state-of-the-art white-box and grey-box tools significantly suffer from the syntactic explosion on all these dimensions.*

5.5 RQ4: Ablation studies

In this section we analyze the impact of GREYSYNTH internal parameters. We focus on (i) the core algorithmical component, namely OES and CHS, (ii) the constant value injection mechanism and (iii) the use of different synthesizers within GREYSYNTH.

Impact of OES, CHS, and caching. We evaluate OES and CHS against a straightforward baseline called *Global*, that performs synthesis on each node of the expression without truncating horizon (in the spirit of QSYNTH, but with synthesis instead of cache lookup). We also study the impact of the caching mechanism in CHS, which has been replicated in *Global*. We evaluate *Global*, OES followed by *Global*, OES alone, CHS alone and CHS with a preable pass of OES (i.e., GREYSYNTH). Comparisons are performed over the *C-Full* dataset obfuscated with *Tigress*.

As expected, caching clearly reduces the computation time, with a spectacular 16 $\times$ ratio for *Global* alone, and halves the computation time for CHS and GREYSYNTH, yet it also significantly increases the false positive rate when OES is not activated. Regarding the components themselves, *Global* is slow (30 seconds in average), output expressions with bad quality (15 – 20 $\times$) and suffers from high false positive rate when cache is activated. Quality goes to 5.4 $\times$ with the help of EOS but is still too high. OES alone achieves a compression factor of 21 $\times$, which is good considering that it only replaces nodes by equivalent ones that are *already* present in the AST, and is extremely fast. However, its 22 $\times$ quality factor disqualifies it as a standalone feature. CHS brings a steady and strong improvement,

as all the configurations built on it yield significantly better results on all dimensions than the 6 other configurations – for example compression is at least 199 $\times$ with CHS (with cache) and at best 85 $\times$ without (OES + Global + cache).

Overall, GREYSYNTH (with cache or without) and CHS alone (no cache) seems to be the three best performers, with slight variations on success – explaining the variations in quality and compression, compensated by (more visible) variations in computation time.

Constant value Injection. Table 5 investigates the impact of injecting constant values into the GREYSYNTH synthesis grammar. We compare the performance of GREYSYNTH when using XYNTIA or XSMIR [4] (extension of XYNTIA with inference rules to better handle constant values) as internal synthesis engines. While constant injection improves XYNTIA performance – reducing the quality factor from 2.3 $\times$ to 1.7 $\times$ and cutting mean computation time from 11s to 4.3s – the same optimization is counterproductive for XSMIR. In the case of XSMIR, constant injection leads to a degradation in both quality (from 3.3 $\times$ to 3.5 $\times$) and latency (from 18s to 21s). We hypothesize that these injected constant values interfere with XSMIR inference rules, leading to inefficiencies in state-space exploration.

Different synthesis engines. We now evaluate the impact of using different blackbox synthesizers (SYNTIA, CVC5, XYNTIA, or XSMIR) in GREYSYNTH, for a varying timeout per synthesis of 2, 5, and 10 seconds. The results, presented in Table 6, confirm that GREYSYNTH is indeed synthesizer-agnostic, successfully functioning with all the evaluated engines. However, significant performance disparities emerge: XYNTIA consistently achieves the most favorable quality factors. Notably, XYNTIA achieves a quality factor of 1.7 $\times$ at a 2s timeout, outperforming SYNTIA (8.8 $\times$), CVC5 (2.9 $\times$) and XSMIR (3.5 $\times$), while maintaining a computation time of only 4.3s, an order of magnitude faster than the other engines. This suggests that while GREYSYNTH provides a flexible scaffolding, its performance is at best when coupled with the search strategies from XYNTIA.

Conclusion. *CHS and constant value injection shows significant and steady impact on GREYSYNTH performance, while OES and caching allows to boost computation time for small loss in success rate. Finally, GREYSYNTH is able to work well with several black-box synthesizers, yet it performs as its best when paired with XYNTIA.*

6 Discussion

Adversarial Robustness. As established, white-box and black-box deobfuscation both suffer from theoretical and practical limits, while GREYSYNTH’s design allows it to bypass several protection mechanisms that typically challenge previous tools. Complex semantic logic, which often causes black-box synthesizers to stall, is easily partitioned and decomposed by the white-box structural traversal. Conversely, constructs that induce severe syntactic explosion for white-box analyzers—such as opaque predicates or mixed boolean-arithmetic (MBA) constructs—are naturally handled through black-box observational sampling.

As an illustration, consider the code snippet below containing an opaque predicate:

When evaluated over this snippet, GREYSYNTH directly recovers the simplified expression $x \cdot y + x$, demonstrating its ability to bypass the opaque predicate protection.

Table 3: Scalability analysis on C-FULL obfuscated with increasingly complex Tigress [15] MBA obfuscations (TO= 1h)

		1	2	3	4	5	6	7	8
QSYNTH	Success	98.6%	98.4%	97.2%	96.4%	96.0%	27.6%	17.6%	14.4%
	False positive	0.4%	0.6%	1.8%	2.6%	3.0%	71.4%	81.4%	84.6%
	No result	1.0%	1.0%	1.0%	1.0%	1.0%	1.0%	1.0%	1.0%
	Max compression	104x	414x	1128x	5080x	27095x	14964x	64847x	309271x
	Compression	3.0x (4.7x)	7.7x (16x)	21x (51x)	66x (224x)	217x (944x)	4.6x (212x)	3.0x (463x)	2.9x (1873x)
	Quality	1.8x (2.1x)	2.4x (3.2x)	3.2x (8.7x)	4.1x (34x)	4.9x (91x)	250x (2098x)	1074x (7386x)	4264x (40211x)
	Time	0.01s (0.02s)	0.03s (0.1s)	0.1s (0.3s)	0.5s (1.9s)	1.8s (9.6s)	9.0s (38.9s)	33.4s (170.6s)	160.8s (1167.9s)
	Timeouts	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	5.8%
ProMBA	Success	98.4%	98.6%	98.6%	98.6%	98.4%	98.6%	98.4%	98.4%
	False positive	0.6%	0.4%	0.4%	0.4%	0.6%	0.4%	0.6%	0.6%
	No result	1.0%	1.0%	1.0%	1.0%	1.0%	1.0%	1.0%	1.0%
	Max compression	78x	249x	1237x	5570x	23103x	20498x	37950x	110973x
	Compression	2.5x (3.6x)	4.8x (10x)	9.9x (37x)	16x (118x)	32x (430x)	19x (412x)	27x (1179x)	25x (3600x)
	Quality	2.2x (3.8x)	3.9x (12x)	6.7x (35x)	16x (156x)	34x (585x)	60x (2835x)	122x (7722x)	494x (113788x)
	Time	88.2s (333.7s)	184.4s (572.5s)	338.9s (921.3s)	595.7s (1328.3s)	762.2s (1685.0s)	1114.0s (1980.5s)	1379.0s (2248.2s)	1796.3s (2575.1s)
	Timeouts	2.2%	2.0%	2.4%	4.4%	12.6%	23.8%	33.8%	47.2%
GAMBA	Success	98.2%	97.6%	96.6%	96.2%	93.8%	26.2%	19.0%	7.4%
	False positive	0.0%	0.2%	0.2%	0.2%	0.2%	54.4%	47.4%	31.8%
	No result	1.8%	2.2%	3.2%	3.6%	6.0%	19.4%	33.6%	60.8%
	Max compression	231x	925x	1388x	6648x	26594x	15986x	57551x	184078x
	Compression	4.4x (7.6x)	13x (29x)	34x (88x)	91x (414x)	199x (1294x)	4.3x (351x)	2.9x (484x)	1.5x (779x)
	Quality	1.2x (5.2x)	1.4x (3.1x)	1.9x (5.8x)	2.9x (22x)	5.3x (190x)	266x (2104x)	1108x (7334x)	8118x (43330x)
	Time	4.0s (10.8s)	5.2s (29.5s)	8.2s (27.9s)	21.4s (74.4s)	66.5s (210.2s)	261.2s (672.2s)	485.6s (1007.4s)	493.0s (1106.1s)
	Timeouts	1.2%	1.6%	2.6%	2.8%	5.0%	16.4%	30.6%	57.8%
GREYSYNTH	Success	95.6%	94.2%	92.8%	94.0%	91.8%	90.6%	91.6%	90.8%
	False positive	2.2%	3.8%	5.0%	4.0%	6.0%	8.2%	7.2%	8.0%
	No result	2.2%	2.0%	2.2%	2.0%	2.2%	1.2%	1.2%	1.2%
	Max compression	145x	342x	1237x	5505x	27504x	40830x	87793x	663931x
	Compression	5.1x (7.5x)	15x (28x)	47x (103x)	183x (494x)	583x (2338x)	555x (2220x)	1490x (7043x)	4798x (36595x)
	Quality	1.1x (1.3x)	1.2x (2.7x)	1.4x (9.5x)	1.5x (61x)	1.8x (180x)	2.1x (331x)	2.2x (746x)	2.6x (4503x)
	Time	1.8s (7.8s)	3.0s (9.5s)	4.3s (12.2s)	4.7s (12.2s)	4.5s (12.6s)	8.0s (23.2s)	10.8s (35.8s)	17.6s (51.8s)
	Timeouts	0.0%	0.0%	0.0%	0.0%	0.0%	0.0%	0.2%	0.2%

Average values of the quality, compression, and time metrics are computed through the geometric mean (the arithmetic mean is provided in parentheses).

Table 4: Ablation study on the C-Full dataset with a 1h TO

		Global	OES + Global	OES	CHS	GREYSYNTH
No cache	Success	95.1%	91.3%	93.3%	96.4%	94.0%
	False positive	1.6%	5.8%	6.7%	2.3%	4.7%
	No result	3.3%	2.9%	0.0%	1.3%	1.3%
	Max compression	15,682x	25,014x	17,566x	29,704x	41,586x
	Compression	23x (558x)	76x (880x)	21x (138x)	299x (1,316x)	274x (1,376x)
	Quality	20x (403x)	6x (269x)	22x (322x)	1.5x (80x)	1.7x (120x)
	Time	477s (1180s)	40s (437s)	0.7s (6.3s)	13s (60s)	7.3s (32s)
	Timeouts	13.1%	3.2%	0.0%	0.0%	0.0%
Cache	Success	87.1%	91.8%	93.3%	90.5%	93.5%
	False positive	11.4%	6.9%	6.7%	8.1%	5.1%
	No result	1.5%	1.3%	0.0%	1.4%	1.4%
	Max compression	41,586x	29,704x	17,566x	41,586x	27,504x
	Compression	32x (773x)	85x (1,047x)	21x (138x)	199x (1,238x)	265x (1,339x)
	Quality	15x (384x)	5.4x (228x)	22x (322x)	2.3x (200x)	1.7x (112x)
	Time	29s (51s)	13s (32s)	0.7s (6.3s)	6.3s (16s)	4.3s (13s)
	Timeouts	0.1%	0.0%	0.0%	0.0%	0.0%

Average values of the quality, compression, and time metrics are computed through the geometric mean (the arithmetic mean is in parentheses).

Table 5: Impact of constant values injection (C-Full, 1h TO)

	GREYSYNTH with Xyntia		GREYSYNTH with XSMIR	
	No injection	Injection	Not injection	Injection
Success	93.2%	93.5%	88.9%	88.7%
False positive	5.5%	5.1%	9.8%	10%
No result	1.3%	1.4%	1.3%	1.3%
Compression max	27,504x	27,504x	41,586x	25,013x
Compression	200x (1,152x)	265x (1,339x)	141x (779x)	132x (789x)
Quality	2.3x (117x)	1.7x (112x)	3.3x (264x)	3.5x (268x)
Time	11s (24s)	4.3s (13s)	18s (69s)	21s (75s)
Timeouts	0.0%	0.0%	0.0%	0.0%

Average values of the quality, compression, and time metrics are computed through the geometric mean (the arithmetic mean is in parentheses).

Table 6: Impact of the internal synthesizer on GREYSYNTH (C-Full dataset, 1h TO)

Timeout		SYNTIA	cvc5	Xyntia	XSMIR
2s	Success	85.6%	91.7%	93.5%	88.7%
	False positive	5.3%	5.8%	5.1%	10.0%
	No result	8.2%	2.5%	1.4%	1.3%
	Compression max	20,168x	26,594x	27,504x	25,014x
	Compression	52x (354x)	159x (845x)	265x (1,339x)	132x (789x)
	Quality	8.8x (294x)	2.9x (180x)	1.7x (112x)	3.5x (268x)
	Time	112s (303s)	22s (46s)	4.3s (13s)	21s (75s)
	Timeouts	2.2%	0.0%	0.0%	0.0%
5s	Success	81.3%	92.1%	93.3%	91.8%
	False positive	5.0%	5.5%	5.4%	6.9%
	No result	13.7%	2.4%	1.3%	1.3%
	Compression max	26,594x	26,594x	29,704x	41,586x
	Compression	53x (468x)	165x (863x)	265x (1,380x)	197x (966x)
	Quality	8.7x (298x)	2.8x (172x)	1.7x (104x)	2.3x (168x)
	Time	156s (379s)	43s (101s)	6.0s (22s)	32s (106s)
	Timeouts	2.0%	0.2%	0.0%	0.0%
10s	Success	79.3%	92.4%	93.2%	90.9%
	False positive	5.1%	5.2%	5.4%	7.8%
	No result	15.6%	5.2%	1.4%	1.3%
	Compression max	26,594x	27,504x	29,704x	41,586x
	Compression	55x (507x)	174x (1,011x)	266x (1,373x)	196x (1,066x)
	Quality	8.5x (385x)	2.7x (116x)	1.7x (96x)	2.4x (158x)
	Time	231s (546s)	73s (173s)	7.9s (37s)	46s (175s)
	Timeouts	3.5%	0.3%	0.0%	0.1%

Average values of the quality, compression, and time metrics are computed through the geometric mean (the arithmetic mean is in parentheses).

Nonetheless, grey-box solutions are not entirely immune to sophisticated protections. First, as the deobfuscation horizon expands, the size and (thus) the number of input variables of the local ASTs

increase rapidly, which significantly extends the search space and degrades synthesis performance.

```

1 int x_val = x;
2 int y_val = y;
3 if ((x_val * x_val + x_val) % 2 == 0){
4     // Always true (opaque predicate)
5     return x_val * y_val + x_val;
6 } else {
7     return x_val * x_val + y_val;
8 }

```

Listing 1: Example code snippet with an opaque predicate

Yet, the impact of this weakness remains somewhat limited, as designing large, systematic semantic transformations preserving global equivalence remains a non-trivial challenge; the few proposed schemes [50] are vulnerable to algebraic simplification techniques [40] and remain brittle.

Second, grey-box synthesis may also suffer from its blackbox sampling, particularly when dealing with point functions or sparse conditional branches where the sampling can simply miss the important but very rare behaviours. For instance, if an obfuscated basic block embeds a key-checking conditional branch activated by a single specific input value, uniform sampling will almost certainly fail to trigger it, leading the synthesizer to ignore this important behaviours. Still, note that the code itself should clearly hide this branch (e.g., through branchless encoding), otherwise the iterative decomposition of subgoals in GREYSYNTH might reveal it.

Locating reverse window. Throughout our evaluation, we always assume a well-defined reverse window, by providing GREYSYNTH with precise expressions. While identifying exact deobfuscation boundaries remains a hard and orthogonal challenge, typically addressed via manual reverse engineering [22, 39] together with some degree of automated support [10, 11, 14], GREYSYNTH’s architecture significantly mitigates boundary sensitivity. Unlike traditional black-box synthesizers that strictly depend on the provided reverse window, GREYSYNTH can synthesize simplified relationships at various steps *within* the reverse window, by capturing internal intermediate expressions.

Consider the nested construct in Listing 2. Given the full boundaries (Lines 1–3), GREYSYNTH directly synthesizes the full expression $((a - (b \oplus 0xff)) + (c \cdot (d \& 0xffffffff00000000)))$. Given only Lines 2–3, it extracts $(x + (c \cdot (d \& 0xffffffff00000000)))$, and given Line 3 alone, it returns $x + y$. All three represent the optimal expression for their respective scope. On the other hand, pure black-box tools such as SYNTIA or XYNTIA succeed on Line 3 but time out for the two other cases (10 minutes budget), as the target expressions are too complex – mixing rather large sizes and non-trivial constant values.

```

1 int x = (a - (b ^ 0xff));
2 int y = c * (d & 0xffffffff00000000);
3 int res = (x | y) + (x & y); // MBA for (x + y)

```

Listing 2: Code snippet illustrating boundary flexibility.

7 Related work

In this section we discuss related works on *black-box* and *grey-box* deobfuscation as well as MBA simplification.

Input/output-based program synthesis. The first I/O-based program synthesis work we could find traced back to ESCHER, a tool refining its synthesis by interacting with the users who would prompt I/O relationships. Later, SYNTIA [11] introduced I/O-based program synthesis applied to bitvector expressions targeting deobfuscation tasks. It uses a stochastic algorithm based on *Monte-Carlo Tree Search*. Menguy et al. [34] proposed to use local-search based algorithm instead of Monte-Carlo Tree Search and introduced XYNTIA. XYNTIA has been extended by Attias et al. [4] to combine the stochastic search with inference rules, allowing the synthesis of more complex expressions such as polynomials.

The *Programming Language* community introduced *Syntax-Guide Synthesis*, whose sub-category *Programming-by-Example* includes I/O-based program synthesis. The tools proposed heavily rely on *SMT* solvers to perform *enumerative* or *deductive* synthesis. The main tools dedicated to Programming-by-Example (PbE) on bitvector input/output relationship are cvc5 and DRYADSYNTH, although we have only focused our study using cvc5 [8] since previous work [4] have proved that DRYADSYNTH [23] was presenting sub-par performance on deobfuscation targets.

Grey-box deobfuscation. Extensive discussion have already been written on *grey-box* state-of-the-art [22, 29]. Succinctly, QSYNTH introduced the concept by matching semantics of global subtrees by their input/output relationship with a pre-computed table mapping semantics to an optimally small representation. This suffered many limitations such as lack of rigidity as well as awareness of local simplification rules. ProMBA on the other hand combines synthesis on locally truncated subtrees with advanced algebraic simplification techniques, however their *black-box* approach was too superficial to be properly exploited.

MBA simplification. There are currently two state-of-the-art methods to simplify MBA expressions : algebraic attacks and synthesis. *Algebraic-based simplification* tools have been introduced by Liu et al. with MBA-Blast [31]. They use n to 1 bit equivalence theorem to reduce the MBA expression into a simple basis. Current state-of-the-art algebraic solver is GAMBA [40] which can also handle some advanced non-linear MBA rewriting. *Synthesis-based simplification* such as ProMBA [29] rely on synthesis to recursively break down the MBA expression into a simpler one.

8 Conclusion

In this paper, we formalize *grey-box* deobfuscation as a principled abstraction layer between syntax and semantics, addressing the lack of a clear definition in prior work. In this view, structural information is leveraged to decompose complex expressions into tractable local synthesis problems, while observational semantics provides a practical notion of equivalence. Building on this perspective, we introduce GREYSYNTH, a context-aware grey-box framework that systematically combines structural guidance with black-box synthesis. By integrating observational equivalence-driven simplification (OES) and context-aware local synthesis (CHS), GREYSYNTH overcomes key limitations of prior approaches, including syntactic

complexity explosion, polluted truncated trees, and semantic context blindness. Our experimental evaluation shows that GREYSYNTH consistently simplifies complex obfuscated expressions to human-interpretable forms, outperforming existing white-box, grey-box, and black-box approaches in both effectiveness and scalability. In particular, GREYSYNTH remains robust across obfuscation levels and expression classes, while being largely synthesizer-agnostic. On MBA-specific benchmarks, it achieves competitive results with dedicated algebraic simplifiers, while significantly extending beyond their scope to general expressions. Overall, this work demonstrates that grey-box deobfuscation is best understood not as a specific technique, but as a unifying paradigm that enables scalable deobfuscation by reconciling structure and semantics. We believe this perspective opens new directions for deobfuscation research beyond the traditional black-box and white-box dichotomy.

Acknowledgments

This work has benefited from grants managed by the National Research Agency under France 2030 with references “ANR-22-PECY-0007” and “ANR-23-CMAS-0014”; from the BPI under Plan France 2030 with reference DOS233319/00; and from the European Union’s Horizon Europe research and innovation programme ENSEMBLE under grant agreement No 101168360.

References

- [1] [n. d.]. OMVLL. <https://github.com/open-obfuscator/o-mvll>.
- [2] [n. d.]. SyGuS: Syntax-guided synthesis. <https://sygus.org/comp/2019/>
- [3] Rajeev Alur, Rastislav Bodík, and Garvit et al. Juniwal. 2013. Syntax-guided synthesis. In *Formal Methods in Computer-Aided Design, FMCAD*. IEEE.
- [4] Vidal Attias, Nicolas Bellec, Grégoire Menguy, Sébastien Bardin, and Jean-Yves Marion. 2025. Augmenting Search-based Program Synthesis with Local Inference Rules to Improve Black-box Deobfuscation. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security* (Taipei Taiwan, 2025-11-19). ACM, 4529–4543. doi:10.1145/3719027.3765134
- [5] Soyeon Baek and Sungho Lee. 2026. Deobfuscation of Control Flow Flattening Based on Abstract Interpretation. *IEEE Transactions on Software Engineering* 52, 3 (2026), 1110–1124. doi:10.1109/TSE.2026.3659437
- [6] Sebastian Banescu, Christian Collberg, and Vijay et al. Ganesh. 2016. Code obfuscation against symbolic execution attacks. In *Annual Conference on Computer Security Applications, ACSAC*.
- [7] Sebastian Banescu, Christian S. Collberg, Vijay Ganesh, Zack Newsham, and Alexander Pretschner. 2016. Code obfuscation against symbolic execution attacks. In *Proceedings of the 32nd Annual Conference on Computer Security Applications, ACSAC 2016, Los Angeles, CA, USA, December 5-9, 2016*, Stephen Schwab, William K. Robertson, and Davide Balzarotti (Eds.). ACM, 189–200. <http://dl.acm.org/citation.cfm?id=2991114>
- [8] Haniel Barbosa, Clark Barrett, and Martin Brain et al. 2022. cvc5: A Versatile and Industrial-Strength SMT Solver. In *Tools and Algorithms for the Construction and Analysis of Systems (TACAS)*. Springer.
- [9] Sébastien Bardin, Robin David, and Jean-Yves Marion. 2017. Backward-Bounded DSE: Targeting Infeasibility Questions on Obfuscated Codes. In *Symposium on Security and Privacy, SP 2017*. IEEE.
- [10] Tim Blazytko. [n. d.]. Binary Ninja plugin to identify obfuscated code and other interesting code constructs. https://github.com/mrphrazer/obfuscation_detection.
- [11] Tim Blazytko, Moritz Contag, Cornelius Aschermann, and Thorsten Holz. 2017. Syntia: Synthesizing the Semantics of Obfuscated Code. In *26th USENIX Security Symposium (USENIX Security 17)* (Vancouver, BC, 2017). USENIX Association, 643–659. <https://www.usenix.org/conference/usenixsecurity17/technical-sessions/presentation/blazytko>
- [12] David Brumley, Cody Hartwig, and Zhenkai Liang et al. 2008. Automatically Identifying Trigger-based Behavior in Malware. In *Botnet Detection: Countering the Largest Security Threat*. Springer.
- [13] José Cambronero, Sumit Gulwani, and Vu Le et al. 2023. FlashFill++: Scaling Programming by Example by Cutting to the Chase. *Proc. ACM Program. Lang.* POPL (2023).
- [14] Roxane Cohen, Robin David, Florian Yger, and Fabrice Rossi. 2025. Identifying obfuscated code through graph-based semantic analysis of binary code. *Applied Network Science* 10, 1 (2025).
- [15] C. Collberg, S. Martin, J. Myers, and B. Zimmerman. [n. d.]. The Tigress C Diversifier/Obfuscator. <http://tigress.cs.arizona.edu/>
- [16] Christian Collberg and Jasvir Nagra. 2009. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*. Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection.
- [17] Christian Collberg, Clark Thomborson, and Douglas Low. 1997. A taxonomy of obfuscating transformations.
- [18] Christian Collberg, Clark Thomborson, and Douglas Low. 1998. Manufacturing cheap, resilient, and stealthy opaque constructs. In *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*.
- [19] Kevin Coogan, Gen Lu, and Saumya K. Debray. 2011. 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*, Yan Chen, George Danezis, and Vitaly Shmatikov (Eds.). ACM, 275–284. doi:10.1145/2046707.2046739
- [20] Robin David. 2021. Greybox Program Synthesis: A New Approach to Attack Dataflow Obfuscation. *Black Hat USA* (2021).
- [21] Robin David, Sébastien Bardin, and Thanh Dinh et al. Ta. 2016. BINSEC/SE: A dynamic symbolic execution toolkit for binary-level analysis. In *Software Analysis, Evolution, and Reengineering (SANER)*. IEEE.
- [22] Robin David, Luigi Coniglio, and Mariano Ceccato. 2020. QSynth - A Program Synthesis based approach for Binary Code Deobfuscation. In *Proceedings 2020 Workshop on Binary Analysis Research* (San Diego, CA, 2020). Internet Society. doi:10.14722/bar.2020.23009
- [23] Yuantian Ding and Xiaokang Qiu. [n. d.]. Enhanced Enumeration Techniques for Syntax-Guided Synthesis of Bit-Vector Manipulations. ([n. d.]).
- [24] UNH SoftSec Group. 2021. *MBA-Solver Code and Dataset*. <https://github.com/softsec-unh/MBA-Solver>.
- [25] Sumit Gulwani. 2016. Programming by examples. *Dependable Software Systems Engineering* 45, 137 (2016), 3–15.
- [26] Sumit Gulwani, Aleksandr Polozov, Rishabh Singh, et al. 2017. Program synthesis. *Foundations and Trends® in Programming Languages* (2017).
- [27] Jaehyoung Lee. 2026. ProMBA. <https://github.com/astean1001/ProMBA>.
- [28] Johannes Kinder. 2012. Towards Static Analysis of Virtualization-Obfuscated Binaries. In *Working Conference on Reverse Engineering, WCRE*.
- [29] Jaehyung Lee and Woosuk Lee. 2023. Simplifying Mixed Boolean-Arithmetic Obfuscation by Program Synthesis and Term Rewriting. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security* (Copenhagen Denmark, 2023-11-15). ACM, 2351–2365. doi:10.1145/3576915.3623186
- [30] Binbin Liu, Weijie Feng, Qilong Zheng, Jing Li, and Dongpeng Xu. 2021. Software obfuscation with non-linear mixed boolean-arithmetic expressions. In *International Conference on Information and Communications Security*. Springer, 276–292.
- [31] Binbin Liu, Junfu Shen, Jiang Ming, Qilong Zheng, Jing Li, and Dongpeng Xu. 2021. {MBA-Blast}: Unveiling and Simplifying Mixed {Boolean-Arithmetic} Obfuscation. In *30th USENIX Security Symposium (USENIX Security 21)*. 1701–1718.
- [32] Zohar Manna and Richard Waldinger. 1980. A Deductive Approach to Program Synthesis. *ACM Transactions on Programming Languages and Systems* (1980).
- [33] Benjamin Mariano, Ziteng Wang, Shankara Pailoor, Christian S. Collberg, and Isil Dillig. 2024. Control-Flow Deobfuscation using Trace-Informed Compositional Program Synthesis. *Proc. ACM Program. Lang.* 8, OOPSLA2 (2024), 2211–2241. doi:10.1145/3689789
- [34] Grégoire Menguy, Sébastien Bardin, Richard Bonichon, and Caum De Souza Lima. 2021. Search-Based Local Black-Box Deobfuscation: Understand, Improve and Mitigate. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security* (Virtual Event Republic of Korea, 2021-11-12). ACM, 2513–2525. doi:10.1145/3460120.3485250
- [35] Jiang Ming, Dongpeng Xu, Li Wang, and Dinghao Wu. 2015. LOOP: Logic-Oriented Opaque Predicate Detection in Obfuscated Binary Code. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, Denver, CO, USA, October 12-16, 2015*, Indrajit Ray, Ninghui Li, and Christopher Kruegel (Eds.). ACM, 757–768. doi:10.1145/2810103.2813617
- [36] nhpcc502. 2026. MBA Obfuscator. <https://github.com/nhpcc502/MBA-Obfuscator>.
- [37] Mathilde Ollivier, Sébastien Bardin, and et al. Bonichon. 2019. How to kill symbolic deobfuscation for free (or: unleashing the potential of path-oriented protections). In *Annual Computer Security Applications Conference*.
- [38] Mathilde Ollivier, Sébastien Bardin, and Richard et al. Bonichon. 2019. Obfuscation: where are we in anti-DSE protections?(a first attempt). In *Workshop on Software Security, Protection, and Reverse Engineering*.
- [39] Alex Petrov. 2023. Hands-Free Binary Deobfuscation with gooMBA. <https://hex-rays.com/blog/deobfuscation-with-goomba>.
- [40] Benjamin Reichenwallner and Peter Meerwald-Stadler. 2023. Simplification of General Mixed Boolean-Arithmetic Expressions: GAMBA. In *2023 IEEE European Symposium on Security and Privacy Workshops (EuroS&PW)* (Delft, Netherlands, 2023-07). IEEE, 427–438. doi:10.1109/EuroSPW59978.2023.00053
- [41] Robin David. 2026. QSynthesis. <https://github.com/quarkslab/qsynthesis>.

- [42] Alan Romano, Daniel Lehmann, Michael Pradel, and Weihang Wang. 2022. Wobfuscator: Obfuscating JavaScript Malware via Opportunistic Translation to WebAssembly. In *43rd IEEE Symposium on Security and Privacy, SP 2022, San Francisco, CA, USA, May 22–26, 2022*. IEEE, 1574–1589. doi:10.1109/SP46214.2022.9833626
- [43] Jonathan Salwan, Sébastien Bardin, and Marie-Laure Potet. 2018. Symbolic deobfuscation: from virtualized code back to the original. In *DIMVA*.
- [44] Moritz Schloegel, Tim Blazytko, and Moritz Contag et al. 2022. Loki: Hardening Code Obfuscation Against Automated Attacks. In *USENIX Security*.
- [45] Sebastian Schrittwieser, Stefan Katzenbeisser, and Johannes Kinder et al. 2016. Protecting Software Through Obfuscation: Can It Keep Pace with Progress in Code Analysis? *ACM Comput. Surv.* (2016).
- [46] Monirul Sharif, Andrea Lanzi, Jonathon Giffin, and Wenke Lee. 2009. Automatic reverse engineering of malware emulators. In *2009 30th IEEE Symposium on Security and Privacy*. IEEE.
- [47] Jon Stephens, Babak Yadegari, and Christian Collberg et al. 2018. Probabilistic Obfuscation Through Covert Channels. In *IEEE EuroS&P*.
- [48] Babak Yadegari and Saumya Debray. 2015. Symbolic Execution of Obfuscated Code. In *Conference on Computer and Communications Security*.
- [49] Babak Yadegari, Brian Johannsmeyer, Ben Whitely, and Saumya Debray. 2015. A Generic Approach to Automatic Deobfuscation of Executable Code. In *Symposium on Security and Privacy, SP*.
- [50] Yongxin Zhou, Alec Main, and Gu et al. 2007. Information Hiding in Software with Mixed Boolean-arithmetic Transforms. In *Conference on Information Security Applications*.

A Open Science

As a commitment to open science, we provide a public artifact containing the paper’s datasets and a binary of GreySynth to both run the paper’s experiments and check the claims, and we also provide instructions on how to repurpose GreySynth, for example to use alternative datasets or test on other examples. The artifact is hosted on Zenodo and can be accessed via <https://doi.org/10.5281/zenodo.22649858>.

Similarly, we will release the code of GreySynth in a public code repository, although we have not made a definitive choice of the format at the time of publication.

B Ethical Considerations

Deobfuscation provides tools and techniques that can be used either by malware analysts and obfuscator vendors.

Stakeholders and potential impact. We identify the following stakeholders and potential impacts.

- *Malware analysts, law enforcement agencies.* In the context of malware analysis, deobfuscation frameworks can be used by various actors, from private companies specializing in malware analysis to state and law enforcement agencies. Hence, research and advances in deobfuscation techniques and tools is positive for these actors, as it helps them to cope with highly obfuscated codes, typically APT-grade malware.
- *IP protection solution providers.* For obfuscator vendors, the development of new deobfuscation methods is more double-edged. Indeed, new deobfuscation techniques could make vendors’ solutions (partly) obsolete. Still, our paper mainly targets data-flow obfuscation while commercial obfuscators often combine multiple kinds of transformations (data-flow but also control-flow obfuscation, anti-dynamic analysis). Hence, it is unlikely that this deobfuscation research will completely break vendors’ solutions. Moreover, deobfuscation is also used by obfuscator vendors to assess the security of their product and calibrate the degree of obfuscation to ensure intellectual property protection.
- *IP protection solution users.* For the obfuscation products end-users, the discovery of new deobfuscation methods brings the risk that the bought obfuscation solution becomes obsolete. Still, as presented above, IP protection vendors constantly adapt their solutions to new deobfuscation. Moreover, open-sourcing deobfuscation frameworks gives end users a way to assess the security of the obfuscation solution and raises awareness of new potential threats and vulnerabilities.

Decision to conduct the research and publish. Dual use is unfortunately common in security, and the community generally considers that open research brings more benefits than harm. The present work stands in this long-lasting line of research on obfuscation and deobfuscation techniques. By proposing open approaches that help legitimate deobfuscation tasks (malware analysis) and contribute to raising awareness of latest attacks to IP protection solution providers and customers, we believe that this work indeed has overall positive outcome.