Augmenting Search-based Program Synthesis with Local Inference Rules to Improve Black-box Deobfuscation

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

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

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

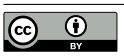
Code obfuscation aims to protect programs from reverse engineering, with applications ranging from intellectual property protection to malware hardening. Recent works on black-box analyses propose to leverage program synthesis in order to infer the semantics of highly obfuscated code blocks. Being fully *black-box*, these approaches are immune to *syntactic* complexity and can thus bypass standard obfuscation mechanisms. Yet, they are restricted by their synthesis capabilities and can only be applied to *semantically* simple code blocks. It explains why they have mainly been used on virtual machine handlers, where behaviors are usually simple enough. Applying black-box deobfuscation at scale beyond virtualization is still an open problem, notably because black-box methods cannot synthesize complex behaviors involving, for example, arbitrary constant values or affine or polynomial relations over mixed-boolean-arithmetic expressions. In this article, we show how to combine *search-based program synthesis* with *local inference rules*, resulting in a new method named *Search Modulo Inference Rules* (Smir) that boosts search-based program synthesis while keeping its generality and flexibility. We instantiate Smir with inference rules for hard synthesis problems like arbitrary constant values and affine or polynomial relations over mixed boolean expressions, yielding the new black-box deobfuscation tool: XSmir. Experiments demonstrate that XSmir significantly outperforms prior black-box deobfuscators, synthesizing overall 76% and 84% of the expressions from our real-world obfuscated and non-obfuscated benchmarks where prior works recover 63% and 55%, together with 2 to 3 times less false positive and slightly improved compression rate.

CCS Concepts

• **Security and privacy → Software reverse engineering**; • **Computing methodologies → Randomized search**.

Keywords

Deobfuscation, reverse engineering, program synthesis



CCS '25, Taipei, Taiwan

ACM ISBN 979-8-4007-1525-9/2025/10
https://doi.org/10.1145/3719027.3765134

ACM Reference Format:
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 (CCS '25), October 13–17, 2025, Taipei, Taiwan.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3719027.3765134

1 Introduction

Obfuscation [18, 19] aims to protect software from reverse engineering. It translates a program P into a functionally equivalent program P_o, harder to analyze. As cryptography has not yet provided a bullet-proof solution for this problem, the obfuscation community mainly focuses on program-analysis-based techniques [19], leading to a cat-and-mouse game between defenders and attackers.

While obfuscation is used to protect *Intellectual Property* and other valuable software assets, it is also used to protect malware. Thus, automated *deobfuscation* methods [12, 21, 39, 50, 51] have been proposed to cope with the quick advances in obfuscation. Given an obfuscated program P_o, the goal here is to simplify it into a simpler yet functionally equivalent program P^* – ideally P^* should be as simple as the original unprotected code P.

In the last decade, *white-box deobfuscation* techniques, lifting the latest methods of static program analysis, have proved highly effective against many standard obfuscation schemes like opaque predicates [7, 54, 60] or virtualization [32, 51, 54]. However, several obfuscation methods significantly impair white-box analysis. Dedicated protections, including Mixed Boolean Arithmetic (MBA) expressions [63], covert channels [55] or path-oriented protections [3, 44, 45, 59], have emerged, many of which increase the syntactic complexity of the original code to break white-box approaches.

The promises of black-box deobfuscation. New deobfuscation methods, based on program synthesis, have emerged to simplify heavily obfuscated local code snippets [12, 39]. These methods rely only on *input-output* (I/O) *observations* to infer the code snippet behavior. Through synthesis, they explore a *space of candidate expressions* generated by an *inference grammar*, seeking an expression that mimics the I/O observations. Unlike white-box approaches, black-box ones are immune to syntactic complexity, but the analyzed code must be semantically simple to synthesize it. This explains why black-box methods primarily focused on virtualized code: handlers

implementing a custom ISA usually stay semantically simple, even in the presence of strong obfuscation [12, 39].

Problem. Applying black-box deobfuscation beyond the virtualization use case is still an open problem. It requires a sufficiently expressive synthesis to capture the semantics of many distinct behaviors. Still, some behaviors stay beyond the reach of state-of-the-art synthesizers – e.g., arbitrary constant values, affine or polynomial combinations of bitvector expressions. Unfortunately, software often includes them. For instance, in the Core utils, Curl, OpenSSL, and FFmpeg binaries, 82% of the basic blocks contain constant values. This hinders the use of black-box methods at scale.

Our solution. We propose *Search Modulo Inference Rules* (Smir), the first synthesis approach combining *search* and *inference*, enabling the synthesis of usually hard-to-recover expressions while keeping a generic approach. Inference rules improve the search in two ways: (1) at each step of the search, inference rules may directly *elevate* the current *candidate solution* into a *definitive solution*, ending the synthesis process; (2) if no direct match is found, the search process itself is altered, favoring candidate solutions that will more likely be elevated to a definitive solution.

Smir can be adapted to any black-box search heuristics and a wide class of inference rules. Specifically, we instantiate Smir to handle behaviors that are beyond the reach of state-of-the-art synthesizers: *arbitrary constant values* (masks, offsets, etc.), as well as *affine* and *polynomial* expressions involving mixed Boolean-Arithmetic terms. We implement Smir on top of the Xyntia black-box deobfuscator [39], yielding a prototype XSmir that largely outperforms prior works: it synthesizes on average 76% of the expressions from code obfuscated with Tigress [17], VMProtect [57], OLLVM [31] and Loki [53], against 63% for the prior work. Over non-obfuscated software, XSmir also synthesizes 84% of the expressions included in the Core Utils, OpenSSL, Curl, and FFmpeg, when prior work only reaches 55%. Note also that XSmir reports, on average, two to three times fewer false positives than prior work on deobfuscation.

Contributions. Our contributions are the following:

(1) We propose *Search Modulo Inference Rules* (Smir, Section 5), a new program synthesis scheme that extends search with inference rules to generate usually hard-to-synthesize expressions. Inference rules serve two roles: they directly extend some candidate solutions into definitive solutions if possible, otherwise they direct the search toward such candidate solutions;
(2) We implement Smir (Section 5.3) on top of the Xyntia deobfuscator [39] with the inference rules from Section 5.2 (offsets, masks, affine and polynomial expressions, etc.). It yields XSmir, the first search-based synthesizer handling these cases efficiently. Internal analysis (Section 6.6) shows that it benefits of each Smir ingredient, and that the technique works for all typical objective functions;
(3) We evaluate XSmir (Section 6) against the state-of-the-art black-box deobfuscators Syntia [12] and Xyntia [39], as well as the *program synthesizers* CVC4 [9], CVC5 [6] and DryadSynth [22], showing that it outperforms them on both real obfuscated and non-obfuscated code. For example, XSmir synthesizes up to +16 (resp. +74) points of percentage more expressions than Xyntia (resp., Syntia) on obfuscated code, while dividing by two to three the number of false positive and producing more compressed expressions. We also compared XSmir against the state-of-the-art white-box deobfuscators ProMBA [33] and Gamba [49] on MBA expressions. Results show that XSmir always successfully simplifies more expressions than ProMBA and Gamba, recovering almost twice as many expressions for the most complex MBA obfuscation.

As a conclusion, *Search Modulo Inference Rules* (Smir) provides a novel and powerful tool for reversers to apply black-box deobfuscation at scale. It also demonstrates the interest of combining both search and inference for program synthesis problems. *All the code and scripts needed to replicate our results are available at* https://zenodo.org/records/17036259.

2 Background

2.1 Programming-by-example

Programming-by-example (PBE) [27] is a special case of program synthesis [28]. It takes a set of input-output examples – referred to as the input-output (I/O) specification $\mathcal{S}$, and a syntactic grammar $\mathcal{G}$ that defines the search space [2]. PBE then tries to infer an expression $e \in \mathcal{G}$ (i.e., a program) mimicking the observed behaviors in $\mathcal{S}$. More formally, we say that $e \in \mathcal{G}$ verifies the specification $\mathcal{S}$, noted $e \models \mathcal{S}$, iff for all $i, o \in \mathcal{S}, e(i) = o$. Three main approaches have been studied to efficiently search such a solution:

- *Enumerative search* [2, 6, 9, 22] exhaustively explores the space of possible expressions in increasing size order, typically applying aggressive pruning to eliminate infeasible candidates early. While general, this approach scales poorly with expression size due to the combinatorial explosion of the search space.
- *Deductive search* [2, 28] derives expressions directly from the specification using logical inference rules. It is often less efficient than enumerative search [2], and may require domain-specific insights for a more efficient reasoning [28].
- *Stochastic search* [2, 52] generates candidate expressions randomly, often guided by heuristics such as distance to expected outputs. This method can sometimes discover larger or more complex expressions that would be missed by enumerative search.

Interestingly, most state-of-the-art PBE synthesizers use enumerative search, for example: FlashFill [15, 26] (string theory) in Microsoft Excel; CVC4/5 (string, integer and bitvector theories) [6, 9]; or DryadSynth [22] (string and bitvector theories). In the following, we focus on synthesis over the bitvector theory, which is especially useful in binary code deobfuscation as the data manipulated are bitvectors. Unfortunately, synthesis over this theory is particularly challenging due to two key difficulties: the sparsity of useful information in I/O examples and the high degree of semantic equivalence between distinct syntactic forms. This makes guiding the synthesis difficult and the search space harder to prune [22].

2.2 Obfuscation and (Black-box) Deobfuscation

Obfuscation is a family of methods that translate a program P into an equivalent program P_o, harder to understand and analyze. While the

theoretical feasibility of obfuscation is still debated,[1] practical obfuscators take a more pragmatic program analysis view [17, 31, 46, 57], with recipes of "hard-to-revert in practice" program transformations. Obfuscation is used in two main contexts. It helps protect the intellectual property embedded in software, e.g., proprietary algorithms or secret keys. For instance, on the Google Play Store, 50% of the most popular Android applications are obfuscated [58]. Conversely, it is also leveraged by malware authors to prevent the detection and comprehension of malicious behavior [7, 14, 51].

Deobfuscation. Given an obfuscated program P_o coming from a clear program P, deobfuscation aims to simplify it back to a simpler equivalent program P^*. Studying automatic deobfuscation techniques is crucial for security, as it allows both to evaluate current obfuscation products and practices in the best effort manner, and to empower malware analysts.

White-box deobfuscation. [3, 7, 14, 32, 51, 54, 60], based on simplifications from compilation and program analyses (static and dynamic), have been largely investigated and shown to be very powerful in some contexts. However, efficient countermeasures recently emerged [3, 44, 45, 55, 59, 63], often exploiting their sensitivity to syntactic complexity. For the particular case of mixed-boolean arithmetic (MBA) obfuscation [63], recent algebraic attacks led to new specific white-box deobfuscation frameworks such as MBABlast [36], MBASolver [25], ProMBA [34] and Gamba [49].

Black-box deobfuscation. Recently, the Syntia [12] and Xyntia [39] papers introduced *black-box deobfuscation*, which relies on search-based (black-box) program synthesis over the bitvector theory [28] to infer the semantics of *highly obfuscated blocks of code*. Black-box methods leverage the fact that obfuscation only increases the code syntactic complexity without altering its semantics. For instance, the expression $x + y$ can be expressed as $(x \vee 2y) \times 2 - (x \oplus 2y) - y$ (standard MBA transformation [63]). The second expression is syntactically more complex, but is semantically equivalent. Given an expression e, black-box methods rely solely on I/O examples to synthesize it. Hence, they are not impacted by the syntactic complexity of e. However, they cannot handle e if it is too semantically complex.

Obviously, black-box deobfuscation does not aim for the automated recovery of the full code, as it is likely infeasible. Instead, we consider that the users are skilled reversers, with high understanding of assembly code and program analysis, and our goal is to empower them with dedicated tools to speed up local reverse engineering tasks. Especially, it could be integrated as an interactive plugin for GUI-based reverse tools like Ghidra [41] or IdaPro [23] (cf. GooMBA [47] or QSynth [21]). In this setting, the reverser selects the subpart of code of interest, and then launches deobfuscation. A more automated mode could also be envisioned, applying black-box deobfuscation on batches of code snippets provided by an obfuscation detector [11, 16].

The typical workflow of black-box deobfuscation. While the core part of black-box deobfuscation, namely programming-by-example (cf. Section 2.1), is automated, the whole process implies some preliminary steps, possibly automated [12] or handmade:

- Given a binary under analysis, it first requires splitting it into different *reverse windows*, i.e., parts of code that the reverser wants to analyze. *In principle*, these reverse windows only need to fulfill the following constraint: have a single entry point and a single exit point. Hence, they can include conditionals, but all the paths must, in the end, merge to a single exit point inside the reverse window. Yet, *in practice*, black-box deobfuscation has mostly been applied to blockwise VM handlers [12, 39];
- Next, the reverser must identify the outputs of interest for each reverse window and determine the corresponding inputs. This can be done manually or automatically, e.g., by using taint analysis for outputs and data dependencies for inputs.

Each output and associated input define a *reverse goal*: find an expression e that represents the reverse window behavior.

Finally, in order to set up the programming-by-example algorithm, we need to choose a suitable inference grammar $\mathcal{G}$ and a sampling strategy. The inference grammar defines the set of expressions possibly generated by the synthesizer. State-of-the-art tools such as Syntia and Xyntia typically use variations of the grammar presented in Table 1, that generates Mixed Boolean-Arithmetic expressions.[2] The sampling strategy specifies how the input of the reverse window are sampled: how many samples are generated and which values are assigned to each input. This sampling strategy should be chosen carefully. If the number of samples is too small or the values are not diverse enough, the result might be incorrect. Conversely, too many samples degrades synthesis performance. Syntia and Xyntia consider respectively 50 and 100 I/O examples, combining a fixed set of input deemed useful (e.g., zeros, ones, minus ones, etc) with randomly generated ones [12, 39].

Applying the sampling strategy over the reverse window yields the I/O specification $\mathcal{S}$. Given h, the *hidden* relationship we are looking for, this I/O specification $\mathcal{S}$ approximates the semantics of h, i.e., for all $(i, o) \in \mathcal{S}, h(i) = o$.

The goal is to infer a (simple) expression that matches the same input-output observations, in the same spirit as programming-by-example (Section 2.1).

Still, the solution may be non-equivalent to the target expression (false positive). This is standard in the black-box setting [12, 39], and considered fine as long as results are rarely incorrect in practice.

Synthesis in black-box deobfuscation. Recent works on black-box deobfuscation (cf. Section 2.2) proposed their own PBE synthesizers over the bitvector theory. They digress from the aforementioned synthesizers by performing stochastic search instead of enumerative search. For instance, Syntia relies on Monte-Carlo-Tree-Search (MCTS) [13] and Xyntia relies on local-search algorithms [37, 56]. Both use an objective function to guide the search toward more likely interesting solutions. Surprisingly, stochastic search was shown to be more efficient for black-box deobfuscation than enumerative approaches [39].

We first present in Algorithm 1 the generic search-based program synthesis algorithm used in black-box deobfuscation:

- It takes as input an inference grammar $\mathcal{G}$ and an objective function $f_{\mathcal{S}}$ that evaluates how close an expression is from a target I/O specification $\mathcal{S}$. Intuitively, $f_{\mathcal{S}}$ should decrease when the

[1] Barak et al. [5] showed that no virtual black-box obfuscator exists. Still, the notion of indistinguishability obfuscation [4] is under active studies.

[2] There is a tradeoff here: richer grammars are more expressive, but they also lead to larger search space and can be intractable.

Table 1: MBA inference grammar used in [39]

E	:=	$\diamond E \mid E \circ E \mid$ 0x1 $\mid$ EAX $\mid$ EBX $\mid \ldots$
$\diamond$	:=	$-_1 \mid \neg$
$\circ$	:=	$+ \mid - \mid \times \mid \wedge \mid \vee \mid \oplus$

Algorithm 1 Generic search-based program synthesis

Input: The objective function $f_{\mathcal{S}}$ over the I/O specification $\mathcal{S}$; inference grammar $\mathcal{G}$;
Output: An expression matching the I/O specification $\mathcal{S}$

1: $L \leftarrow \{initial\_expression(\mathcal{G})\}$
2: **while** *true* **do**
3: pick $e \in L$ according to $f_{\mathcal{S}}$
4: **if** $f_{\mathcal{S}}(e) = 0$ **then**
5: **return** e
6: **else**
7: $e_1, ..., e_n \leftarrow next(e, \mathcal{G}, f_{\mathcal{S}})$
8: $L \leftarrow (L \setminus \{e\}) \cup \{e_1, ..., e_n\}$
9: **end if**
10: **end while**

Algorithm 2 Local-search-based program synthesis

Input: The objective function $f_{\mathcal{S}}$ over the I/O specification $\mathcal{S}$; inference grammar $\mathcal{G}$;
Output: An expression matching the I/O specification $\mathcal{S}$

1: $e \leftarrow initial\_expression(\mathcal{G})$
2: $N \leftarrow N_{max}$
3: **while** *true* **do**
4: **if** $f_{\mathcal{S}}(e) = 0$ **then**
5: **return** e
6: **else**
7: $e' \leftarrow mutate(e)$
8: **if** $f_{\mathcal{S}}(e') < f_{\mathcal{S}}(e) \vee N \leq 0$ **then**
9: $e \leftarrow e'$
10: $N \leftarrow N_{max}$
11: **else**
12: $N \leftarrow N - 1$
13: **end if**
14: **end if**
15: **end while**

candidate expression gets closer to the solution. Notably, it must nullify iff an expression matching the I/O specification is found;
- Throughout the synthesis, Algorithm 1 maintains a set L of expressions to check. At line 1, L is initialized with a unique expression. Synthesis then loops until an expression e verifying $\mathcal{S}$ has been found. To do so, it picks an expression $e \in L$ at line 3 depending on the used search algorithm. If $f_{\mathcal{S}}(e) = 0$, then e verifies $\mathcal{S}$ and e is returned (line 5). Otherwise, e is removed from L, and new candidate expressions $e_1, \ldots, e_n$ are added (line 7).

Local search approach. This generic framework depends on an underlying search algorithm. We focus in the paper on an instantiation based on local search – as found in Xyntia (depicted in Algorithm 2), since our own implementation relies also on it. The set L is removed, as local search only manipulates one expression at a time. The initial expression is usually a variable name or a constant value. The *next* procedure is inlined at lines 7 to 13. It generates a new candidate expression by randomly mutating e, only keeping the expression if it decreases the objective function. If no such expression can be found in N_{max} mutations, the search considers that a local minimum is reached and thus accepts any mutation in the hope of escaping this local minimum.

Scope. Black-box deobfuscation can simplify code snippets with simple enough semantic complexity. Such methods have already been used to simplify virtual machine handlers, even in presence of strong obfuscations like mixed-boolean-arithmetic expressions [12, 39], covert channels obfuscation [55], opaque predicates [19] or path-oriented protections [44]. Still, using black-box deobfuscation beyond the virtualization case remains an open problem. Here, we seek to extend the approach to arbitrary blocks found in real code.

3 Motivation

Real-world obfuscated code are still challenging for black-box deobfuscation. We showcase several real-world examples, providing clear examples of the challenges faced by state-of-the-art black-box deobfuscators and synthesizers, illustrating the benefits of XSmir:

- *MBA encoding of hard-to-synthesize expression (Snapchat)*: The Listing 1 displays an obfuscated expression found in Snapchat and documented in [29]. The obfuscated expression computes tv_sec = tv_sec×1000, presumably to convert seconds into milliseconds. It is protected using Mixed Boolean-Arithmetic (MBA) and has a size of 306 (number of nodes in the abstract-syntax-tree). Both Syntia and Xyntia fail to recover the original expression, even after one hour of computation, whereas XSmir handles it in 5ms. This failure stems from their inability to infer the constant value 1000 within reasonable time. Indeed, their *inference grammar* only includes the constant value 1, and all others must be derived from it. Generating 1000 would require synthesizing a complex expression like $((1{+}1{+}1{+}1{+}1){\times}(1{+}1{+}1{+}1{+}1)) \ll (1{+}1)$. As a result, the probability of producing arbitrary constant values is extremely low. Simply adding constant values to the grammar enlarges the search space and significantly degrades performance [39]. Furthermore, as obfuscation often conceals constant values [18], relying on constant values extracted from the code may not help or even lead to incorrect results;
- *Affine Encoding (Tigress)*: Tigress applies affine transformations to encode integer variables (cf. Listing 2). In this example, neither Syntia nor Xyntia can synthesize a, b or r, which hampers code understanding. In contrast, XSmir synthesizes a and b in 2ms, and r in 524ms;
- *Opaque Predicates (Xtunnel)*: The APT-grade malware Xtunnel [7] includes opaque predicates that are difficult to synthesize, such as $7y^2 - 1 \neq x^2$. While x^2 is easily handled in under 1s by Syntia, Xyntia and XSmir, synthesizing $7y^2 - 1$ is significantly harder. Syntia times out after 1h, and Xyntia takes 35 minutes to produce an expression that still requires manual simplification. XSmir, however, synthesizes the simplified version in just 6ms.

Such examples of obfuscation are common. Moreover, as shown in Table 2, after filtering out duplicate blocks, 82% of the blocks in Coreutils, Curl, OpenSSL, and FFmpeg binaries contain non-trivial

Listing 1: Example from Snapchat [29]

```
# Original: TV_SEC := TV_SEC * 1000
add     x0,sp,#0x1b8  ;struct timeval *tval
mov     x1,#0x0     ;struct timezonze *tzone
adrp    x8,0x109499000
ldr     x8,[x8, #0x1d0]
blr     x8          ;gettimeofday(tval,tzone)
ldr     x8,[sp, #0x1b8] ;tval->tv_sec
mov     x9,#0x3e8
mul     x8,x8,x9
ldrsw   x9,[sp, #0x1c0] ;tval->tv_usec
lsr     x9,x9,#0x3
mov     x10,#0xf7cf
movk    x10,#0xe353, LSL #16
movk    x10,#0x9ba5, LSL #32
movk    x10,#0x20c4, LSL #48
umulh   x9,x9,x10
mov     x10,#0xe6b3
movk    x10,#0x7dba, LSL #16
movk    x10,#0xecfa, LSL #32
movk    x10,#0xd0e1, LSL #48
add     x9,x10,x9, LSR #0x4
orr     x11,x9,x8
lsl     x11,x11,#0x1
eor     x8,x9,x8
sub     x8,x11,x8
eor     x9,x8,x10
mov     x10,#0xe6b3
movk    x10,#0x7dba, LSL #16
movk    x10,#0xecfa, LSL #32
movk    x10,#0x50e1, LSL #48
bic     x8,x10,x8
sub     x8,x9,x8, LSL #0x1 ;tv_sec *= 1000
```

Table 2: Percentage of blocks containing constant values.

	Core Utils	Curl	OpenSSL	FFmpeg	Total
All blocks	84%	81%	79%	79%	81%
Unique blocks	92%	84%	80%	81%	82%

Listing 2: Example of integers encoded with Tigress

```
void foo(int x, int y) {
    a = 1789355803 * x + 1391591831;
    b = 1789355803 * y + 1391591831;
    [ ... ] // Some code

    // Compute the encoding of x * y
    r = ((3537017619 * (a * b) - 3670706997 * a)
            - 3670706997 * b) + 3171898074;
    [ ... ] // Some code
}
```

Table 3: Syntia, Xyntia, CVC4/5, DryadSynth and XSmir on a selection of expressions found in binaries (TO = 1h).

	Syntia	Xyntia	CVC4/5	Dryad.	XSmir
Snapchat : $x \times 1000$	✗(TO)	✗(TO)	✗(TO)	✓(4.6s)	✓(2ms)
Tigress : $C_1 \times x + C_2$	✗(TO)	✗(TO)	✗(TO)	✗(OOM)	✓(2ms)
Tigress : $C_1 \times (x * y) + C_2$	✗(TO)	✗(TO)	✗(TO)	✗(OOM)	✓(524ms)
Xtunnel : $7y^2 - 1$	✗(TO)	✓(35mins)	✗(TO)	✓(20ms)	✓(6ms)
Xtunnel : $(2 \times x) \wedge$ 0x03ffffff	✗(TO)	✗ (4s)	✗(TO)	✓(3mins)	✓(45ms)
Loki : $x + 74$	✗(TO)	✗(TO)	✗(TO)	✓(75ms)	✓(5ms)
Core Utils : $C_1 - (\neg x)$	✗(TO)	✗(TO)	✗(TO)	✗(OOM)	✓(2ms)
FFMPeg : $(x \wedge \neg 1) \gg 16$	✗(TO)	✗(TO)	✗(TO)	✗(OOM)	✓(2ms)
FFMPeg : $(x \vee y) \gg 5$	✗(TO)	✗(TO)	✗(TO)	✗(OOM)	✓(3ms)
Curl : $(((x + y) \wedge 1) \oplus y)$ ror 1	✗(TO)	✗(TO)	✗(TO)	✗(OOM)	✓(2s)

✗: no result; ✗: incorrect result; ✓: correct result

constant values. Since standard obfuscation techniques preserve code semantics, it is expected that these behaviors will also appear in their obfuscated counterparts.

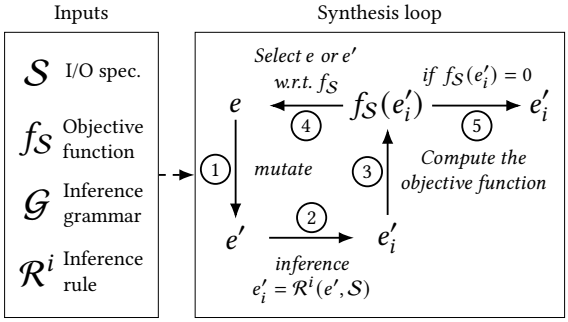


Figure 1: Black-box deobf. modulo inference rules

Inference rules. As illustrated in Table 3, Syntia and Xyntia fail to recover quickly the expressions from our challenging examples. In sharp contrast, XSmir not only succeeds across the board, but does so in under 2s — consistently producing concise, human-readable expressions. This demonstrates both its power and practicality. By augmenting search-based methods with inference rules, XSmir is able to efficiently synthesize complex, real-world expressions that remain out of reach for other existing tools.

4 Overview

We propose to extend black-box deobfuscation with Search Modulo Inference Rules (Smir). Intuitively, it aims to guide the search in the space of expressions modulo some inference rules rather than in the space of all expressions. Thus, for each candidate expression e, we apply an inference rule before computing the objective function. It leads to a new candidate that might directly fulfill the target specification or be used to guide the search.

Workflow. Fig. 1 presents Smir workflow. It takes as input an I/O specification $\mathcal{S}$ approximating the target expression, an objective function $f_{\mathcal{S}}$ based on $\mathcal{S}$, an inference grammar $\mathcal{G}$, and inference rules $\mathcal{R}^1, ..., \mathcal{R}^k$. For the sake of simplicity, we first describe Smir for one inference rule $\mathcal{R}^i$. The full process is described in Section 5.

(1) The synthesis loop starts from a candidate solution e (not matching the specification $\mathcal{S}$, otherwise e is directly returned);
(2) From e, a new candidate solution e' is generated through mutation, to check a new part of the search space;
(3) The inference rule $\mathcal{R}^i$ is then applied on e' (knowing $\mathcal{S}$), to retrieve a new candidate e'_i *a priori* closer to the I/O specification;
(4) The e'_i expression is sampled, and the objective function $f_{\mathcal{S}}$ is used to compare its outputs to the outputs of $\mathcal{S}$. If $f_{\mathcal{S}}(e'_i) = 0$ then e'_i matches the I/O specification and is returned; otherwise, the search continues with the most promising expression, w.r.t., $f_{\mathcal{S}}$, between e and e'.

Note that e'_i is not only used as a termination criterion. Even if all the I/O examples do not match, the e'_i objective function score is kept to guide the search. Finally, to deal with several inference rules $\mathcal{R}^1 \dots \mathcal{R}^k$, we compute all their images $e'_1, \dots, e'_k$ to check whether we can directly find a solution, and take the product of the corresponding objective functions to guide the search.

Example 1. *Consider the example from Listing 2 for the r output, with the* $\mathcal{R}^{\text{Affine}}$ *inference rule from Section 5.2, which aims to recover an affine relation over a MBA expression. We take as I/O*

specification $\mathcal{S} = \{(x = 5, y = 2) \rightarrow$ `0x7d7c08a5`*;* $(x = 20, y = 10) \rightarrow$ `0xa5ba8eaf`$\}$*. Using* Smir*, synthesis generates a first expression, e.g.,* $e_1 = x$*. As* e_1 *does not match* $\mathcal{S}$*, it is mutated randomly, leading, for example, to* $e_2 = x \times y$*. Here* Smir*'s inference rule* $\mathcal{R}^{\text{Affine}}$ *is applied on* e_2*, directly generating* `0x6aa7671b`$\times(x \times y)+$`0x52f20197`*. This is a clear shortcut compared to synthesizing the full expression through mutations.*

5 Search Modulo Inference Rules

We now present in details our Smir proposal. We then provide a set of inference rules that reveal useful to apply deobfuscation at scale on real binaries (cf. Section 6). Finally, we describe how we integrated Smir in the state-of-the-art black-box deobfuscator Xyntia, leading to the new prototype XSmir.

5.1 General framework

Algorithm 3 describes our framework to handle arbitrary inference rules with a local-search based synthesis algorithm. For clarity and precision purposes, Algorithm 3 is depicted as an extension of local search program synthesis (cf. Algorithm 2). Still, it fits any search framework.

Problem statement. Given an input and output space $\mathbb{I}$ and $\mathbb{O}$, we note E the set of expressions $e : \mathbb{I} \rightarrow \mathbb{O}$ in a user-given inference grammar $\mathcal{G}$. Typically, $\mathbb{I} = BV[m_1] \times ... \times BV[m_n]$ and $\mathbb{O} = BV[m]$, with $BV[s]$ the set of bit-vectors of size s. Given an I/O specification $\mathcal{S} \subset \mathbb{I} \times \mathbb{O}$, synthesis can theoretically generate any expression that validates $\mathcal{S}$ in E. Still, some expressions are much harder to synthesize than others in practice, typically because they are very large or contain hard-to-build parts. To synthesize such expressions, we extend synthesis, allowing the search to examine not only one expression at a time but a whole set of expressions linked to the current expression through adequate inference rules.

In the local-search framework, we also give ourselves an objective function $f_{\mathcal{S}} : E \rightarrow \mathbb{R}^+$, where $\mathbb{R}^+$ is the standard set of positive or zero real numbers. The objective function is used to guide the search by measuring how far a candidate expression is from $\mathcal{S}$. We only require that $f_{\mathcal{S}}(e) = 0$ iff $e \models \mathcal{S}$, i.e. if $\forall (i, o) \in \mathcal{S}, e(i) = o$. For instance, given a distance δ in $\mathbb{O}$, an appropriate objective function could be $f_{\mathcal{S}}(e) = \sum_{i,o \in \mathcal{S}} \delta(o, e(i))$.

Inference rules. We rely on a finite set of inference rules to extend synthesis. Formally speaking, an inference rule is a function $\mathcal{R} : E \times \mathcal{P}(\mathbb{I} \times \mathbb{O}) \rightarrow E$, with $\mathcal{P}(\mathbb{I} \times \mathbb{O})$ the power set of $\mathbb{I} \times \mathbb{O}$ i.e., the set of all possible I/O specifications. Hence, $\mathcal{R}$ takes as input a candidate expression and, given $\mathcal{S}$, returns a new expression better matching the I/O specification. The goal of an inference rule is twofold: (1) it empowers synthesis with a smarter termination criterion, which can return a solution from a "close enough" expression, instead of waiting to find the exact good match; (2) it guides the search, removing the need to explore the search space exhaustively.

When multiple inference rules $\mathcal{R}^1, \ldots, \mathcal{R}^k$ are provided, we assume that the reverser ranks them in order of preference (with the most preferred rules listed first). However, we do not elaborate on this ordering, as Section 6 demonstrates that it has no significant effect on synthesis outcomes. *From now on, we assume that the rules* $\mathcal{R}^1, ..., \mathcal{R}^k$ *are ordered according to their indices.*

Algorithm 3 Smir on Local search program synthesis

Input: The objective function $f_{\mathcal{S}}$ over the I/O specification $\mathcal{S}$; inference grammar $\mathcal{G}$; ordered inference rules $\mathcal{R}^1, \ldots, \mathcal{R}^k$;
Output: An expression matching the I/O specification $\mathcal{S}$

```
1:  e ← initial_expression(G)
2:  e_1, ..., e_k ← R^1(e, S), ..., R^k(e, S)
3:  N ← N_max
4:  while true do
5:    if ∏_{1≤i≤k} f_S(e_i) = 0 then
6:      pick i ∈ {1...k} the first index s.t., f_S(e_i) = 0
7:      return e_i
8:    else
9:      e' ← mutate(e)
10:     e'_1, ..., e'_k ← R^1(e', S), ..., R^k(e', S)
11:     if ∏_{1≤i≤k} f_S(e'_i) < ∏_{1≤i≤k} f_S(e_i) ∨ N ≤ 0 then
12:       e, e_1, ..., e_k ← e', e'_1, ..., e'_k
13:       N ← N_max
14:     else
15:       N ← N − 1
16:     end if
17:   end if
18: end while
```

Synthesis. Algorithm 3 depicts our synthesis approach. It takes as input the target I/O specification $\mathcal{S}$ with the objective function $f_{\mathcal{S}}$, the inference grammar $\mathcal{G}$, and a set of ordered inference rules $\mathcal{R}^1, ..., \mathcal{R}^k$. It first generates at line 1 an initial expression e from the inference grammar $\mathcal{G}$. Given this expression and the input inference rules, new candidate expressions are computed at line 2. Algorithm 3 then loops until it finds an expression $e \in \mathcal{G}$, and an inference rule $\mathcal{R} \in (\mathcal{R}^1, ..., \mathcal{R}^k)$ s.t., $\mathcal{R}(e, \mathcal{S}) \models \mathcal{S}$. To do so, it multiplies the objective function value of each expression generated by an inference rule (line 5). If it nullifies, there exists an $i \in \{1 \ldots k\}$ s.t., $\mathcal{R}^i(e, \mathcal{S}) \models \mathcal{S}$. We search for the first match based on the given preference order and return the final expression (lines 6 and 7). Otherwise, a new candidate expression e' is generated at line 9. For each inference rule, a new candidate expression is then produced at line 10. If the product of the objective functions for these new candidates decreases (line 11), the algorithm replaces e with e' and repeats the loop. Contrary to usual local search synthesis (Section 2.2), the objective function is never applied on e or e' but on the expressions produced by the inference rules. This guides the synthesis process modulo the inference rules as shown in Section 6.

5.2 Inference rules

Formally, an inference rule is a function that takes an expression e and an I/O specification $\mathcal{S}$ and returns a new expression e' such that $f_{\mathcal{S}}(e') \leq f_{\mathcal{S}}(e)$. This framework is very flexible and can be instantiated with diverse methods. Especially, we highlight two cases that produce useful rules:

- *Inverse operators.* Any (*possibly partial or one-to-many*) inversion procedure can be applied as an inference rule. Notably, abstract interpretation provides *backward operators* [62] for addition, multiplication, xor, and/or masks over bit-vectors. These inverse

procedures help handle top-level constant values for these operators. In addition, usual interpolation methods, e.g., for affine functions or polynomials, can also be leveraged as they are essentially a matrix inversion, enabling the inference of affine and polynomials over MBA expressions;

Example: For instance, the Addition rule $\mathcal{R}^{+}$ depicted in Table 4 takes an expression e and an I/O specification $\mathcal{S}$ and returns $e + c$ with c a constant value that helps decrease the objective function. To do so, for each I/O sample (i, o), it computes $o - e(i)$, leading to a non-empty set of candidate constant values $\mathcal{V}$. If there exists indeed a constant value $c^{\star}$ such that the target expression is equivalent to $e + c^{\star}$, then $\mathcal{V} = \{c^{\star}\}$ (as $+$ is invertible in $BV[m]$). We can then retrieve the constant value $c^{\star}$ and terminates the search. Otherwise, we pick the constant value that decreases the most the objective function through the argmin procedure.

The process is exactly the same for $\mathcal{R}^{\oplus}$ as xor is also invertible. If the operator is not always invertible, the rule must have a fallback result when no inverse can be computed (e.g., pick e). This is the case for $\mathcal{R}^{\times}$, Affine *and* Polynomial. *On the other hand, if the inverse operator is a one-to-many procedure, the rule picks the solution minimizing the objective function, as done by $\mathcal{R}^{\wedge\vee}$.*

- *Exhaustive search.* In some cases, it is also possible to find a new, more suitable, expression through an exhaustive search. To do so, the set to enumerate should stay small to keep the exhaustive search reasonably fast. This is the case for rotations and shifts with a constant value, which only require to check m possibilities, with m the bitvector size of e (typically 8, 16, 32 or 64).

  Example: For instance, the Left shift rule $\mathcal{R}^{\ll}$, enumerates all the possible shifts between 0 et $m - 1$ and keeps the one reducing the most the objective function through the argmin procedure. The search being exhaustive, if there is a constant value $c^{\star}$ such that the target expression is equivalent to $e \ll c$, then $\mathcal{R}^{\ll}$ will find it. This is the same process for the right shift and the rotation.

Rules description. We now detail how such *inverse operator* and *exhaustive search* cases can be instantiated. This leads to 9 inference rules, whose definitions are gathered in Table 4:

- As discussed above, the $\mathcal{R}^{+}$ and $\mathcal{R}^{\oplus}$ rules are straightforward because an inverse can always be found in $BV[m]$;
- In the $\mathcal{R}^{\times}$ case not all outputs are invertible in $BV[m]$. Still, modular inversion can be computed over any odd bit-vector. Thus, for each $(i, o) \in \mathcal{S}$, s.t., $e(i)$ is odd, $\mathcal{R}^{\times}$ computes a candidate $c = o \times e(i)^{-1}$. If no odd output exists, the rule returns e itself;
- In the $\mathcal{R}^{\wedge\vee}$ case, inversion can lead to multiple candidates. Still, all candidates share the same objective function value and one can be picked arbitrarily. In practice, $\mathcal{R}^{\wedge\vee}$ pick one such candidate by computing the two masks in a row in order to reduce the bit-to-bit difference between the outputs of e and $\mathcal{S}$;
- The *Affine inference rule* $\mathcal{R}^{\text{Affine}}$ infers expressions of the form: $c_1 \cdot e + c_2$ with e the current expression, and c_1, c_2 two coefficients. c_1 and c_2 are retrieved by solving a linear system. As for $\mathcal{R}^{\times}$, this rule requires to find two inputs on which e has opposite parities. If not found, it returns e itself;

Table 4: Inference rules

Name	Type*	Definition
Addition	I	$\mathcal{V} = \{o - e(i) \mid i, o \in \mathcal{S}\}$ $c = \underset{k \in \mathcal{V}}{\text{argmin}}\, f_{\mathcal{S}}(e + k)$ $\mathcal{R}^{+}(e, \mathcal{S}) = e + c$
Xor	I	$\mathcal{V} = \{o \oplus e(i) \mid i, o \in \mathcal{S}\}$ $c = \underset{k \in \mathcal{V}}{\text{argmin}}\, f_{\mathcal{S}}(e \oplus k)$ $\mathcal{R}^{\oplus}(e, \mathcal{S}) = e \oplus c$
Multiplication	I	$\mathcal{V} = \{o \times e(i)^{-1} \mid i, o \in \mathcal{S} \text{ s.t. } e(i) \text{ is odd}\}$ $c = \underset{k \in \mathcal{V}}{\text{argmin}}\, f_{\mathcal{S}}(e \times k)$ if $\mathcal{V} \neq \emptyset$ else 1 $\mathcal{R}^{\times}(e, \mathcal{S}) = e \times c$
AND-OR	I	$Z(\mathcal{S}) = \bigvee_{i,o \in \mathcal{S}} o \qquad O(\mathcal{S}) = \bigwedge_{i,o \in \mathcal{S}} o$ $\mathcal{S}_e = \{i, e(i) \mid i, o \in \mathcal{S}\}$ $c_{\wedge} = Z(\mathcal{S}) \vee \neg Z(\mathcal{S}_e)$ $c_{\vee} = O(\mathcal{S}) \wedge \neg O(\mathcal{S}_e)$ $\mathcal{R}^{\wedge\vee}(e, \mathcal{S}) = (e \wedge c_{\wedge}) \vee c_{\vee}$
Rotation *Log. left shift* *Log. right shift*	E	$c = \underset{k \in \{0 \ldots m-1\}}{\text{argmin}}\, f_{\mathcal{S}}(e \circ k)$ $\mathcal{R}^{\circ}(e, \mathcal{S}) = e \circ c$
Affine	I	$c_1 = (o_1 - o_2)(e(i_1) - e(i_2))^{-1}$ $c_2 = o_2 - c_1 \cdot e(i_2)$ $\mathcal{R}^{\text{Affine}}(e, \mathcal{S}) = c_1 \cdot e + c_2$
Polynomial	I	Lagrange interpolation on bit-vectors** $(c_0, \ldots, c_k)^T =$ $V^{-1}(e(i_0), \ldots, e(i_k)) \cdot (o_0, \ldots, o_k)^T$ $\mathcal{R}^{\text{Poly}}(e, \mathcal{S}) = \sum_{i=0}^{k} c_i \cdot e^i$

(*) I : Inverse ; E: Exhaustive search
(**) V^{-1} a pseudo-inverse of the Vandermonde matrix (see [10]); v^T is the transpose of v;

- The *Polynomial inference rule* $\mathcal{R}^{\text{Poly}}$ targets expressions of the form: $\sum_{i=0}^{k} c_i \cdot e^i$ with k a given integer and c_i some constant value. We adapt the Lagrange Interpolation for $BV[m]$ [10], solving a linear system of $k + 1$ equations, with constraints over the I/Os to ensure the existence of a solution. If the interpolation can not be done, it falls back to the affine rule;
- $\mathcal{R}^{\text{ror}}$, $\mathcal{R}^{\ll}$ and $\mathcal{R}^{\gg}$ exhaustively search for the best candidate through an internal minimization procedure (*argmin*). It is feasible as the set of candidates stays small.

Observe that the Addition, Xor, AND-OR masks, Shifts, and Rotations rules cannot miss the target expression if there is a possible match. This is explained by the fact that 1. $+$, $\oplus$, and $\wedge\vee$ are always invertible; 2. shifts and rotations perform an exhaustive search. Conversely, the Multiplication, Affine, and Polynomial rules are not always invertible. In such a case, they fall back to a default result, missing a possible match. In any case, for each rule, if it applies with no fallback, it returns the best match possible, i.e., the reachable expression that decreases the objective function score the most.

5.3 Implementation in XSmir

We implemented Smir in a prototype named XSmir, built on top of the black-box deobfuscator Xyntia[3] [39] from the Binsec framework [20]. We follow the optimal set-up advised for Xyntia:

[3] https://github.com/binsec/xyntia

- **Search algorithm.** We use Iterated Local Search (ILS) [37], directly following Algorithm 3;
- **Inference grammar.** We use the Expr inference grammar from Xyntia, which includes mixed boolean-arithmetic operators ($\neg$, $-_1$, $+$, $-$, $\times$, $\wedge$, $\vee$, $\oplus$) together with division;
- **Objective function.** We use the default Xyntia objective function: $f_{\mathcal{S}}(e) = \sum_{i,o \in \mathcal{S}} log_2(1 + |e(i) - o|)$
- **Sampling.** We consider the default sampling strategy, i.e., 5 fixed input vectors – setting all input variables to the same constant c among 0, 1, $2^m - 1$, $2^{m-1} - 1$, and -2^{m-1} – together with 95 randomly generated inputs with m the size of the variable.

6 Evaluation

Our evaluation answers the following Research Questions:

RQ1 *How does XSmir compare to state-of-the-art tools on real-world obfuscated binaries ?* We compare XSmir synthesis capabilities against the state-of-the-art *program synthesis* tools CVC4, CVC5 and DryadSynth, as well as black-box deobfuscators Syntia and Xyntia, on a real-world dataset of obfuscated binaries, evaluating their success and false positive rate;

RQ2 *How well does XSmir compress obfuscated expressions ?* We evaluate XSmir compression capabilities on the largest expressions extracted from obfuscated binaries and MBA expressions generated by the Tigress obfuscator [17];

RQ3 *Can XSmir handle the diversity of expressions in binaries ?* We compare XSmir against CVC4, CVC5, DryadSynth, Syntia, and Xyntia on non-obfuscated binaries to evaluate how they behave on a variety of expressions found in the wild;

RQ4 *How does XSmir compares to white-box deobfuscation?* We compare XSmir against the state-of-the-art white-box deobfuscator ProMBA and Gamba over MBA expressions;

RQ5 *How do internal parameters impact synthesis ?* We study whether the choices of rule ordering and objective function impact our results, and whether the distance based on inference rules is fundamental to effectively guide the search.

6.1 Methodology

We compare XSmir with the *program synthesizers* CVC4/5 [6, 9], and DryadSynth [22], the Mixed-Boolean Arithmetic (MBA) simplifiers ProMBA [33], and Gamba [49],[4] and the black-box deobfuscators Syntia [12] and Xyntia [39]. XSmir includes the nine inference rules of Table 4 and the parameters from Section 5.3.

Metrics. We use the following metrics: *success rate*, *false positive rate*, *precision*, *compression* and *synthesis time*. The *success rate* is the ratio of expressions equivalent to their ground truth over the whole dataset. Equivalence checking is performed with Bitwuzla [40], by comparing the expression extracted with Binsec [20] and the synthesized one. The *false positive rate* is the difference between the Synthesis rate and the Success rate, where the Synthesis rate is the ratio of expression found by the synthesizer, including non-equivalent results (Synthesis rate $\geq$ Success rate). Hence, the false positive rate indicates the percentage of cases in which an expression has been found to match the I/O specification $\mathcal{S}$ but the equivalence check failed. The *precision* is the ratio of equivalent cases over the number of expressions found by the synthesizer. *Compression* is the ratio of the sizes between the expression extracted from the binary code and our synthesized expression. The size of an expression is computed as the number of nodes in the expression abstract-syntax-tree (including operators, variables, and constant values). For instance, in the Snapchat example (Listing 1), XSmir recovers `tv_sec` $\times$ 1000, which has a size of 3. As the obfuscated expression has a size of 306, the compression rate is 102x. Over non-obfuscated code, a compression ≥ 1 indicates that synthesis returned the best result, i.e., as simple (or even simpler) as the original one. Finally, the *synthesis time* gives the average and median time spent on synthesis *for each succesfully recovered expression.*

Real obfuscated dataset (RQ1, RQ2). We consider a dataset of binaries obfuscated with state-of-the-art obfuscation. It comprises the APT-grade malware *X-Tunnel* studied in prior work [7] and highly protected with opaque predicates; *VMProtect* a code computing various MBA expressions obfuscated with VMProtect, a leading industrial obfuscator; *Loki*, a binary computing AES encryption, obfuscated with the recently proposed Loki obfuscator [53]; and the *Obfuscation-Dataset* by David et al.[5], the largest obfuscated dataset using real-world binaries, namely SQLite, FreeType, zlib and lz4, obfuscated with Tigress and OLLVM. We split this last dataset in two parts, *OLLVM* on which we apply the *EncodeArith* option from the eponymous obfuscator, and *Tigress* v4 on which we apply Tigress' *EncodeArith*, *EncodeLiteral* and *Virtualize* options (Table 6 summarizes the applied obfuscations).

To retrieve expressions, we divide each binary into *basic blocks* (simply called blocks), i.e., code units without jumps or calls, using Ghidra [41]. We filter out all duplicate blocks and randomly select up to 10,000 of the remaining blocks. We then extract all input-output relations from the blocks with symbolic execution using Binsec [20], ending up with an expression for each output detected. Finally, we filter out expressions with a size smaller than 3 (register and memory reads, constant values, and unary operators) to keep meaningful expressions. Statistics are provided in Table 5.

Real clean code dataset (RQ3). We also consider a dataset of non-obfuscated software. It includes code blocks from: *Core Utils*, i.e. system software available in any reputable Linux distribution; the *OpenSSL* libraries `libssl` and `libcrypto`; *Curl* for networking procedures; *FFmpeg* for image manipulation capabilities.

The rationale behind this benchmark and the focus on non-obfuscated binaries is twofold. First, it aims to collect as many diverse behaviors as possible from real-world code. Core Utils, Curl and OpenSSL represent a diversity of features that could appear in malware or protected software (networking, file handling, and cryptography), while FFmpeg adds complex (hence, worth protecting) processing functions, typically obfuscated in DRMs. Second, since usual obfuscation does not impact black-box deobfuscation [39], if a synthesizer recovers an expression from the dataset, it will also recover its obfuscated versions.

Expressions are generated in the same manner as for the obfuscated dataset. Table 5 reports statistics for the two datasets.

[4]We did not consider MBABlast and MBASolver as they are older than ProMBA and Gamba, MBABlast is very restrictive (can only handle expressions with up to 3 variables) and MBASolver is not publicly available.

[5]https://github.com/quarkslab/diffing_obfuscation_dataset

Table 5: Number of blocks for each real-world and obfuscated binary programs

	Core Utils	OpenSSL	Curl	FFmpeg	Total	VMProtect	X-Tunnel	Loki	OLLVM	Tigress	Total
# blocks	38,847	66,359	13,204	291,196	409,606	414	33,923	589	41,966	146,310	223,202
(selected)	(10,000)	(10,000)	(10,000)	(10,000)	(40,000)	(414)	(10,000)	(589)	(30,000)	(89,004)	(130,007)
Max size blocks	139	810	89	1,006	1006	47	1496	856	353	56,001	56001
Avg./median size blocks	7/5	6/4	5/4	9/5	7/5	10/7	212/22	8/5	6/4	11/6	11/6
Max size expressions	1,343	25,483	168,016	85,762	168,016	106	14,045	3,716,623	25,234	1,936,040	3,716,623
Avg./median size expressions	14/7	82/9	254/8	49/9	72/8	18/9	50/9	9,637/5	26/6	249/7	267/7

Table 6: Obfuscation applied to each binary in the *OLLVM* and *Tigress* datasets. Empty cells indicate binaries that were not available at the time of the experiments.

	OLLVM EncodeArith	Tigress EncodeArith	Tigress EncodeLiteral	Tigress Virtualize
FreeType	✓	✓	✓	
SQLite	✓	✓	✓	✓
lz4	✓	✓	✓	✓
zlib		✓	✓	✓

Synthetic dataset (RQ5). We consider the $B2_{COMB}$ synthetic dataset that extends the B2 dataset proposed by Menguy et al. [39] (which improved the dataset given by Blazytko et al. [12] with more diverse and hard-to-synthesize expressions). As B2, our $B2_{COMB}$ dataset contains 1110 mixed-boolean arithmetic expressions. They use between 2 and 6 inputs, manipulate the $-_1, \neg, +, -, \times, \wedge, \vee, \oplus$ operators. Each expression in $B2_{COMB}$ extend an expression from B2, matching one of the first 7 inference rule from Table 4 – we exclude the affine and polynomial rules as they lead to expressions too complex to be checked with Bitwuzla. For instance, $B2_{COMB}$ contains $((x \times y) \vee (y + z)) \times \texttt{0x5d544f40}$ as B2 includes $(x \times y) \vee (y + z)$. We took special care in balancing $B2_{COMB}$ among the rules.

Set up. We run all experiments on a server with an Intel Xeon 4214 with 48 cores running at 3.2GHz and 400GB of memory. All synthesis tasks are run in parallel on the 48 cores.

6.2 XSmir on obfuscated code (RQ1)

Table 7 compares the success, false positive, precision rate, the compression rates, and the synthesis time of CVC4/5, DryadSynth, Syntia, Xyntia, and XSmir on our real-world obfuscated dataset.

First, we observe that XSmir outperforms all the other tools both within a 1s and a 60s time budget, except on VMProtect, where Xyntia finds 1 ppt (points of percentage) more than XSmir on 1s, but then is 6 ppt behind for 60s. Indeed, withing 1s, CVC4/5 synthesizes about 28% of the dataset, DryadSynth 49%, Syntia synthesizes 2%, Xyntia 54% and XSmir 65%. In 60s, CVC4/5 find about 37%, DryadSynth 54%, Syntia 4% and Xyntia 63%, against 76% for XSmir. Hence, for both time budgets, XSmir achieves a gain of more than 64 ppt and 12.5 ppt compared to Syntia and Xyntia, as well as 36 ppt and 15 ppt for CVC4/5 and DryadSynth. Interestingly, XSmir also halves the false positive rate compared to Syntia and Xyntia in the 60s case and divides it by 4 for the 1s case.Hence, XSmir increases the trust in synthesis results.

Considering each use-case separately, we observe that VMProtect-obfuscated code has higher resilience to black-box deobfuscation tools, with a 54% success rate for Xyntia up to 61% for XSmir. It is conversely the binary with the highest success rate for CVC4/5, synthesizing 54% of expression against ≈35% for the other cases. Still, all expressions found by CVC4/5 and DryadSynth (but one) are also found by XSmir. For *Loki*, Xyntia already successfully synthesizes 84% of the expressions, leaving little room for improvement to XSmir, although an increase of 4ppt for XSmir is observable, which corresponds to some very large expressions with size ≥ 100 (cf. Section 6.3) that Xyntia cannot handle. More interestingly, we observe an increase of about 16ppt between Xyntia and XSmir on the *OLLVM* and *Tigress* obfuscated binaries.

Regarding the average synthesis time, XSmir takes overall 10% more time to find expressions than Xyntia. Still, XSmir synthesizes additional expressions. On the expressions recovered by both XSmir and Xyntia, XSmir is actually 10% faster. CVC4/5 is about twice as slow, and DryadSynth seems faster, but on the expressions recovered by both XSmir and DryadSynth, they have similar speed.

Notably, we observe that stochastic search-based Xyntia already outperformed the enumeration-based CVC4/5 and DryadSynth, and that XSmir deepens the gap.

Conclusion. On obfuscated code, XSmir performs noticeably better than prior black-box deobfuscators and synthesizers in terms of success rate (+13.5ppt on average, up to +16.1ppt) and false positives (divided by 2.4 on average compared to black-box deobfuscators).

6.3 XSmir compression capabilities (RQ2)

We now evaluate XSmir compression capabilities.

Compression on obfuscated dataset. As shown in Table 7, XSmir reaches on average a compression rate of 3.4 against 2.6 for Xyntia, 1.2 for DryadSynth and 2.7 for CVC4/5. Syntia does reach a higher compression rate of 4, but only because it synthesizes fewer expressions. Over the subset of expressions found by both XSmir and Syntia, XSmir achieves a compression rate of 5.3.

To better showcase the true compression capabilities of XSmir, we gathered very large expressions from our obfuscated dataset. Table 8 compares XSmir and Xyntia compression capabilities for expressions of size larger than 50 and 100. We observe that, on average, XSmir compresses expressions better than Xyntia. Indeed, for expressions larger than 50, XSmir achieves a compression of 1012x against 658x for Xyntia. For expressions larger than 100, XSmir yields a compression of 1776x compared to 1128x for Xyntia.

On *X-Tunnel*, we observe an average compression of 780x for expressions of size over 100 and up to 2600x in some cases. For *Tigress* the compression is 79x and 31x on average for expressions

Table 7: Comparison of XSmir on real-world obfuscated code blocks

		Timeout=1s						Timeout=60s					
		VMProtect	X-Tunnel	Loki	OLLVM	Tigress	**Total**	VMProtect	X-Tunnel	Loki	OLLVM	Tigress	**Total**
CVC4/5	Success rate	31.1%	22.6%	7.1%	26.6%	29.7%	28.4%	53.7%	36.7%	42.5%	32.0%	38.1%	37.2%
	Precision / FP	99.6/0.1%	95.8/1.0%	90.1/0.8%	99.0/0.3%	98.5/0.5%	98.3/0.5%	99.8/0.1%	95.6/1.7%	97.6/1.1%	97.9/0.7%	95.4/1.8%	95.8/1.6%
	Max compression	5x	880x	54880x	2x	71x	54880x	5x	880x	54880x	3x	71x	54880x
	Avg compression	1.2x	12.8x	715.6x	1.0x	1.2x	3.3x	0.9x	8.2x	120.3x	0.9x	1.1x	2.7x
	Mean time	0.17s	0.075s	0.084s	0.098s	0.097s	0.096s	3.857s	5.207s	1.376s	1.901s	2.302s	2.519s
	Median time	0.054s	0.039s	0.037s	0.055s	0.039s	0.044s	0.476s	0.118s	1.28s	0.06s	0.055s	0.058s
DryadSynth	Success rate	6.5%	5.6%	1.9%	48.6%	55.8%	49.4%	8.5%	7.2%	4.2%	52.5%	60.2%	53.5%
	Precision / FP	98.2/0.1%	88.2/0.8%	58.7/1.3%	95.9/2.1%	98.4/0.9%	97.9/1.1%	98.6/0.1%	87.9/1.0%	75.0/1.4%	94.4/3.1%	97.6/1.5%	97.0/1.7%
	Max compression	5x	880x	8622x	2x	71x	8622x	5x	880x	8622x	2x	71x	8622x
	Avg compression	1.3x	48.3x	728.1x	0.4x	0.6x	1.2x	1.0x	37.5x	346.4x	0.4x	0.5x	1.2x
	Mean time	0.04s	0.069s	0.08s	0.109s	0.086s	0.089s	5.74s	5.492s	18.732s	0.652s	0.657s	0.731s
	Median time	0.04s	0.047s	0.044s	0.048s	0.045s	0.045s	0.045s	0.055s	11.466s	0.05s	0.046s	0.047s
Syntia	Success rate	6.7%	7.3%	1.6%	0.7%	0.9%	1.5%	21.1%	15.8%	4.2%	2.5%	2.5%	3.8%
	Precision / FP	83.8/1.3%	78.5/2.0%	52.3/1.5%	11.1/5.8%	17.0/4.4%	25.7/4.4%	74.6/7.2%	75.1/5.2%	55.6/3.4%	18.5/11.1%	19.6/10.2%	28.0/9.8%
	Max compression	5x	265x	3695x	1x	8x	3695x	5x	265x	18293x	1x	8x	18293x
	Avg compression	0.7x	3.2x	184.3x	0.4x	0.6x	3.2x	0.6x	1.9x	375.9x	0.4x	0.6x	4.0x
	Mean time	0.368s	0.336s	0.301s	0.474s	0.44s	0.393s	6.499s	5.628s	7.134s	3.94s	2.972s	4.204s
	Median time	0.305s	0.256s	0.11s	0.452s	0.421s	0.348s	2.147s	1.141s	1.81s	1.946s	1.565s	1.462s
Xyntia	Success rate	53.5%	61.7%	82.0%	47.9%	53.4%	53.6%	53.9%	70.6%	84.2%	60.4%	62.2%	62.8%
	Precision / FP	95.6/2.5%	91.3/5.9%	93.1/6.1%	93.9/3.1%	92.4/4.4%	92.5/4.4%	95.2/2.7%	86.6/10.9%	92.6/6.7%	87.6/8.5%	87.3/9.1%	87.3/9.1%
	Max compression	5x	2640x	25868x	2x	286x	25868x	5x	2640x	25868x	3x	286x	25868x
	Avg compression	1.0x	11.7x	48.3x	0.8x	1.0x	2.6x	1.0x	10.3x	75.1x	0.7x	0.9x	2.6x
	Mean time	0.023s	0.139s	0.16s	0.126s	0.108s	0.114s	0.249s	1.149s	0.296s	1.757s	1.369s	1.389s
	Median time	0.009s	0.039s	0.09s	0.029s	0.021s	0.023s	0.009s	0.062s	0.096s	0.064s	0.036s	0.041s
XSmir	Success rate	52.1%	68.8%	84.9%	64.3%	64.6%	65.0%	60.7%	76.4%	88.8%	76.5%	76.2%	76.3%
	Precision / FP	99.1/0.5%	96.6/2.5%	97.7/2.0%	99.2/0.5%	98.5/1.0%	98.4/1.1%	98.3/1.1%	93.3/5.5%	96.6/3.1%	96.7/2.6%	95.2/3.8%	95.3/3.8%
	Max compression	5x	2640x	109761x	4x	286x	109761x	5x	2640x	109761x	5x	286x	109761x
	Avg compression	1.0x	12.3x	190.5x	1.0x	1.1x	3.9x	1.0x	11.1x	182.2x	0.9x	1.1x	3.4x
	Mean time	0.11s	0.037s	0.099s	0.096s	0.083s	0.081s	0.63s	1.692s	0.295s	1.551s	1.516s	1.525s
	Median time	0.023s	0.017s	0.022s	0.047s	0.047s	0.038s	0.029s	0.018s	0.024s	0.05s	0.05s	0.049s

Table 8: XSmir against Xyntia on big expressions from real-world obfuscated code blocks

		Size ≥ 50						Size ≥ 100					
		VMProtect	X-Tunnel	Loki	OLLVM	Tigress	**Total**	VMProtect	X-Tunnel	Loki	OLLVM	Tigress	**Total**
Xyntia	#Expressions simplified	0	267	16	1	46	330	0	168	13	0	6	187
	Avg. compression	-	477x	5534x	2x	26x	658x	-	726x	6797x	-	88x	1128x
XSmir	#Expressions simplified	0	287	23	0	61	371	0	179	20	0	8	207
	Avg. compression	-	508x	9896x	-	31x	1012x	-	780x	11368x	-	79x	1776x

Table 9: Results over Tigress MBA expressions with increasing complexity for 60s (5582 expressions for each level)

	Success	Precision / FP	Time (avg/median)	Compression Lvl-1	Lvl-2	Lvl-3
CVC4/5	81.0%	99.6% / 0.3%	0.91s / 0.004s	17×	28×	77×
Dryad.	30.1%	98.5% / 0.5%	2.28s / 0.05s	12×	20×	53×
Syntia	34.3%	75.8% / 23.4%	3.67s / 1.61s	10×	18×	42×
Xyntia	86.2%	96.3% / 3.1%	0.53s / 0.003s	17×	27×	74×
XSmir	90.5%	98.3% / 1.1%	1.8s / 0.06s	17×	29×	80×

The data are provided for three obfuscation levels ordered by their complexity.

larger than 100 and 50. Interestingly, on *Loki*, for expressions larger than 100, XSmir reaches a compression rate 1.7 times higher than Xyntia. Indeed, XSmir handles 7 more expressions than Xyntia where it divides their size by 11,368x. Some of the original expressions even reached a size over 250,000 that XSmir reduced by a factor 100,000x. Overall, XSmir deobfuscates 41 expressions larger than 50 and 20 expressions larger than 100 on which Xyntia fails.

MBA extracted from Tigress. Table 9 shows the results of CVC4/5, DryadSynth, Syntia, Xyntia and XSmir when focusing on the 5,582 mixed boolean-arithmetic expressions created by Tigress over the Freetype, Sqlite, and lz4 binaries (zlib is not included because Tigress MBA extraction failed on it). It considers three levels of obfuscation, ordered by their complexity. We observe that XSmir achieves a compression comparable to Xyntia (size is divided by 80 on average for level 3 expressions). Moreover, XSmir synthesizes more MBA expressions than Xyntia, with a smaller false positive rate. It ends up with XSmir drastically simplifying 366 highly obfuscated MBA expressions that were out-of-reach of Xyntia. In addition, XSmir compression is similar to CVC4/5 while it reaches a better success rate (90% vs. 80%), and 1.5x better than DryadSynth, which recovers only about 30% of expressions with a lower compression rate (about 54× for level 3). Interestingly, XSmir outperforms all the competitors with an acceptable synthesis time: successful synthesis take on average less that 2s (median: 0.06s).

Conclusion. XSmir can achieve significant reductions of large obfuscated code and retrieve their original size, while being able to succeed more often than competitors and with less false positive than Xyntia. Interestingly, on average, XSmir achieves better compression over obfuscated code than prior black-box approaches, getting closer to minimal recovery.

6.4 XSmir on non-obfuscated code (RQ3)

Table 11 depicts the results of XSmir, Xyntia, Syntia, CVC4/5, DryadSynth on our real clear dataset.

XSmir significantly outperforms Xyntia there, demonstrating its superior ability to recover a wide range of semantic behaviors, with gains up to ≈ 38ppt for *Core Utils* and *OpenSSL*, progressing from ≈ 52% to ≈ 90% success rate for a 60-second timeout. *Curl* and *FFMpeg* have a smaller but still marked gain of approximately 25 and 15 ppt. Syntia is largely behind Xyntia and XSmir. We observe that the gain is even stronger for a one-second timeout, with gains up to 47 ppt from Xyntia to XSmir, and an average of 35 ppt. The raw success rate is about 80%, very close to the one achieved within a 60-second timeout (84%).

Interestingly, CVC4/5 and DryadSynth showcase strikingly poor results here, averaging success about 18% for CVC4/5 and 5% for DryadSynth. Recall that this dataset encompasses a wide range of semantic behaviors, and that many of them require non-trivial constant values, which are hard to synthesize for CVC4/5 and DryadSynth.

Moreover, XSmir reaches a compression close to 1, which shows that the results returned by XSmir are very close to the perfect solutions – this is not the case for Xyntia and DryadSynth, which return expressions about twice the size. CVC4/5, however, present a similar compression rate.

Finally, the false positive rate is drastically reduced from Xyntia, from an average of 5% to about 1%, significantly strengthening the trust in XSmir as a black-box deobfuscator.

Conclusion. XSmir shows a significantly higher ability than prior black-box approaches to recover a wide range of semantic behaviors found in the wild (+29ppt on average). Moreover, it must be noted that on non-obfuscated code, XSmir achieves a compression close to 1x, demonstrating its superior ability to recover close-to-minimal expressions.

Table 10: Comparison between Xyntia, XSmir, ProMBA and Gamba on the Tigress MBA expressions (TO = 60s)

		Level-1	Level-2	Level-3
Xyntia	Success	84.1%	83.6%	83.2%
	Precision / FP	93.7% / 5.6%	94.0% / 5.3%	93.4% / 5.8%
	Avg. / Median Time	0.3s / 0.002s	0.3s / 0.002s	0.3s / 0.002s
	Compression	17×	28×	76×
XSmir	Success	**89.4%**	**88.9%**	**88.9%**
	Precision / FP	98.9% / 1.3%	98.9% / 1.4%	98.9% / 1.1%
	Avg. / Median Time	1.8s / 0.06s	1.8s / 0.06s	1.8s / 0.06s
	Compression	17×	29×	79×
ProMBA	Success	85.4%	77.4%	44.8%
	Precision / FP	100% / 0%	100% / 0%	100% / 0%
	Avg. / Median Time	18s / 7s	26s / 24s	37s / 38s
	Compression	15×	23×	56×
Gamba	Success	76.7%	70.6%	49.5%
	Precision / FP	100% / 0%	100% / 0%	100% / 0%
	Avg. / Median Time	1.3s / 0.6s	3s / 1s	10s / 5s
	Compression	16×	24×	57×

The data are provided for three obfuscation levels ordered by their complexity. The analysis is considered a success if the retrieved expression is less than five times the size of the ground truth.

6.5 XSmir vs White-box Deobfuscation (RQ4)

We compare XSmir against the state-of-the-art white-box deobfuscators ProMBA and Gamba over the MBA extracted with Tigress v4 and an obfuscated version of $B2_{COMB}$. ProMBA and Gamba always return a result, even if they did not fully simplified the expression. Thus, in this part, we consider that the deobfuscation is a success if the simplified expression is at most 5 times bigger than the target non-obfuscated expression.

Table 10 shows the results of XSmir against ProMBA and Gamba on MBA expressions extracted with Tigress for 3 levels of obfuscation. We observe that XSmir reaches the best success, recovering 89% of the expressions independently of the level of obfuscation applied. On the other hand, ProMBA and Gamba are impacted by the level of obfuscation. Hence, they respectively reach a success of 85.4% and 76.7% over the simplest MBA expressions (level 1), which is close to XSmir. However, with more complex obfuscation (level 2 and 3), we observe that they recovers a lot fewer expressions than XSmir: respectively 77.4% and 70.6% for the second level of obfuscation and only 44.8% and 49.5% for the third level of obfuscation.

Table 12 shows the results of XSmir against ProMBA and Gamba on the $B2_{COMB}$ dataset obfuscated with Tigress. Here also XSmir highly outperforms them, recovering 44.9% of the expressions against 3.0% for Xyntia, 5.2% for ProMBA and 22% for Gamba. The compression of XSmir is also a lot higher, dividing expression sizes by 219 on average against 90 for ProMBA and 118 for Gamba.

The origin of XSmir good results over white-box deobfuscation is two-fold. First, the more obfuscated the expression is, the more time white-box methods need for simplification. A 1-minute time budget is already not enough for ProMBA and Gamba on the first level of obfuscation. Conversely, XSmir is not impacted by the obfuscation and synthesis often takes less than 1s. Second, ProMBA and Gamba breaks down the expressions into an algebraic basis, which may not be the simplest representation.

Conclusion. XSmir outperforms the MBA deobfuscators ProMBA and Gamba over all MBA obfuscations. Over highly complex MBA expressions, XSmir succeed almost 2 times more often than white-box deobfuscators. This is an unexpected outcome as ProMBA and Gamba are dedicated to handling MBA simplification while XSmir is general and can be applied to many different obfuscations.

6.6 XSmir Internal Evaluation (RQ5)

We evaluate the impact of internal parameters on synthesis, namely, the effect of inference-based guidance and the impacts of the objective function and of the inference rule order. To highlight subtle behaviors of XSmir, we rely on the $B2_{COMB}$ synthetic dataset.

Cost of Smir. We first evaluate the impact of Smir on the search. Table 13 compares Xyntia and XSmir on $B2_{COMB}$ with a one-hour delay. We observe that Xyntia can only synthesize 3.4% of $B2_{COMB}$ while XSmir reaches a 47.7% success rate, when the *program synthesizers* fail totally — success equals 0.9% for CVC4/5 and DryadSynth. This shows that Smir highly improves efficiency on the hard problems provided in $B2_{COMB}$. Interestingly, XSmir reaches such a good success rate even though the number of explored solutions per second is 120× smaller for XSmir than for Xyntia (300 vs. 38,000

Table 11: Comparison of XSmir on real-world code blocks

		Timeout=1s					Timeout=60s				
		Core Utils	OpenSSL	Curl	FFmpeg	**Total**	Core Utils	OpenSSL	Curl	FFmpeg	**Total**
CVC4/5	Success rate	12.0%	11.0%	11.4%	19.9%	13.8%	16.8%	14.2%	14.7%	25.2%	18.1%
	Precision / FP	95.0/0.6%	95.6/0.5%	91.5/1.1%	94.3/1.2%	94.3/0.8%	92.9/1.3%	94.4/0.8%	89.9/1.7%	92.4/2.1%	92.5/1.5%
	Max compression	20x	49x	62x	7x	62x	20x	49x	62x	7x	62x
	Avg compression	1.1x	1.2x	1.0x	1.0x	1.1x	0.9x	1.1x	0.9x	0.9x	1.0x
	Mean time	0.078s	0.069s	0.071s	0.085s	0.078s	2.124s	2.402s	2.616s	2.127s	2.256s
	Median time	0.051s	0.051s	0.05s	0.052s	0.051s	0.055s	0.054s	0.052s	0.054s	0.054s
DryadSynth	Success rate	3.6%	4.3%	3.5%	2.7%	3.5%	4.9%	5.5%	5.5%	4.3%	5.0%
	Precision / FP	94.0/0.2%	91.9/0.4%	86.7/0.5%	78.7/0.7%	88.3/0.5%	92.5/0.4%	92.1/0.5%	86.8/0.8%	75.2/1.4%	86.4/0.8%
	Max compression	20x	49x	62x	7x	62x	20x	49x	62x	7x	62x
	Avg compression	1.0x	1.3x	0.8x	0.7x	1.0x	0.7x	1.0x	0.5x	0.5x	0.7x
	Mean time	0.094s	0.067s	0.095s	0.123s	0.092s	4.271s	5.975s	8.045s	8.029s	6.446s
	Median time	0.046s	0.044s	0.061s	0.065s	0.05s	0.057s	0.05s	0.087s	0.134s	0.069s
Syntia	Success rate	2.5%	2.2%	2.1%	4.1%	2.8%	8.8%	6.9%	7.5%	13.3%	9.3%
	Precision / FP	73.9/0.9%	65.7/1.2%	67.3/1.0%	67.3/2.0%	68.5/1.3%	67.0/4.3%	62.7/4.1%	64.9/4.0%	61.7/8.3%	63.7/5.3%
	Max compression	3x	12x	8x	2x	12x	3x	12x	8x	2x	12x
	Avg compression	0.5x	0.5x	0.5x	0.5x	0.5x	0.5x	0.5x	0.5x	0.4x	0.5x
	Mean time	0.454s	0.43s	0.436s	0.428s	0.436s	5.949s	4.392s	5.16s	5.451s	5.346s
	Median time	0.433s	0.405s	0.4s	0.404s	0.411s	2.231s	1.846s	2.059s	2.016s	2.053s
Xyntia	Success rate	39.5%	46.7%	51.4%	43.4%	44.7%	52.0%	55.0%	64.7%	51.8%	55.2%
	Precision / FP	92.7/3.1%	94.9/2.5%	94.6/2.9%	92.2/3.7%	93.5/3.1%	88.5/6.7%	93.2/4.0%	92.0/5.6%	86.4/8.1%	89.8/6.2%
	Max compression	20x	198x	11x	23x	198x	20x	198x	11x	23x	198x
	Avg compression	0.7x	0.7x	0.7x	0.8x	0.7x	0.6x	0.6x	0.6x	0.7x	0.6x
	Mean time	0.209s	0.228s	0.228s	0.147s	0.201s	3.482s	1.199s	1.934s	1.435s	2.037s
	Median time	0.12s	0.153s	0.152s	0.027s	0.111s	0.233s	0.209s	0.232s	0.072s	0.189s
XSmir	Success rate	86.3%	88.9%	87.9%	61.6%	80.4%	89.8%	91.6%	90.2%	67.6%	84.1%
	Precision / FP	99.3/0.6%	99.2/0.7%	98.5/1.4%	98.1/1.2%	98.8/1.0%	98.0/1.9%	98.4/1.5%	97.1/2.7%	95.4/3.3%	97.3/2.3%
	Max compression	20x	198x	60x	27x	198x	20x	198x	60x	27x	198x
	Avg compression	1.0x	1.1x	1.0x	1.0x	1.0x	1.0x	1.1x	1.0x	1.0x	1.0x
	Mean time	0.036s	0.036s	0.032s	0.059s	0.04s	0.437s	0.267s	0.244s	1.1s	0.499s
	Median time	0.017s	0.017s	0.017s	0.018s	0.018s	0.018s	0.018s	0.017s	0.018s	0.018s

Table 12: Xyntia, XSmir, ProMBA and Gamba on $B2_{COMB}$ obfuscated by Tigress (MBA obfuscation level 3; TO=1h)

	Success	FP	Precision	Avg./Median Time	Compr.
Xyntia	3.0%	1.3%	78.0%	24s / 1s	266×
XSmir	44.9%	3.2%	93.7%	365s / 143s	219×
ProMBA	5.2%	0%	100%	194s / 78s	90×
GAMBA	22%	0%	100%	256s / 60s	118×

We count a success if the result is less than five times the size of the ground truth.

Table 13: Evaluation on synthetic datasets (TO=1h)

		$B2_{COMB}$
Xyntia	Success	3.4%
	Precision / False positive	72.0% / 1.3%
	Avg. compression	2.6x
	Avg. / Median time	42.3s / 1.7s
XSmir	Success	47.7%
	Precision / False Positive	92.8% / 3.2%
	Avg. compression	1.4x
	Avg. / Median time	332s / 29.9s
XSmir $^-$	Success	27.8%
	Precision / False positive	95.5% / 1.3%
	Avg. compression	1.6x
	Avg. / Median time	343s / 35.1s

candidate solutions per second). This highlights that good guidance introduced with Smir counterbalances rules computing costs. Nevertheless, adding too many inference rules would certainly impact synthesis at some point, as adding too many rewriting rules may hinder a solver. The evaluation shows that our set of inference rules offers a good balance between expressiveness and speed. Still, finding methods to limit rule overhead is a promising direction.

Ablation Study – Guidance Impact. Table 13 evaluates the impact of our extended guidance on XSmir. To do so, we consider a new version of XSmir, noted XSmir $^-$ where the inference rules are only applied for the termination condition i.e., at line 5 of Algorithm 3 to check if there is an inference rule $\mathcal{R}$ s.t., $f_{\mathcal{S}}(\mathcal{R}(e, \mathcal{S})) = 0$. If so, it returns $\mathcal{R}(e, \mathcal{S})$. Otherwise, it computes the usual Xyntia objective function, i.e., line 11 computes $f_{\mathcal{S}}(e')$. Hence, unlike XSmir, the guidance is not performed modulo inference rules.

XSmir$^-$ synthesizes only 27.8% of $B2_{COMB}$, compared to 47.7% for XSmir. Overall, these results show that guidance modulo inference rules is fundamental to make the most of Smir and XSmir.

Parameters Analysis. We now examine the impact of the *objective function* choice and the *inference rules order*.

- The guidance depends on the objective function $f_{\mathcal{S}}$. We checked that XSmir always outperforms Xyntia independently of the following typical choices of $f_{\mathcal{S}}$: *arithmetic*, *logarithmetic*, *hamming* and *xor* (see Table 14) on $B2_{COMB}$ within 1h. Table 15 shows

Table 14: Definition of the objective functions

Arithmetic	$f_{\mathcal{S}}(e) = \sum_{i,o \in \mathcal{S}} \lvert e(i) - o \rvert$
Logarithmetic	$f_{\mathcal{S}}(e) = \sum_{i,o \in \mathcal{S}} \texttt{log2}(1 + \lvert e(i) - o \rvert)$
Hamming	$f_{\mathcal{S}}(e) = \sum_{i,o \in \mathcal{S}} \texttt{bitcount}(e(i) \oplus o)$
XOR	$f_{\mathcal{S}}(e) = \sum_{i,o \in \mathcal{S}} \lvert e(i) \oplus o \rvert$

Table 15: Impact of the objective function ($B2_{COMB}$, TO=1h)

		Arith	**Logarith**	Hamm	XOR
Xyntia	Succ.	1.9%	3.4%	3.1%	1.9%
	Precision / FP	60.6% / 1.2%	72.0% / 1.3%	67.3% / 1.5%	58.8% / 1.3%
	Avg. compr.	3.5x	2.6x	2.6x	3.6x
	Avg. / Median time	187s / 2.2s	42.3s / 1.7s	107s / 1.6s	140s / 0.6s
XSmir	Succ.	40.7%	47.7%	47.1%	38.2%
	Precision / FP	93.7% / 2.7%	92.8% / 3.7%	91.7% / 4.2%	94.2% / 2.4%
	Avg. compr.	1.4x	1.4x	1.3x	1.5x
	Avg. / Median time	328s / 17.1s	332s / 29.9s	315s / 44.9s	257s / 18.3s

Table 16: Impact of the operators' order ($B2_{COMB}$, TO=1h)

Orders	Succ.	Prec.	FP	Avg. Compr.	Avg. Time	Median Time
1 (default)	47.7%	92.8%	3.7%	1.4x	332s	29.9s
2	47.1%	90.4%	5.0%	1.4x	362s	26.3s
3	45.2%	88.2%	6.1%	1.4x	324s	34.1s
4	44.9%	89.4%	5.3%	1.4x	295s	28.6s
5	47.3%	88.9%	5.9%	1.4x	341s	29.1s
6	48.7%	91.5%	4.5%	1.4x	354s	32.6s

Orders Associations	
1	+, ×, ⊕, ∧∨, ror, shiftl , ushiftr , affine , poly
2	affine , shiftl , ∧∨, ushiftr , ror, +, ×, poly, ⊕
3	shiftl , ∧∨, +, ushiftr , ⊕, affine , poly, ×, ror
4	poly, ushiftr , ⊕, affine , shiftl , +, ×, ror, ∧∨
5	×, ror, ⊕, ∧∨, +, ushiftr , shiftl , affine , poly
6	⊕, shiftl , +, ×, ∧∨, ushiftr , ror, affine , poly

Table 17: XSmir with the product against the min inference rule combinator (TO=1h)

		$B2_{COMB}$
XSmir	Success	47.7%
	Precision / False Positive	92.8% / 3.2%
	Avg. compression	1.4x
	Avg. time / Median time	332s / 29.9s
XSmir-min	Success	56.6%
	Precision / False Positive	90.4% / 7.5%
	Avg. compression	1.27
	Avg. time / Median time	155s / 24s

that, as expected, XSmir always highly outperforms Xyntia (≈+36ppt) with all tested objective functions;

- We study the impact of the inference rule order (Table 16). Over 6 inference rules order from Table 4, we observe no significant result variation on $B2_{COMB}$ within 1h. Both success and false positive rates only vary of 4 ppt among the permutations;
- XSmir guides the search w.r.t. multiple inference rules by computing the *product* of the objective function results (cf. line 5 and 10 in Algorithm 3). Table 17 shows the results on $B2_{COMB}$ when taking instead the *minimum* objective function result. It yields a better success rate (56.6% against 47.7%) and halves the average success time (155s against 332s), yet doubles the false positive rate, which we consider bad. Overall, there is still room for improvement, and better combinators is a promising direction.

Conclusion. Internal evaluation demonstrates that the novel guidance mechanism in XSmir is indeed important; that the ordering of inference rules and the choice of the objective function do not alter our main findings; that while the Smir mechanism has some significant cost in term of search speed, it is positively balanced by higher inference abilities; and finally that the product inference rules combinator is a good balance between success and false positives.

7 Discussions

Smir enables the application of black-box deobfuscation in a more general manner than previous works, that were mostly applied to *virtual machine handlers*. We discuss a few limits and possible extensions hereafter.

Limits. Smir empowers the reverser with more control over the synthesis process by crafting a set of dedicated rules that can help overcome typical weaknesses of black-box synthesis as well as specific hard patterns that could be found in obfuscated codes.

Still, Smir inherits the usual black-box deobfuscation limitations, which are orthogonal to the work presented here. First, its expressiveness is restricted by the inference grammar $\mathcal{G}$. While there is no a priori limit as long as the operands in the grammar can be executed against sample inputs, in practice this line of work considers rather simple expression languages, not full fledge program grammars (ex: no loop nor conditonal expression). Second, the method crucially relies on the selected *reverse window*. In Section 6, these are automatically extracted for the sake of systematic evaluation. However, black-box deobfuscation is not restricted to such reverse windows, as it can in principle cope with any single-entry single-exit code subpart [39] – possibly including conditionals, loops or even multi-threading – as long as the input/output are clearly identified and the underlying behaviour remains semantically simple.[6] Automated detection of such complex *reverse windows* is an interesting research direction. Finally, black-box inference may also, in some cases, crucially depend on the I/O sampling strategy, for example, for point functions [39].

Synthesis may also fail. Common failure cases of XSmir include the need for *nested* constant values and large expressions. Also, an interesting source of false positive is the division operator, that can yield expressions incorrect at a single point. For instance, XSmir sometimes generates x/x instead of 1, which differs only on x=0. Detecting such patterns would reduce false positives even more.

[6] It is even possible in principle to synthesize code with (bounded) loops or conditions as all executable operations can be added to the XSmir grammar. Yet, it would drastically increase the search space size. Especially, loops are beyond SOTA synthesizers capabilities. For conditionals, the major challenge is to correctly sample the input to activate all branches. Using ideas from CEGIS (counter-example guided synthesis) seems then mandatory, hence leaving the black-box setting.

Genericity of Smir and XSmir. Our proposal is very generic and can be applied and customized in many ways. Adding inference rules can be easily done for any search-based synthesis method, be it enumerative [2, 15] or stochastic [2, 28]. Also, while we presented the approach in a black-box scenario, it can also be used in a greybox deobfuscation scenario [21], where synthesis is recursively attempted on each sub-expression of the obfuscated expression. Finally, customization to specific examples can be achieved either by extending the inference grammar, or by adding new inference rules, giving flexibility to an advanced reverser.

Rules rationale. We designed rules for standard bitvector operators and generalized them to affine, masks and polynomial relations. Following the principles from Section 5.2, we either check if some inverse operation could be applied or if full enumeration is possible. Finding new rules could follow the same process. We believe it is worth the effort: as obfuscation does not change the semantics of the underlying code, new inference rules will easily transfer between use cases and obfuscation techniques.

8 Related work

Black-box deobfuscation. [12, 39] has already been extensively discussed. Smir extends synthesis to handle usual hard-to-synthesize expressions, making it possible to apply black-box deobfuscation beyond the usual virtualization scenario.

White- and grey-box deobfuscation. Recent advances leverage static program analysis, especially symbolic methods, to simplify obfuscated programs [3, 7, 14, 32, 51, 54, 60]. These are white-box and usually show strong guarantees. Conversely, they are inherently impacted by syntactic complexity, impeding their use on heavily obfuscated binaries. More importantly, efficient countermeasures have been proposed [44, 45, 55].

David et al. [21] proposed QSynth, a grey-box approach where black-box synthesis is applied recursively to the expression under analysis and its sub-expressions. Our method can be used to replace the cache-based black-box recovery method from QSynth.

Program synthesis. [2, 28] aims to infer a program based on a user-given specification. The specification can take various forms, e.g., a logical formula, a trace, or a set of input/output relations, a.k.a Programming by Example (PBE) [28]. In this work, we propose the Smir PBE framework that uses inference rules i) to elevate a candidate solution to a final one; and ii) to guide the synthesis.

Other works use automated reasoning to complement candidate solutions. Cegis(τ) [1] attempts to find non-trivial constant values by generating incomplete programs and using an SMT solver to fill in the gaps with appropriate constant values. It is, however, computationally heavy [1] and it requires a ground-truth (e.g., an expression) to perform counter-example guided synthesis. Thus, it is not suited for black-box deobfuscation. DryadSynth [22] combines previous candidate solutions using *bottom-up deduction rules*. It only enables the combination of already seen candidate expressions, and cannot create new sub-terms (like constant values). Moreover, these bottom-up deduction rules do not guide the synthesis. This is thus very different from our mechanism, and possibly complementary.

Regarding guidance, different approaches [8, 35, 61] use machine learning to bias the search toward likely solutions. EuPhony [35] uses a custom probabilistic model to enumerate expressions more likely to be a solution — according to the training. However, Barke et al. [8] showed that it is brittle outside its own benchmarks and proposed Probe [8] to learn likely programs *just-in-time*, avoiding training datasets overfitting. Li et al. [61] use Large Language Models for guidance. These approaches could benefit from Smir and conversely. Neo [24] implements a CDCL-like algorithm to learn from past mistakes and guide enumerative synthesis with additional constraints. However, it is not applicable to black-box deobfuscation since it checks against a *ground-truth*.

Finally, *deductive synthesis* [30, 38, 48] is not to be confused with our contribution. *Deductive synthesis* aims at generating programs starting from the specifications and using *rewrite rules* to deduce a program, in a *top-down* fashion. Similarly, some work do *rule inference* [42, 43], trying to learn new *rewrite rules* from a corpus of programs to improve inference of whole programs from specification, while we exploit *inference rules*, that deduce a final solution from a candidate one, bridging the gap between *bottom-up* and *top-down* synthesis.

9 Conclusion

Black-box deobfuscation is a promising research area, yet limited by current synthesis capabilities. Applying it at scale beyond the virtualization case is still an open problem, as it requires to recover a wide range of semantically distinct behaviors, including for example arbitrary constant values, linear or polynomial expressions, etc. With a view to expand the scope of black-box deobfuscation from virtualization to generic programs, we propose Search Modulo Inference Rules (Smir), the first synthesis framework combining search with inference rules in order to recover usually hard-to-synthesize expressions. We implement the approach in XSmir, a novel black-box deobfuscator able to fully recover most basic blocks from real-world binaries, be they obfuscated or not, demonstrating superior performance compared to state of the art in terms of higher success rate and lower false positive rates, and achieving potentially high compression on heavily obfuscated code.

Ethical Considerations

Practical deobfuscation tools and techniques helps in malware analysis and to assess the strength of current obfuscators used for legitimate Intellectual Properties protection, and are thus important for security. On the other hand, deobfuscation tools could be used to help malicious reversers stealing intellectual property included in software. Such a 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.

Acknowledgments

This work has benefited from a government grant managed by the National Research Agency under France 2030 with reference "ANR-22-PECY-0007"; from the BPI under Plan France 2030 with reference DOS0233319/00; and from the European Union's Horizon Europe research and innovation programme ENSEMBLE under grant agreement No 101168360.

References

[1] Alessandro Abate, Haniel Barbosa, and Clark Barrett et al. 2023. Synthesising Programs with Non-trivial Constants. *J. of Automated Reasoning* (2023).

[2] Rajeev Alur, Rastislav Bodík, and Garvit et al. Juniwal. 2013. Syntax-guided synthesis. In *Formal Methods in Computer-Aided Design, FMCAD*. IEEE.

[3] Sebastian Banescu, Christian Collberg, and Vijay et al. Ganesh. 2016. Code obfuscation against symbolic execution attacks. In *Annual Conference on Computer Security Applications, ACSAC*.

[4] Boaz Barak. 2016. Hopes, fears, and software obfuscation. *Commun. ACM* (2016).

[5] Boaz Barak, Oded Goldreich, and Impagliazzo et al. 2012. On the (im) possibility of obfuscating programs. *Journal of the ACM (JACM)* (2012).

[6] 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.

[7] 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.

[8] Shraddha Barke, Hila Peleg, and Nadia Polikarpova. 2020. Just-in-time learning for bottom-up enumerative synthesis. *Proc. ACM Program. Lang.* (2020). Issue OOPSLA.

[9] Clark Barrett, Christopher L. Conway, and Morgan Deters et al. 2011. CVC4. In *Conference on Computer Aided Verification (CAV '11)*. Springer.

[10] Lucas Barthelemy, Ninon Eyrolles, and Guénaël et al. Renault. 2016. Binary permutation polynomial inversion and application to obfuscation techniques. In *Proceedings of the 2016 ACM Workshop on Software PROtection*. 51–59.

[11] Tim Blazytko. [n. d.]. Binary Ninja plugin to identify obfuscated code and other interesting code constructs. https://github.com/mrphrazer/obfuscation_detection.

[12] Tim Blazytko, Moritz Contag, Cornelius Aschermann, and Thorsten Holz. 2017. Syntia: Synthesizing the Semantics of Obfuscated Code. In *Usenix Security*.

[13] Cameron Browne, Edward Powley, and Daniel et al. Whitehouse. 2012. A survey of monte carlo tree search methods. *IEEE Transactions on Computational Intelligence and AI in games* (2012).

[14] 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.

[15] 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).

[16] Roxane Cohen, Robin David, and Florian Yger et al. 2025. Identifying Obfuscated Code through Graph-Based Semantic Analysis of Binary Code. *CoRR* (2025).

[17] C. Collberg, S. Martin, J. Myers, and B. Zimmerman. [n. d.]. The Tigress C Diversifier/Obfuscator. http://tigress.cs.arizona.edu/

[18] Christian Collberg and Jasvir Nagra. 2009. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*. Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection.

[19] Christian Collberg, Clark Thomborson, and Douglas Low. 1997. A taxonomy of obfuscating transformations.

[20] 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.

[21] Robin David, Luigi Coniglio, and Mariano Ceccato. 2020. QSynth-A Program Synthesis based Approach for Binary Code Deobfuscation. In *BAR 2020 Workshop*. Internet Society.

[22] Yuantian Ding and Xiaokang Qiu. [n. d.]. Enhanced Enumeration Techniques for Syntax-Guided Synthesis of Bit-Vector Manipulations. ([n. d.]).

[23] Chris Eagle. 2011. *The IDA pro book*. no starch press.

[24] Yu Feng, Ruben Martins, and Osbert et al. Bastani. 2018. Program synthesis using conflict-driven learning. In *PLDI'18*. ACM.

[25] UNH SoftSec Group. 2021. *MBA-Solver Code and Dataset. https://github.com/softsec-unh/MBA-Solver*.

[26] Sumit Gulwani. 2011. Automating string processing in spreadsheets using input-output examples. In *POPl'11*. ACM.

[27] Sumit Gulwani. 2016. Programming by examples. *Dependable Software Systems Engineering* 45, 137 (2016), 3–15.

[28] Sumit Gulwani, Oleksandr Polozov, Rishabh Singh, et al. 2017. Program synthesis. *Foundations and Trends® in Programming Languages* (2017).

[29] hot3eed. 2024. Reverse Engineering Snapchat (Part I): Obfuscation Techniques. https://hot3eed.github.io/snap_part1_obfuscations.html.

[30] Shachar Itzhaky, Hila Peleg, and Nadia et al. Polikarpova. 2021. Cyclic program synthesis. In *POPl*. ACM.

[31] Pascal Junod, Julien Rinaldini, and Johan Wehrli et al. 2015. Obfuscator-LLVM – Software Protection for the Masses. In *Workshop on Software Protection*. IEEE.

[32] Johannes Kinder. 2012. Towards Static Analysis of Virtualization-Obfuscated Binaries. In *Working Conference on Reverse Engineering, WCRE*.

[33] Jaehyung Lee and Woosuk Lee. 2023. Simplifying Mixed Boolean-Arithmetic Obfuscation by Program Synthesis and Term Rewriting. In *Conference on Computer and Communications Security*.

[34] Jaehyung Lee and Woosuk Lee. 2023. Simplifying Mixed Boolean-Arithmetic Obfuscation by Program Synthesis and Term Rewriting. In *Conference on Computer and Communications Security, CCS*. ACM.

[35] Woosuk Lee, Kihong Heo, and Rajeev Alur et al. 2018. Accelerating search-based program synthesis using learned probabilistic models. In *PLDI*. ACM.

[36] Binbin Liu, Junfu Shen, and Jiang Ming et al. 2021. MBA-Blast: Unveiling and Simplifying Mixed Boolean-Arithmetic Obfuscation. In *USENIX Security*.

[37] Helena Ramalhinho Lourenço, Olivier C Martin, and Thomas Stützle. 2019. Iterated local search: Framework and applications. In *Handbook of metaheuristics*.

[38] Zohar Manna and Richard Waldinger. 1980. A Deductive Approach to Program Synthesis. *ACM Transactions on Programming Languages and Systems* (1980).

[39] Grégoire Menguy, Sébastien Bardin, and Bonichon et al. 2021. Search-Based Local Black-Box Deobfuscation: Understand, Improve and Mitigate. In *Conference on Computer and Communications Security*.

[40] Aina Niemetz and Mathias Preiner. 2023. Bitwuzla. In *Computer Aided Verification*. Springer.

[41] National Security Agency (NSA). [n. d.]. Ghidra. https://ghidra-sre.org/

[42] Maxwell Nye, Armando Solar-Lezama, and Joshua et al. Tenenbaum. 2020. Learning compositional rules via neural program synthesis. In *Conference on Neural Information Processing Systems*.

[43] Andres Nötzli, Andrew Reynolds, and Haniel et al. Barbosa. 2019. Syntax-Guided Rewrite Rule Enumeration for SMT Solvers. In *Theory and Applications of Satisfiability Testing*. Springer.

[44] 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*.

[45] 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*.

[46] Oreans Technologies. 2020. Themida – Advanced Windows Software Protection System. http://oreans.com/themida.php.

[47] Alex Petrov. 2023. Hands-Free Binary Deobfuscation with gooMBA. https://hex-rays.com/blog/deobfuscation-with-goomba.

[48] Nadia Polikarpova and Ilya Sergey. 2019. Structuring the synthesis of heap-manipulating programs. *POPL* (2019).

[49] Benjamin Reichenwallner and Peter Meerwald-Stadler. 2023. Simplification of General Mixed Boolean-Arithmetic Expressions: GAMBA. In *WORMA'23*.

[50] Rolf Rolles. 2009. Unpacking Virtualization Obfuscators. In *USENIX Conference on Offensive Technologies (WOOT'09)*.

[51] Jonathan Salwan, Sébastien Bardin, and Marie-Laure Potet. 2018. Symbolic deobfuscation: from virtualized code back to the original. In *DIMVA*.

[52] Eric Schkufza, Rahul Sharma, and Alex Aiken. 2013. Stochastic superoptimization. *ACM SIGARCH Computer Architecture News* (2013).

[53] Moritz Schloegel, Tim Blazytko, and Moritz Contag et al. 2022. Loki: Hardening Code Obfuscation Against Automated Attacks. In *USENIX Security*.

[54] 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).

[55] Jon Stephens, Babak Yadegari, and Christian Collberg et al. 2018. Probabilistic Obfuscation Through Covert Channels. In *IEEE EuroS&P*.

[56] El-Ghazali Talbi. 2009. *Metaheuristics: From Design to Implementation*. Wiley Publishing.

[57] VM Protect Software. 2020. VMProtect Software Protection. http://vmpsoft.com.

[58] Dominik Wermke, Nicolas Huaman, and Yasemin Acar et al. 2018. A Large Scale Investigation of Obfuscation Use in Google Play. In *Computer Security Applications Conference, ACSAC*.

[59] Babak Yadegari and Saumya Debray. 2015. Symbolic Execution of Obfuscated Code. In *Conference on Computer and Communications Security*.

[60] Babak Yadegari, Brian Johannesmeyer, Ben Whitely, and Saumya Debray. 2015. A Generic Approach to Automatic Deobfuscation of Executable Code. In *Symposium on Security and Privacy, SP*.

[61] Li Yixuan, Julian Pasert, and Elizabeth Polgreen. 2024. Guiding Enumerative Program Synthesis with Large Language Models. In *Computer Aided Verification*.

[62] Yongho Yoon, Woosuk Lee, and Kwangkeun Yi. 2023. Inductive program synthesis via iterative forward-backward abstract interpretation. PLDI (2023).

[63] Yongxin Zhou, Alec Main, and Gu et al. 2007. Information Hiding in Software with Mixed Boolean-arithmetic Transforms. In *Conference on Information Security Applications*.