Automating binary code reverse engineering with black-box solutions

Vidal Attias, CEA List, Loria.
Under supervision of Grégoire Menguy, CEA List, Sébastien Bardin, CEA List and Jean-Yves Marion, Loria

Abstract—We motivate the need for exploration of black-box based deobfuscation techniques. We show that the traditional white-box approach rapidly gets mitigated and has fundamental semantical limitations. Our work builds upon an emerging field of deobfuscation based on black-box techniques and demonstrates that although current state of the art solutions outperform white-box techniques, they should be improved and should support additional capabilities.

Index Terms—Binary analysis, reverse engineering, deobfuscation, black-box

I. INTRODUCTION

Software may contain sensitive information, such as industrial proprietary algorithm or an exploit used by a malware, that developers may want to conceal. In the Man-at-the-End attacker scenario [2], the user is the attacker and can read, run and modify the code directly. A widespread solution called obfuscation transforms the original code, making it harder to understand and/or analyze automatically. Formally, given a program \( P \), obfuscation creates a new equivalent program \( P' \) that is harder to analyze. Unfortunately, no bullet-proof obfuscation exists [2]. Still, various protections arose to make reverse engineering as hard as possible [5].

Deobfuscation. On the opposite side, deobfuscation aims at retrieving a program close to the original code that is conceivably analyzable or legible. Formally, given an obfuscated program \( P \), a deobfuscation algorithm \( A \) returns \( P' \), a program semantically equivalent to \( P \) but tractable to analyze.

The main challenge of analyzing heavily obfuscated binary code resides in simplifying the code. Traditional approaches employ white-box based techniques (symbolic execution, taint analysis, abstract evaluation, ...). They reason on the actual code and prove properties regarding variable’s values over the program’s course. Such techniques include: i) Forward taint analysis which consists in following all instructions affected by input variables, hence identifying the semantically important parts of the code and avoiding spending time analyzing appended obfuscated code; ii) Backward analysis, germane to forward taint analysis, will identify the instructions that affect the code’s outputs and; iii) Symbolic execution which builds a representation of the possible states each variable can take during the code’s execution by tracking each assignment and using an intermediate language.

While they have been shown very powerful, these analyses share a common limitation: they must read the code and maintain a state throughout the analysis. Hence, they are very brittle regarding syntactic obfuscation.

Breaking white-box deobfuscation. Taking advantage of such a limitation, recent obfuscation techniques [6] have proven intractable by traditional white-box methods. They increase the code’s syntactic complexity so much that white-box methods fail to scale. For instance, Path-based protection [8] is tailored to kill symbolic-based deobfuscation [3], [12]. It accentuates the path explosion problem by artificially augmenting the number of possible paths a symbolic execution engine would have to explore to provide an exhaustive view of the program. Mixed-boolean arithmetic (MBA) encoding [11], on the other hand, aims to make arithmetic and boolean expressions unintelligible and hard to solve for SMT solvers. It relies on the fact that solvers are usually very efficient at understanding boolean operations and arithmetic separately but fall short when combining both. Hence, modern obfuscation tools will convert pure arithmetic or boolean expressions into mixed expressions. For example, a simple \( x + y \) expression can be rendered as \( x \oplus y + 2 \cdot (x \land y) \), which is much harder to understand. Of course, MBA encoding is usually applied multiple times to increase the expression complexity even more. Last but not least, Virtual Machine (VM) based obfuscation converts the original binary’s Instruction Set Architecture (ISA) into a custom one and embeds in the code an interpreter to emulate this custom ISA. This thwarts traditional white-box deobfuscation tools that would provide analysis on the interpreter but fail to understand the custom ISA (which can be obfuscated itself).

From white-box to black-box. Since these techniques increase syntactic complexity to the point that traditional white-box deobfuscation tools break, there is a need for syntactically resilient methods. Recently, new approaches [4], [7] relying on program synthesis [1] have been proposed. They are completely black-box and are not impacted by the syntactical complexity of the underlying code. Thus, they bypass completely the anti-white box analysis protections.

In this paper. The following sections present:
- A first review of the state-of-the-art of black-box deobfuscation and discuss their current limitations;
- A discussion on interesting research directions to build on black-box deobfuscation, namely reverse window splitting, algorithmic correction and grey box approaches

II. DEOBFUSCATION

A way to be syntactically resilient consists in considering only the relation between a code’s inputs and outputs and synthesize an alternative code that is semantically equivalent.
We reformulate here the deobfuscation problem for the \textit{black-box} approach.

**Black-box deobfuscation.** Given the input $I$ and output $O$ of an unknown (i.e., computationally intractable to analyze) code $P'$, find a code $P$ which outputs $O'$ on $I$ are equivalent to $O$.

*Program synthesis* generates code that is semantically equivalent to the source, focusing only on the \textit{input/output} relations. *Program synthesis* functions in two main parts. \( i) \) sampling a certain amount of input/outputs and \( ii) \) generating a candidate expression that matches the sampling. If the expression doesn’t match, then it can be mutated by different means.

1) \textit{Syntia}: Syntia is the seminal paper for program synthesis applied to code deobfuscation [4]. Candidate expressions are generated using Monte Carlo Tree Search (MCTS) in the space of possible expressions. In order to avoid local minima, a corrected Simulated Annealing puts emphasis on the exploration in the beginning and focuses on guiding as synthesis goes. However, it has been shown [7] that Syntia does not perform better than an enumerative search. Nonetheless, Syntia paved the way to program synthesis methods and being insensitive to syntactic deobfuscation it yields excellent results against current obfuscation strategies. Syntia proved efficient against VM obfuscation and MBA although it was pretty slow.

2) \textit{Xyntia}: Xyntia is an evolution of Syntia [7]. After realizing that MCTS does not provide actual guidance, Menguy et al. proposed to replace MCTS with \textit{locally iterated search} techniques. The main difference lies in the search space. MCTS works in the space of candidate expressions, maintaining a score for each visited candidate expression, while Xyntia only maintains an \textit{abstract syntax tree} and randomly mutates parts of this tree. In general, a mutation is discarded if the distance between the new expression and the sampled outputs is larger than the previous expression’s. However, in place of Simulated Annealing, Xyntia avoids local maxima allowing some random mutations to be applied even though the distance increases, called \textit{perturbations}. Furthermore, it redefines the formal framework, expressing it as an optimization problem rather than a single player game. This allows to better manipulate the distance function that is used to compare the generated expression against the sampled outputs.

Xyntia has been shown to actually be guided, yielding better results than Syntia on a synthetic dataset, especially for small timeout experiments. On a 1s run per synthesis, Xyntia synthesized 78\% of the expressions when Syntia only managed to get 18\% and for a timeout of ten minutes, Xyntia had a success rate of 95\% and Syntia around 41\%.

### III. Future work

We believe there is still a lot of work to be done. Notably, [7] and [10] show that migrating from \textit{syntactical obfuscation} to \textit{semantical obfuscation} can thwart black-box based methods. Such protection can be achieved by the following methods:

- **Handlers merging.** Several semantically simple handlers can be merged into a larger one that would be intractable. Although a naive implementation would have easily separable independent outputs, some arithmetic transformations help blend all the operations.

- **Handlers division.** In contradistinction with \textit{handlers merging}, a semantically simple handler is divided into a collection of handlers that are highly complex taken separately but when computed sequentially output the same result as the original one. This assumes that the code is divided into windows matching exactly the definition of handlers. An additional obfuscation is to duplicate the handlers into different forms that compute the same subresult but are hard to match between them.

In order to mitigate these protections and future ones, Xyntia and black-box deobfuscation in general need improvement. During this doctoral study which started in October 2023, we intend to explore the following directions.

- **Reverse window splitting:** A first goal could be to improve the quality of the \textit{reverse window} that we consider, i.e. the part of the code we want to synthesize. This would greatly impact the performance as a large window can result in covering more semantic than the one we are interested whereas a short window may only catch intermediary values of the output, inducing a more complex semantic as well as unexploitable results.

- **Correction:** Current black-box implementations have no correction guarantee other than on the sampled input/outputs. A way to mitigate this is a variation of *Counter-example guided inductive synthesis* (CEGIS) in which we test a successfully synthesized expression and try to find an input whose output differs from the original obfuscated code. We then run the deobfuscation again by adding this input in the sampled set, hoping to find a more suitable expression.

- **Grey-box approaches:** The rule of thumb is that white-box implies high semantic capabilities and black-box syntactic insensitivity. QSynth [9] proposes to combine both by scanning the obfuscated code’s AST, identify parts whose input/output match with some expressions in a precomputed table and swap the matching parts. Using modern program synthesis techniques may further enhance performance.

### References