ESANN 2026 proceedings, European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning. Bruges (Belgium) and online event, 22-24 April 2026, i6doc.com publ., ISBN 9782875870964. Available from http://www.i6doc.com/en/.

Deobfuscation as a GNN-Based Graph-Edit Problem by Reinforcement Learning

Roxane Cohen[1,2], Robin David[1], Samuel Hangouët[1], Florian Yger[3] and Fabrice Rossi[4] *

1- Quarkslab, Paris
2- LAMSADE, CNRS, UMR 7243, Université Paris-Dauphine - PSL
3- INSA Rouen Normandie, Normandie Univ, LITIS UR 4108, Rouen
4- CEREMADE, CNRS, UMR 7534, Université Paris-Dauphine - PSL

Abstract. Obfuscation is a software protection technique that transforms a program's binary code to conceal its behavior and to hinder analysis. Conversely, deobfuscation is an adversarial process that seeks to partially or fully remove the applied obfuscation in order to recover the original, unobfuscated code. This work introduces the first deobfuscation framework based on Reinforcement Learning (RL). It models an obfuscated function's binary code using a novel graph representation integrating both data and control-flow. The graph is then progressively simplified through a sequence of graph-edit operations, selected iteratively by a Graph Neural Network (GNN)-based agent operating within a RL pipeline. Experiments demonstrate promising results on Mixed Boolean Arithmetic (MBA) obfuscation, where multiple variants of diverse expressions can successfully be simplified into valid deobfuscated variants.

1 Introduction

Binary code obfuscation aims to protect programs from external analysis by transforming their binary code such that the semantics of the program becomes significantly more difficult to understand. Such software protection techniques serve legitimate purposes (safeguarding intellectual property) or unethical ones (concealing malicious malware behavior). For example, the Mixed Boolean Arithmetic (MBA) transformation replaces simple calculations with more complex ones, yet semantically equivalent. An addition expression, expressed as `add rax, rbx` in `x64` assembly, may thus be expanded into multiple instructions, such as:

$$x + y \equiv \sim ((\sim x) + (-(x \oplus (x \oplus y))))$$

Deobfuscation, defined as the partial or complete removal of obfuscation, is a challenging task, both theoretically and practically [1]. Existing approaches rely on expert-designed algorithms tailored to specific obfuscation techniques [6], on classical Machine Learning (ML) methods [8], or on formulations that cast the task as a specialized instance of program synthesis [2]. While Reinforcement Learning (RL) has shown promising results for program synthesis, it has not been used for deobfuscation.

*This work was partially supported by the Agence Innovation Defense (AID).

ESANN 2026 proceedings, European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning. Bruges (Belgium) and online event, 22-24 April 2026, i6doc.com publ., ISBN 9782875870964. Available from http://www.i6doc.com/en/.

Deobfuscation operates on binary programs in which most of the abstraction conveyed by the source code has been removed, leaving only low-level instructions. By disassembling the binary, one extracts those instructions and recovers higher-level constructs, mainly program functions and interactions (calls) between them. A function is thus usually represented as a Control-Flow Graph (CFG) that captures the execution flow between code blocks, denoted as Basic Block (BB).

In the presence of obfuscation, both the contents of individual BB and the CFG structure may be altered. In this context, deobfuscation can be formulated as a structured prediction problem, where the goal is to transform a structured input (the obfuscated graph) into a structured output (the unobfuscated graph). Using RL to iteratively construct structured predictions has proven effective [5] when standard vector inputs are considered. A natural extension consists in combining Graph Neural Network (GNN) to handle graph-structured input and RL for graph-structured output.

In this work, we first propose a new graph format to represent functions recovered from binary code with both data and control flow (Section 2). Then we use RL to modify iteratively the graph of an obfuscated function to build a valid deobfuscated variant of the graph (Section 3). The method is illustrated on MBA obfuscated code (Section 4).

2 A new binary graph: combining data and control-flow

A function binary code is primarily represented as a CFG. However, this graph does not explicitly encode data-related information, which is instead implicitly embedded within CFG nodes. Recent efforts have focused on designing richer graphs that capture both function control and data relationships, such as Semantic Oriented Graph (SOG) [4]. While the SOG exhibits several properties beneficial for binary analysis, such as instruction reordering invariance, it does not retain assembly data which is critical for comprehensive binary code analysis. To overcome this limitation, we propose a new graph representation, the Control-Data Graph (CDG). It preserves the core ideas of the SOG while providing a more lightweight and flexible structure, detailed node types, data operations and control-flow execution, suitable for supporting a wider range of downstream binary code analysis tasks.

Definition 2.1 (Control-Data Graph (CDG))**.** Given a function f, its CDG is a directed graph $G = (V, A, \mu, \psi)$, where V is the set of nodes, representing code elements with τ_V = (Input, Output, Constant, Operation, Phi, Memory) an associated type. A are edges representing relationships between code elements $\mu : V \to \tau_V$ and are associated with a type τ_A = (Data, Control).

`Input nodes` are input registers that are read before any prior write operation, and have no predecessors (e.g., green `RAX`, `RBX` in Figure 1). Similarly, `output nodes` hold the final computed values, written without subsequent reads, and have no successors (e.g., blue `RAX`, `RCX` in Figure 1). `Constant nodes` correspond to fixed values, such as immediates, that remain unchanged throughout execution. `Operation nodes` capture the computations performed on data within the

ESANN 2026 proceedings, European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning. Bruges (Belgium) and online event, 22-24 April 2026, i6doc.com publ., ISBN 9782875870964.
Available from http://www.i6doc.com/en/.

function. `Phi nodes` serve as merging state nodes, converging multiple execution paths into one. Finally, `memory nodes` partially model memory cells that are read from or written to, without providing aliasing support.
This work is restricted to function-level MBA obfuscation. As such, the obfuscated instructions are all executed sequentially inside a single code block, denoted as Basic Block (BB), without control-flow redirection, as shown on Figure 1.

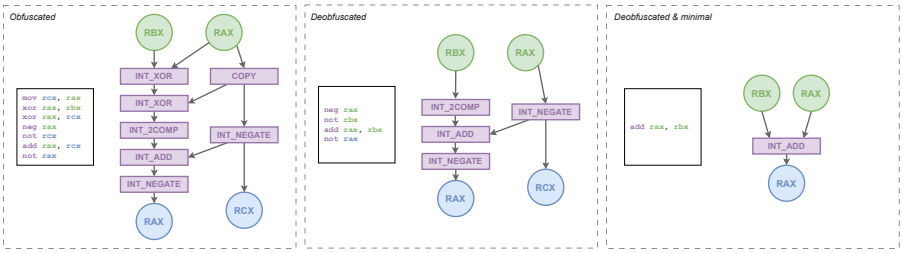


Fig. 1: Obfuscated CDG (left), deobfuscated CDG (center), minimal deobfuscated CDG (right). None of the graphs contains control-flow redirection.

Given a function, its CDG is constructed by first translating the function's assembly code into an Intermediary Representation (IR), in our case, Pcode[1], which models the semantic of instructions in an architecture agnostic language. The resulting sequence of Pcode instructions is then analyzed to recover data and control dependencies between inputs, operations, and outputs, ultimately producing the CDG. The difficulty of generating the CDG depends on the complexity of the function: functions limited to data-flow operations are relatively straightforward, whereas those involving branching, loops, or indirect memory accesses raise substantially greater challenges.

3 RL framework

Starting from an obfuscated CDG, we aim to find a simplified semantically equivalent deobfuscated CDG using graph edit operations chosen by RL.

Our agent operates in a graph valued environment: at a given step of the process, the state of the environment is a CDG. The initial state is the CDG. The agent has no internal state but maintains access to the initial CDG all along the process (see below). The action space is limited to 5 categories of actions: *termination* (i.e. the agent is satisfied by the current CDG), node *insertion* or *deletion*, edge *insertion* or *deletion*.

With the exception of the termination action, each operation requires additional parameters that the agent must specify e.g. the node to delete for a node deletion action. However, in a graph-based state setting, selecting an action and its associated parameters, such as identifying which node or edge to insert or delete, is complex. Despite extensive research on RL, its application to problems

[1] `https://ghidra.re/ghidra_docs/languages/html/pcoderef.html`

ESANN 2026 proceedings, European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning. Bruges (Belgium) and online event, 22-24 April 2026, i6doc.com publ., ISBN 9782875870964. Available from http://www.i6doc.com/en/.

characterized by large graph-based environment state spaces remains relatively underexplored.

In fact, in graph-based environment states, the action space is inherently tied to the input graph. Selecting a node, for example the fourth node, as an action does not have a consistent semantic meaning across graphs, or even across different permutations of the same graph, since node identifiers are arbitrary and lack intrinsic meaning. This permutation dependence, directly related to the graph isomorphism problem, implies that two isomorphic graphs with different node labelings could result in distinct action spaces, potentially leading to divergent agent behaviors, even though the underlying task remains identical. This challenge highlights the need for RL methods that are invariant or equivariant to node permutations.

In this context, leveraging Graph Neural Network (GNN) architectures is particularly advantageous for RL-based approaches. The agent's architecture consists of multiple GCN layers, followed by several output layers, or heads. The first head produces action scores from the current CDG embedding, from which the action is selected by sampling from the corresponding logits distribution. The second head computes node scores from node embeddings, enabling the selection of a node when the action involves node edits. Similarly, the third head produces edge scores for actions affecting edges. However, this GNN design alone does not allow the agent to retain information about the original objective, i.e., the initial obfuscated CDG. To address this limitation and preserve awareness of the temporal evolution of the simplification process, the embedding of the initial graph is also provided to the agent. This maintains awareness of the initial simplification objective, even as nodes or edges are deleted.

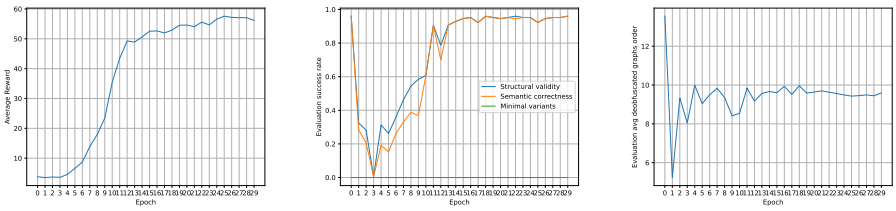
In this setting, starting from a given obfuscated CDG, the agent performs a sequence of graph-edit operations until either a maximum number of steps L is reached or the agent decides to terminate the episode. The resulting deobfuscated graph is then evaluated by a reward function, which assesses its validity according to multiple criteria. First, the resulting graph must remain structurally valid, demonstrating that the agent has learned to preserve the syntactic constraints of the CDG, such as no operation node has more than two predecessors. Second, it must be semantically equivalent to the original obfuscated graph. In this work, such semantic equivalence is assessed using Satisfiability Modulo Theories (SMT)-based verification, using Z3 [3], which is the only way to guarantee correctness, although this is not always feasible in practice.[2] Third, the deobfuscated graph should be smaller than the initial graph, otherwise the simplification has failed. Such simplification is measured as the difference in node count between obfuscated and simplified graphs. If a deobfuscated CDG cannot be further simplified, it is considered as minimal. Careful design of the reward function is crucial to avoid undesirable RL behaviors. For example, if structural and semantic correctness are rewarded too heavily relative to simplification, the agent may learn that doing nothing yields a higher return than actively simplifying the graph.

[2]Although SMT solving is theoretically sound for MBA, it may become computationally intractable for functions using non-linear arithmetic or branching conditions.

ESANN 2026 proceedings, European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning. Bruges (Belgium) and online event, 22-24 April 2026, i6doc.com publ., ISBN 9782875870964. Available from http://www.i6doc.com/en/.

4 Methodology and experimental results

These experiments are restricted to MBA, drawn from the smallest variants of the Loki obfuscator [7]. The resulting dataset comprises 5,000 MBA variants, for multiple initial expressions: $x + y$, $x - y$, $x \,\&\, y$, $x \,|\, y$, $x \oplus y$. Each node in the initial obfuscated CDG is represented by a 140-dimensional feature, that consists of: a one-hot encoding of the node type; the size of the corresponding code element; the one-hot encoded opcode for Operation Nodes; the one-hot encoded register name for Input and Output Nodes; and, for Constant Nodes, the associated bit value. Edge features are not exploited in this work, although they could be defined, such as indexed-incoming edge. footnoteThe order of assembly operands matters, and the x64 instructions `add rax, rbx` and `add rbx, rax` are not semantically equivalent. Edges could be indexed to preserve operand ordering. The agent employs a three-layer GCN with 512-dimensional node embeddings, which size was chosen as a trade-off between representational capacity and computational cost, as smaller sizes reduced performance and larger ones exceeded memory limits. Alternative GNN layers, such as Graph Isomorphism Network (GIN), did not produce results that were significantly different from those of GCN. Action logits are computed using a sum readout over node embeddings, followed by a fully connected layer with five outputs (four graph edit operations and termination). Action parameters are generated by applying fully connected layers to the final GCN embeddings, yielding parameter scores rather than embeddings; these scores serve as the initialization of the corresponding probability distributions from which parameters (either nodes or edges) are sampled. The evaluated RL algorithm is REINFORCE [9], with a 64-batched implementation, trained over 30 epochs with a discount factor $\gamma = 0.99$ and a maximum of $L = 15$ steps per episode. The average reward over time is illustrated in Figure 2a, whereas structural validity, semantic correctness and minimal success rates are shown in Figure 2b. The average deobfuscated graph orders are shown in Figure 2c.



(a) Average agent reward across time.

(b) Structural, semantic and minimal success rates.

(c) Average deobfuscated graph orders.

These plots demonstrate that the agent successfully learns a policy leading to valid deobfuscated graphs. Initially, the agent learns to generate syntactically valid graphs, and only subsequently begins to preserve semantic correctness, as evidenced by the consistently higher syntactic success rate compared to the semantic one. As expected, the average graph order decreases rapidly, while

ESANN 2026 proceedings, European Symposium on Artificial Neural Networks, Computational Intelligence and Machine Learning. Bruges (Belgium) and online event, 22-24 April 2026, i6doc.com publ., ISBN 9782875870964. Available from http://www.i6doc.com/en/.

both syntactic and semantic success rates increase to satisfactory levels, reaching approximately 95%. However, the minimal success rate remains at zero, due to the extreme rarity of trajectories that yield minimal graphs within the vast action space, making such trajectories unlikely to be sampled during simulation. Consequently, the agent, already achieving reasonably high rewards, lacks incentive to pursue further simplification, despite employing various techniques to encourage exploration, including entropy-based regularization, intrinsic rewards, and a minimal-based replay buffer. More advanced RL methods may yield improved performance on tasks characterized by such large action spaces.

5 Conclusion

In this work, deobfuscation is formulated as a graph simplification problem, where an obfuscated CDG is sequentially modified using RL. The results demonstrate that applying RL to graphs is not only feasible but also effective, as valid deobfuscation is consistently achieved for MBAs based on diverse expressions. This work offers methodological contributions to both the fields of RL and binary code deobfuscation, although it requires further development before being applied in practical deobfuscation scenarios. Key extensions include ensuring minimality of deobfuscated graphs, supporting increasingly complex MBA and exploiting both CDG data-flow and control-flow dependencies, which theoretically enables deobfuscation on both function-level data and control-oriented obfuscations at the same time, an aspect often neglected in existing literature.

References

[1] A. Appel. Deobfuscation is in np. *Princeton University, Aug*, 21(2), 2002.

[2] V. Attias, N. Bellec, G. Menguy, S. Bardin, and J.-Y. Marion. Augmenting search-based program synthesis with local inference rules to improve black-box deobfuscation. 2025.

[3] L. De Moura and N. Bjørner. Z3: an efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'08/ETAPS'08, page 337–340, Berlin, Heidelberg, 2008. Springer-Verlag.

[4] H. He, X. Lin, Z. Weng, R. Zhao, S. Gan, L. Chen, Y. Ji, J. Wang, and Z. Xue. Code is not natural language: Unlock the power of semantics-oriented graph representation for binary code similarity detection. In *33rd USENIX Security Symposium (USENIX Security 24), PHILADELPHIA, PA*, 2024.

[5] F. Maes, L. Denoyer, and P. Gallinari. Structured prediction with reinforcement learning. *Machine learning*, 77(2):271–301, 2009.

[6] B. Mariano, Z. Wang, S. Pailoor, C. Collberg, and I. Dillig. Control-flow deobfuscation using trace-informed compositional program synthesis. *Proceedings of the ACM on Programming Languages*, 8(OOPSLA2):2211–2241, 2024.

[7] M. Schloegel, T. Blazytko, M. Contag, C. Aschermann, J. Basler, T. Holz, and A. Abbasi. Loki: Hardening code obfuscation against automated attacks. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 3055–3073, 2022.

[8] R. Tofighi-Shirazi, I.-M. Asavoae, P. Elbaz-Vincent, and T.-H. Le. Defeating opaque predicates statically through machine learning and binary analysis. In *Proceedings of the 3rd ACM Workshop on Software Protection*, pages 3–14, 2019.

[9] R. J. Williams. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning*, 8(3):229–256, 1992.