Generating Effective Software Obfuscation Sequences with Reinforcement Learning

Huaijin Wang, Shuai Wang, *Member, IEEE,* Dongpeng Xu, Xiangyu Zhang, and Xiao Liu

Abstract—Obfuscation is a prevalent security technique which transforms syntactic representation of a program to a complicated form, but still keeps program semantics unchanged. So far, developers heavily rely on obfuscation to harden their products and reduce the risk of adversarial reverse engineering. However, despite its spectacular progress, one crucial hurdle is that each of existing obfuscation method is designed specifically for obfuscating one program feature (e.g., identifier name, control flow), so an effective obfuscation scheme usually composes a considerable amount of different obfuscation methods. Therefore, one primary challenge lies in identifying effective combinations of obfuscation methods. In this research, we propose a principled technique for generating an optimal program obfuscation scheme by adopting a reinforcement learning approach. Given a program and a set of obfuscation transformations, a reinforcement learning model is progressively trained to select a sequence of obfuscation transformations, such that applying each transformation in order towards the program yields the optimal obfuscation result, making programs dissimilar while retaining reasonable instrumentation overhead. Our implementation can directly work on raw binary executables without source code, and our evaluation demonstrates that the trained models can effectively obfuscate executable files with low cost.

Index Terms—Software obfuscation, reinforcement learning, reverse engineering, software similarity

1 Introduction

Software obfuscation [1] is a semantic-preserving transformation aiming to notably change a program's syntactic appearance. Since the transformed program looks significantly different from the original program, it prevents attackers from understanding, reverse engineering, or even extracting some features related to the original program. Software obfuscation techniques have been widely used in software protection [2], information hiding [3], cryptography [4], and malware development [5], [6]

A great variety of obfuscation methods have been developed for transforming software from different aspects, e.g., scrambling identifier names [1], [7], opaque predicate [2], data encoding [3], and control flow flattening [8]. However, only applying one type of obfuscation cannot produce a significantly different program, because each of these existing methods only obfuscates one particular software feature without changing others. Therefore, effectively obfuscating a program requires 1) an elaborate scheme of synergizing various obfuscation methods; 2) iteratively applying the obfuscation methods in the scheme to the program. Particularly, the scheme must consider performance cost, since obfuscation methods often introduce redundant structures, leading to considerable execution slowdown (e.g., by complicating program control structures). Consequently, how to select and combine various obfuscation methods to effectively and efficiently obfuscate a program still remains a challenge to existing research.

- *Huaijin Wang and Shuai Wang are with the Department of Computer Science and Engineering, HKUST, Hong Kong SAR. E-mail: (hwangdz@cse.ust.hk; shuaiw@cse.ust.hk)*
- *Dongpeng Xu and Xiangyu Zhang are with University of New Hampshire, NH 03824 USA. E-mail: (dongpeng.xu@unh.edu; xz1057@wildcats.unh.edu)*
- *Xiao Liu is with Facebook Inc., CA 94025 USA. E-mail: bamboo@fb.com*

In this work, we propose a new method to automatically generate optimal obfuscation schemes based on reinforcement learning (RL). The key idea is to let an RL model guide the selection, configuration and scheduling of different obfuscation methods, so as to achieve a high-quality obfuscation result, i.e., the transformed program is remarkably different from the original program with only trivial performance overhead. Given a program P and a set of obfuscation transformations $T = \{t_1, t_2, \ldots, t_n\}$, our method outputs a sequence of obfuscations $t_i, t_j, \ldots, t_k$, where $i, j, \ldots, k \in [1, n]$. Iteratively applying this sequence of obfuscations to program P will produce a highly-obfuscated program with low-performance cost.

To this end, we design and implement RLObf, a practical tool employing deep reinforcement learning (DRL), particularly Deep Q-Network (DQN) to synthesize low-cost obfuscation sequences for use in practice. The DQN model is rewarded by simultaneously considering: 1) similarity between the obfuscated program and the input program, and 2) execution slowdown of the obfuscated program. A sequence of obfuscations gets a higher reward if the similarity score decreases significantly while does not incur too much execution slowdown.

We have implemented RLObf to directly transform x86 program executables, and we have conducted an evaluation on a comprehensive set of commonly-used Linux applications with diverse functionality. Our evaluation demonstrates promising findings: RLObf can successfully find obfuscation sequences of a good quality for *all* the evaluated executable files to reduce the similarity score to less than 0.68 — the average score for two different programs in our dataset. More importantly, the synthesized obfuscation sequences impose only 18.8% execution slowdown to the obfuscated software. Further evaluation shows that obfuscated programs are highly obscure and stealthy w.r.t.

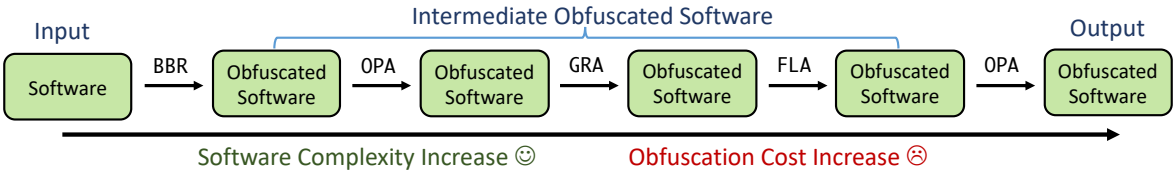


Fig. 1. The dilemma of constructing obfuscation sequences. With more obfuscation passes applied, software becomes more complex, but could lead to higher cost (e.g., execution slowdown), and vice versa. Abbreviations (e.g., `BBR`, `OPA`, `FLA`) denote different obfuscation methods (see Table 1).

standard metrics, and security impact evaluation shows that obfuscated programs can primarily eliminate chances for code reuse attacks [9] and adversarial reverse engineering. Model interpretation analysis shows that our trained models can successfully capture subtle but critical features from software control structures and make decisions properly. We also show that the obfuscated programs can evade binary diffing effectively, while incurring very low cost, compared to randomly obfuscated programs. To our knowledge, RLOBF is the first work to synthesize obfuscation sequences with AI techniques, addressing a key challenge in today's cybersecurity landscape. In summary, we make the following contributions:

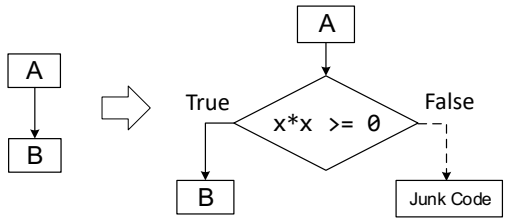
- We advocate a new focus to leverage deep reinforcement learning model to promote software obfuscation by synthesizing high-quality and low-cost obfuscation sequences. We design a unified and systematic framework for generating diverse and comprehensive obfuscation results.
- We implement RLOBF, a practical tool to directly process *executable files*, being generic to software written in any programming language and significantly broadening the application scope.
- Our extensive evaluation has synthesized optimal obfuscation sequences to all the tested Linux applications with modest cost despite their diverse functionality, by successfully capturing critical features on program control structures.
- We will publicly maintain RLOBF to benefit follow-up research. To facilitate the reproduction and reviewing of our results, an snapshot of RLOBF's codebase has been uploaded to GitHub [10]. Evaluation data, including the trained models and the obfuscated programs, have been shared publicly as well [11].

before	after

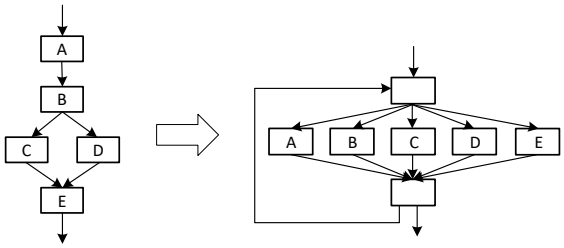
`movsb`	`push eax`
	`mov al, [esi]`
	`inc esi`
	`mov [edi], al`
	`inc edi`
	`pop eax`

`mov eax, 0`	`xor eax, eax`
`add eax, 1`	`not eax` `neg eax`

(a) Instruction replacement substitutes one instruction with a sequence of syntactically different but semantics-preserving instructions.



(b) Opaque predicate inserts a tautology path condition that is hard to analyze, but will be always evaluated to one direction during runtime.



(c) Control flow flattening transforms control flow graph (CFG) into a `switch` statement, which leverages two dispatcher nodes (top and bottom on the flattened CFG) to guide the control flow transfers and decides the target basic blocks.

Fig. 2. Three software obfuscation methods.

2 Background and Motivation

2.1 Software Obfuscation

Obfuscation is an important and timely topic for protecting software systems from adversary analysis. Typical obfuscation techniques substantially change program control flow structures and have been shown to be effective in hiding the underlying functionalities and complicating the execution flow of a program. The strategy of using obfuscation to defeat adversary analysis can be traced back to Collberg et al. [2], [12]. Since then, many research efforts have been made to develop obfuscation techniques for various scenarios [13], [14], [15]. Obfuscation methods play a crucial role in protecting software from malicious reverse engineering. Fig. 2 lists three commonly used software obfuscation methods. They make a program more complex in terms of instructions, path conditions, and control flow, while still preserving the original functionality.

To date, a large set of obfuscation methods has been developed. As shown in Fig. 1, the common practice is to iteratively transform software by pipelining and syner-

gizing obfuscation passes, until getting a satisfied result. Each obfuscation pass reasonably complicates the program presentation, and can potentially generates new program components (e.g., more basic blocks) to be used for further iterations of obfuscation. For example, a popular obfuscation framework, LLVM-Obfuscator [16], provides three obfuscation methods and allows users to specify the number of iterations and which obfuscation to use for each pass.

2.2 Deep Reinforcement Learning (DRL)

RLOBF is built on top of a DRL model for synthesizing high-quality obfuscation traces. Before presenting the details of RLOBF we first introduce the basic background of RL and Deep Q Learning. The following formulation of a typical RL process is related to the presentation given in [17].

RL is a machine learning method for an agent to learn behavior through a trial/error interaction with a dynamic environment. Various behaviors cause different rewards/penalty from the environment and the learning goal aims to maximize the cumulative reward. The environment is normally described as a Markov decision process (MDP), and the whole learning process can be formalized as a stochastic process. For every step in the MDP, the agent gets reward/penalty by interacting with the environment and the states in the environments also update accordingly.

We present the formal description of a typical RL process as follows. First, we define an MDP as a triple $\mathcal{M} = (\mathcal{X}, \mathcal{A}, \mathcal{P})$, where $\mathcal{X}$ represents a set of states in the environment, $\mathcal{A}$ is a set of actions an agent can take, and $\mathcal{P}$ represents the *transition probability kernel*, which assigns a probabilistic value denoted as $\mathcal{P}(\cdot|x, a)$ for each state-action pair $(x, a) \in \mathcal{X} \times \mathcal{A}$. For each $U \subset \mathcal{X} \times \mathbb{R}$, $\mathcal{P}(U|x, a)$ is the probability such that performing action a at state x induces the system to transition from x into $x' \in \mathcal{X}$ and to return reward value $U \in \mathbb{R}$. Therefore, for any state transition triple $(x, a, y) \in \mathcal{X} \times \mathcal{A} \times \mathcal{X}$ that corresponds to the state transition from x to y with action a, the probability is defined as follows:

$$P(x, a, y) = \mathcal{P}(y \times \mathbb{R}|x, a)$$

$\mathcal{P}$ also provides the immediate reward function, namely, $r : \mathcal{X} \times \mathcal{A} \to \mathbb{R}$, which specifies the expected immediate reward $r \in \mathbb{R}$ that is received if action a is selected at state x as follows:

$$r(x, a) = \mathbb{E}[R_{(x,a)})]$$

Therefore, in the stochastic process $(x_{t+1}, r_{t+1}) \sim \mathcal{P}(\cdot|x_t, a_t)$, the objective of the agent is to select a sequence of behaviors that maximizes the expected cumulative rewards:

$$\mathcal{R} = \sum_{t=0}^{\infty} \gamma^t R_{t+1}$$

Typically, $\gamma \in (0, 1)$; therefore, according to the above formula, rewards received in the near future are of higher priority than those rewards received at the later stages. Let $\pi(a|x)$ to denote the probability of action a being taken at state x, where policy π maps states to actions (i.e., $\pi : \mathcal{X} \to A$). Then, we define a Q function

$$Q^{\pi}(x, a) = \mathbb{E}\Big[\sum_{t=0}^{\infty} \gamma^t R_{t+1} | X_0 = x, A_0 = a\Big], x \in \mathcal{X}, a \in \mathcal{A}$$

where A_0 is the first action that is selected randomly and X_0 is the starting environment. $Q^{\pi}(x, a)$ denotes the cumulative reward for an agent that is received along the interaction process according to a policy π. To approximate an optimal Q function, usually the Q function is typically expressed into the following notation for update:

$$Q^{\pi}(x_t, a_t) \to \alpha\Big(r_t + \gamma \max_a Q(x_{t+1}, a) - Q(x_t, a_t)\Big)$$

where $\alpha \in (0, 1)$ is the learning rate and γ is the Q function update rate. In the above notation, the value of $Q(x_t, a_t)$ will be updated as the maximum expected future reward that the agent will obtain if it takes action a_t at certain state x_t.

A common method for maintaining and updating the Q function is by using a table, namely, a Q table [18]. However, as the space and number of action candidates largely increase, the performance cost for maintaining a Q table becomes a burden. The state-of-the-art research seeks more "fuzzy" representations to lower the cost. Particularly, Mnih et al. [19] propose deep Q-networks (DQN) for approximating the Q function. They translate the definition of Q function update to the following loss function L. After the translation, the new learning process of DQN aims at minimizing the loss function.

$$L = \Big(r + \gamma \max_a Q(x_{t+1}, a) - Q(x_t, a_t)\Big)^2$$

DQN has been broadly applied to solving practical problems and gained substantial success, e.g., playing strategic board games [20] and video games [21]. In this research, we demonstrate that DQN can be used to synthesize quality obfuscation sequences.

3 Research Motivation

As shown in Fig. 1, the software obfuscator typically transforms an input program for several iterations, each pass applies one obfuscation method toward the (already-obfuscated) code. It is easy to see that a *synergistic effect* could be imposed by composing different obfuscation methods together: an obfuscation method could provide more *transformable code fragments* to its follow-up obfuscation methods [22]. For instance, by first transforming a function with opaque predict insertion, extra basic blocks will be inserted into the function. The obfuscated function can benefit a follow-up control flow flattening obfuscation (cf. Fig. 2c), and further be obfuscated into a complex "switch" statement. The output by stacking two obfuscation methods would be obviously more complex than using only control flow flattening or opaque predication insertion. Hence, the obfuscation sequences become the prominent factor to decide the effectiveness and efficiency of obfuscation campaigns. Although a variety of software obfuscation techniques have been proposed and used in real-world software development, how to effectively adapt and schedule these

obfuscations still remains an unsolved problem. Overall, we note that to present an effective obfuscation sequence, the following two aspects both play a critical role.

Length of Obfucation Sequence. Fig. 1 depicts a practical dilemma in obfuscating software with a sequence of obfuscation passes. By increasing the number of applied passes and obfuscating more program components that could have been obfuscated before, the software naturally manifests less similarity compared to the input program. On the flip side, a usually unforeseen drawback is that "lengthy" obfuscation sequence overly transforms software, imposing noticeable cost (i.e., execution slowdown), and therefore becoming less desired in real-world scenarios. Nevertheless, while using more succinct sequences of obfuscations intuitively reduce the cost, obfuscation can become less comprehensive and therefore the output is still similar to its input program.

Selection of Particular Obfuscation Methods. As discussed in Sec. 2.1, obfuscation schemes vary in granularity from a single instruction to a pair of basic blocks, or even the entire control flow structures. Different obfuscation methods can therefore impose various impacts on obfuscation effectiveness and cost. For instance, as will be discussed in the evaluation of this research (Sec. 6), relatively heavyweight methods like control flow flattening can usually incur higher cost compared to lightweight methods like instruction replacement. Overall, given an obfuscation sequence of N methods (i.e., $t_1, t_2, \ldots, t_k, \ldots, t_N$), to decide a particular obfuscation method for the kth pass (i.e., t_k), both the accumulated cost and effectiveness derived from $t_1, t_2, \ldots, t_{k-1}$ should be taken into account. A heavyweight method could be adopted, if the accumulated cost is still low, and versa vice.

The aforementioned two aspects, if properly handled, constitute a quality obfuscation sequence. Nevertheless, most popular obfuscation tools let users construct obfuscation sequences, which places a heavy burden on users because they rarely have insights on the quality of composed obfuscations. Typically to protect software from adversary analysis (e.g., similarity analysis, the basis of various software attacks [23], [24], [25]), users will need to conduct obfuscations for 5 to 30 iterations and face a large space to explore. For instance, given the three obfuscations provided by LLVM-Obfuscator, number of obfuscation combinations are 3^N, revealing a considerably large search space where N is the length of an obfuscation sequence. Without an in-depth understanding of obfuscations' synergy effect toward specific software, it is challenging, if at all possible, for users to identify optimal sequences. In sum, the lack of guidelines to compose obfuscation schemes may impede reaching the full potential of program obfuscation schemes, placing concern to the quality of obfuscated code. This observation primarily motivates our research on automatically generating obfuscation sequences via DRL.

4 Design of RLObf

As presented in Fig. 3, RLObf can directly take *binary executables* as the input, so that it is not restricted by specific programming languages. Given an executable file, we perform disassemble to recover assembly programs as well as program control structures. We then train an agent to select an obfuscation method (see Sec. 4.1) and transform the recovered assembly code. RLObf is constructed as a DQN with two LSTM layers (the first is bidirectional) followed by a fully connected layer (each layer has 512 neurons). We use tokenizer encoding to encode the assembly opcode for each instruction in the disassembled assembly program and connect the encoding sequence with the LSTM layers.

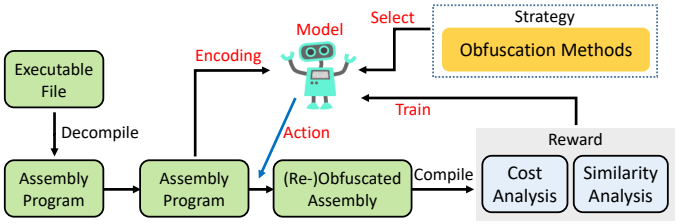


Fig. 3. Workflow of RLObf. Note that the workflow forms an iterative process, where the disassembled assembly program can be re-obfuscated for multiple steps. For each step, we compute a reward to train the model.

When training the model, we measure the quality of applied obfuscation by considering *syntactic similarity* and *performance cost*. Syntactic similarity compares the syntax-level similarity between obfuscated executable file and the input. A low similarity score indicates a better obfuscation result, because it "appears" to be much different from the original program. Note that all obfuscations are semantics-preserving transformation, that is, they only change the syntactic-level difference but not touch the real program behavior. Performance cost measures how much performance overhead is introduced by the obfuscation. An obfuscation is deemed as good if it introduces small overhead.

Program syntactic similarity and performance cost will be used as the learning reward to train the agent (Sec. 4.2). The overall workflow forms an *iterative process* and we re-apply this process to the already obfuscated programs until either the obfuscated executable successfully reduces the similarity rate below a predefined threshold T, or we have applied over N obfuscations (i.e., obfuscation sequence becomes too lengthy). Parameters T and N can be configured by users.

4.1 Action Space

In the task of reinforcement learning, the learning agent is trained to select an action to perform among the action space at each step. For a given binary executable to obfuscate, the learning goal can be interpreted as to select a sequence of obfuscation methods. To implement RLObf, we use seven widely-used, practical obfuscations indexed by existing research survey [26].

The roster of these methods are given in Table 1. These obfuscation methods work on various granularities, ranging from instructions, basic blocks, to functions. We now briefly introduce each obfuscation method and how it is implemented in RLObf. Note that due to the limited space, here we only discuss the design and implementation essential. Readers can always refer to the released code [10] for implementation details of each obfuscation pass.

TABLE 1
Obfuscation methods implemented in RLOBF.

Class	Methods	Abbreviations
Instruction Level	instruction replacement	IR
	garbage code insertion	GRA
Basic Block Level	basic block reordering	BBR
	basic block splitting	BBS
	opaque predict insertion	OPA
	branch function insertion	BFI
Function Level	function reordering	FR
	control flow flattening	CFA
	function inline	FIL

Instruction Replacement. As aforementioned, instruction replacement substitutes one or a sequence of instructions with a set of semantics-identical instructions. For the implementation, we leverage three instruction mappings to pinpoint and substitute one special instruction with its semantics-identical instructions. For instance, `mov %eax, 0` resets register `%eax` with zero, which can be replaced with `xor %eax, %eax`. For each pass, we scan for all substitution candidates and perform the replacement. It is worth noting that although not obvious, some replacement, e.g., from `mov %eax, 0` to `xor %eax, %eax`, could actually change CPU flags into different values and therefore stealthily alter the control flow and semantics. To tackle this challenge, we put `pushf` and `popf` instructions before and after the replaced instruction to temporarily store and retrieve CPU flag values from the stack.

Applying instruction replacing can make the obfuscated programs reasonably differ from the original code. Nevertheless, as instruction replacement does not change the program structures, program obfuscated with instruction replacement can still be matched to its original program according to control structures. Our evaluation shows consistent findings.

Garbage Code Insertion. This obfuscation scheme inserts meaningless instruction sequences (i.e., "garbage code") into the program. While the inserted instructions change the program representation, "garbage code" does not change program functionality, thus can be inserted into arbitrary places. For the implementation, we insert a random number (1–5) of garbage instructions within every function. The three insertion candidates are `nop`, `mov val, val`, and `xchg val, val`, where `val` could be CPU registers or memory cells.

Garbage code can be inserted into arbitrary positions of program layout, exhibiting appealing features to obfuscate program syntactic-level representations with reasonable cost. Nonetheless, this obfuscation scheme does not change program structures as well.

Basic Block Reordering. This obfuscation scheme reorders the relative positions of two basic blocks. To guarantee the functionality correctness, we insert *extra control transfers*, which induce more edges on the control flow graph. To implement this scheme, each pass iterates every function and checks if this function has at least three basic blocks. If so, we randomly select and reorder one pair of basic blocks within that function.

Our tentative studies show that the extra control transfer instructions introduced by reordering a pair of basic blocks leads to performance penalty. Nevertheless, this method adds extra edges to the control flow graph, raising extra challenges to program similarity analysis which relies on program control structures.

Basic Block Splitting. This obfuscation method splits one basic block into two. We insert an extra `jmp` instruction between two adjacent instructions within a basic block b. The inserted `jmp` points to the second instruction, thus splitting block b with an additional control transfer. Similar to basic block reordering, our implementation gathers all functions with more than ten instructions. Then, for each function candidate, we randomly select an instruction and insert the `jmp` instruction after it. This obfuscation method brings in extra edges and nodes into the control flow graph and therefore, can reasonably defeat software similarity analyzers from adversaries.

Opaque Predicate Insertion. As mentioned in Fig. 2, opaque predicate introduces bogus branch conditions which expose high challenge for static reasoning, but will be always evaluated to "true" (or "false") during the runtime. For the implementation, we use a set of number-theoretic constructions (e.g., `(x*(x-1) % 2 == 0)`) as opaque predicate candidates. For each obfuscation pass, we first randomly select an opaque predicate from the number-theoretic construction set. We then randomly select a basic block within a function and insert the opaque predicate ahead of it.

The opaque predicate scheme can generate a considerable number of new blocks and edges on the program control flow. Hence, compared to basic block reordering/splitting, each pass can more fruitfully complicate the program control structures. Nevertheless, relatively higher cost can be introduced, given the complexity of the opaque predicts. In contrast to existing tools which let users decide to apply this scheme or not toward a particular program, we train a DRL model to make the decision.

Control Flow Flattening. As discussed in Sec. 2.1, this scheme flattens the control flow graph into a big "switch" statement. Two "dispatcher" blocks are deployed to redirect the execution flow, preserving the semantics of the original program. To implement this obfuscation pass, we randomly select one function each time and flatten its control flow graph. Note that we do not obfuscate a function if it uses indirect jumps. "Dispatcher" blocks hard-code destinations of each control transfer, and it is generally challenging to reason the control transfer destinations of x86 indirect jump instructions.

Our preliminary studies show that this obfuscation method can introduce considerable amount of extra cost; this is intuitive because this scheme largely complicates the control flow structure. For every control transfer in the original program, it is converted into three control transfers (entering, exiting, and jumping between two "dispatcher" nodes) in the flattened control structures.

Branch Function Insertion. This scheme identifies jump instructions and replaces them with function calls. Each function call instruction will call an artifact function, named "branch routine", to redirect the control transfer back to the destinations of the original jump instructions (the destination address is stored in a global variable and accessed by

the branch routine). To implement this obfuscation, each pass randomly picks $x\%$ jump instruction to obfuscate, where x is empirically decided as 2.5 for our current implementation.

The opaque predicate scheme reasonably complicates the control flow graph and call graph, by replacing jump instructions with function calls to a branch routine function. More importantly, our deliberately implemented routine function contain less than ten instructions, introducing small performance penalty. Indeed, our observation and evaluation shows that this obfuscation is highly likely to be selected by our RL model, given its high effectiveness and modest cost.

Function Reordering. Similarly to basic block reordering, performing function reordering swaps the relative positions of two functions in the program memory layout. For the implementation, we randomly select one pair of functions and swap their positions.

This method reasonably changes the program layout by swapping functions, but does not primarily introduce extra edges on program control structures. Our observation shows consistent findings: this method reduces the similarity of obfuscated programs without overwhelmingly lagging the program.

Function Inline. This obfuscation scheme inlines functions into their call-sites by changing call instructions into push and jump instructions to preserve the original semantics. At this step we only identify and inline direct call-sites. Hence, we conservatively preserve the inlined function at its original place. To implement this obfuscation method, one function is randomly selected each time to transform as long as its size is less than a threshold (the threshold is set as 500 bytes).

This method largely changes the program layout by using a relatively small function to extend its callers, which can adequately complicate the intra-procedural control flow graph of the caller functions and introduce extra nodes and edges in the obfuscated binary code. For each inlined function, we change its return instructions into jump instructions, whose destination is the instruction adjacent to the original callsite of the caller functions. Our observation shows that IDA-Pro can incorrectly treat the newly-created jump instructions as "function exit point", thus causing considerable errors in function boundary recovery. See our evaluation in Sec. 6.3.1 and Sec. 6.6.2.

Overall, instruction-level obfuscations perform a relatively lightweight transformation, while basic block and function-level obfuscations take higher-level control structures into account, thus imposing larger changes to the program. On the other hand, complicating program structures with basic block and function-level obfuscations can usually incur a higher performance penalty, thus can be undesired for some cases. To obfuscate a particular software, it is generally challenging to decide which methods in Table 1 to use and how to apply them. This research proposes to train a DRL model to rapidly explore the search space and decide an appropriate combination of obfuscation methods for each particular software.

4.2 Reward

The reward function is key to reinforcement learning frameworks in terms of formulating our learning goals: 1) keep the code appearance as dissimilar to the original program as possible; and 2) avoid introducing too much execution cost. As aforementioned, we measure the performance cost for each step, along with whether the obfuscated code can largely reduce program similarity. To this end, the reward function at each step is formalized as follows:

$$R = \begin{cases} 5.0 - \alpha \times \frac{perf_{obf} - perf_{orig}}{perf_{orig}}, & \text{if success} \\ -0.05 - \alpha \times \frac{perf_{obf} - perf_{orig}}{perf_{orig}}, & \text{if failure} \\ -0.05 - \beta \times \frac{perf_{obf}}{perf_{orig}} + \gamma \times \frac{1}{similarity} & \text{otherwise} \end{cases} \quad (1)$$

where the obfuscation cost is computed by measuring both the execution time of obfuscated code ($perf_{obf}$) and the original input ($perf_{orig}$) w.r.t. the same input. We penalize obfuscation transformations in case it leads to high execution time slowdown. We also measure the program similarity between the obfuscated result and the input ($similarity$). Low similarity score indicates good obfuscation result, and therefore leads to higher learning reward (see Sec. 6 for how performance and similarity are measured). We use a small negative value (-0.05 in our current setting) to penalty for each transformation. As a result, the model is progressively trained to find short obfuscation sequence.

We penalize obfuscations such that if the obfuscated software imposes a large execution overhead, we compute a low reward score for the agent. Additionally, the performance cost depends on the whole obfuscation sequence that has been applied so far rather than on the last obfuscation, namely, our formulation considers long-term rewards and progressively infers quality obfuscation sequences. For long-term reward harvesting, we use a discount rate of 1.0, and therefore, highlight the importance of the last learning step (where we terminate the episode and decide the "success" or "failure"). We adopt a dynamic learning rate initialized with 0.0001 and reduced to a half of itself every 20 episodes. We follow the common practice to adopt a ϵ-greedy policy with ϵ decayed from 1.0 to 0.01 (decay ratio 0.995). ϵ will be fixed at 0.01 thereafter. Hence, the training starts with balanced exploitation and exploration while gradually converging to optimal decisions. Overall, while we follow common and standard practice to decide these hyperparameters, evaluation results already report promising findings (see Sec. 6).

Termination. There are two conditions forcing the termination of an episode. We assign a large positive reward and terminate the current learning epoch if the obfuscated code can successfully reduce the similarity score to below T, implying the success of code obfuscation. We also define a maximal iteration number N on one episode; when this iteration is hit, it means that there should be few chances we can find an optimal obfuscation sequence during this episode. Hence, we terminate the current episode.

5 Implementation

We implement RLOBF, in total approximate 2,500 lines of Python code, to directly obfuscate binary executables. To

facilitate the verification of our results, we have uploaded a snapshot of our codebase to GitHub [10]. We will provide an official release version with detailed documentation on reproducing our results, once this paper is officially published.

RLOBF consists of two modules to 1) perform reverse engineering and executable file obfuscation (with about 1,500 lines of Python code), and 2) train DRL models to guide the obfuscation procedure and synthesize optimal obfuscation sequences (with about 1,000 lines of Python code). In the DRL module, we use Keras (version 2.3.1) with TensorFlow (version 2.0) as the backend to develop DRL models.

RLOBF is directly applicable to executable files. That is, RLOBF does not rely on any specific programming language and can directly protect closed-source software. The reverse engineering module is implemented based on a reverse engineering platform: Uroboros [27], which provides infrastructures for executable file disassembling and instrumentation. We implement the proposed nine obfuscation methods (see Table 1) as passes in Uroboros.

The prerequisite for binary code obfuscation and instrumentation is disassembly; obfuscation is performed to manipulate the disassembled output into a hardened representation. While precise disassembly is known to be hard in principle, current algorithms have been shown to perform very well in practice and to realize fool-proof disassembly of real-world complex software [28], [29], [30]. The Uroboros framework used by RLOBF implements an advanced disassembling algorithm which has been demonstrated to smoothly disassemble and instrument commonly-used Linux applications. Without reinventing the wheel, in this study, we reuse Uroboros and assume that reverse engineering is *reliable*.

Nevertheless, the reverse engineering platform is orthogonal to the design of RLOBF, and users can replace Uroboros with other popular reverse engineering platforms. For instance, one recent paper in this field, Ramblr [29] enhances this design and reports better results in a very effective way. We also note that the underlying disassembling infrastructure of Ramblr, `angr` [31], is very actively developed and maintained. `angr` also has a highly supportive community. We agree migrating RLOBF from Uroboros to Ramblr can presumably induce better implementation, conducting binary reverse engineering and instrumentation in a handy way.

Our current implementation of RLOBF, as a research prototype, is on the basis of Uroboros. We consider the current prototype *suffices* demonstrating the key research idea and can also smoothly transform `coreutils` programs on the Linux platform (both Uroboros and Ramblr evaluated `coreutils` programs in their papers). Also, we have already released RLOBF for the reference and to facilitate future research [10].

Overall, we consider it as an advantage to promote the practical usage of the presented technique by directly protecting executables. To our best knowledge, all non-trivial and actively-maintained obfuscation tools, including LLVM-Obfuscator [16], can only process program source code or particular intermediate representation (IR).

6 EVALUATION

6.1 Evaluation Setup

We first discuss the setup of our evaluation. To evaluate RLOBF, our evaluation dataset is derived from GNU `coreutils` (version 8.28), a software set commonly used by software security research.[1] This dataset consists of 106 programs which are the "must-have" utilities on Linux platforms. They provide diverse functionalities such as textual processing, crypto computation, and system management. We remove those programs with destructive semantics (e.g., `rm`) or relatively simple functionality (e.g, `true`), resulting in a set of 48 programs. Without loss of generality, in this research we randomly pick 20 programs from these 48 programs as the benchmark programs.

Learning Reward. As discussed in Sec. 4.2, we compute reward during each step of learning w.r.t. reducing program similarity and retaining low cost. Therefore, we seek for two types of rewards: *similarity* and *cost*. First, for similarity, we investigate how well the obfuscated software can mislead the de facto software similarity analyzer, BinDiff [32]. BinDiff is designed as a plugin of the popular commercial decompiler, IDA-Pro [33]. Bindiff takes two executable files as inputs and computes a similarity score to indicate how similar they are. BinDiff first leverages IDA-Pro to dissect each binary executable into functions and reconstruct the CFG of each function. Then, it performs graph isomorphism-based comparison [34] to measure the similarity of two functions w.r.t. their CFGs. The similarity of two executable files is derived by averaging the function-level similarity scores. Enabled by its advanced graph isomorphism comparison, BinDiff has been demonstrated to effectively capture the program structure-level similarity and widely used by cybersecurity analysts and attackers [23], [24], [25]. We feed BinDiff with obfuscated executables to compute the similarity score compared with the input executable. Second, for *cost*, we measure the performance of (obfuscated) executable files by using standard test cases shipped by GNU `coreutils` open source project.

Thresholds. To accurately define the reward of RLOBF, we use two thresholds, i.e., T, representing the similarity score of deciding the success of an obfuscation procedure, and N, denoting the longest obfuscation passes that are allowed to apply. To decide T, we use BinDiff to measure the similarity of every two programs from `coreutils` and the average similarity score between any two programs is 0.68. It indicates that, when the obfuscated program has a similarity score below 0.68 compared with the input, adversaries would reasonably treat them as "different" and presumably lose attack opportunities (e.g., code-reuse attack [9]). Hence, T is defined as 0.68 in our evaluation. In addition, we set N as 20 to limit the maximal number of obfuscation passes. Moreover, with sufficient training, RLOBF can synthesize optimal sequences with on average only 11.2 obfuscation passes.

6.2 Model Training

Our learning and testing were conducted on a server machine with an Intel Xeon E5-2683 v4 CPU at 2.40 GHz with

1. https://www.gnu.org/software/coreutils

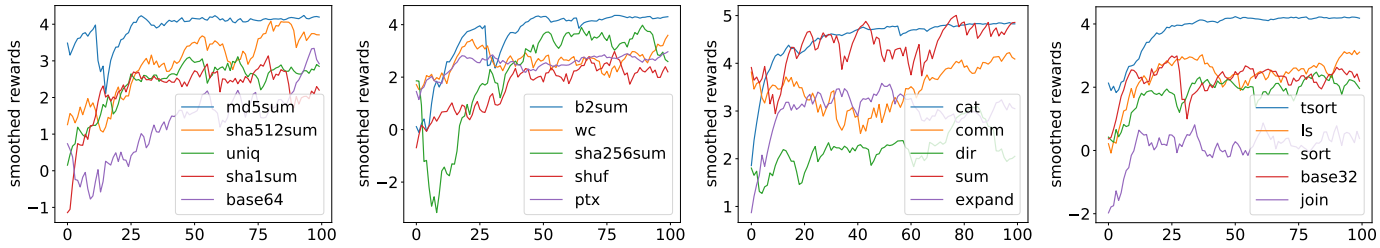


Fig. 4. Smoothed reward increases over episodes.

256 GB of memory and two NVIDIA RTX 2080 Graphics cards. The machine runs Ubuntu 18.04.

We train an RL model for each program binary code, and we set the maximal episode as 100. When training, we fine-tuned three parameters (α, γ, and β) used in defining the reward. Overall, we report the average range of performance ratio ($\frac{perf_{obf}}{perf_{orig}}$) falls within $[1.00, 10.83]$, and after applying β, this term is in the range of $[0.37, 4.06]$. The similarity ratio ($\frac{1}{similarity}$) is in the range of $[0.41, 0.86]$, and after applying γ, it is in the range of $[0.38, 0.79]$. We interpret that after applying the coefficient for smoothing, cost and similarity factors contribute comparably.

Our experiment takes approximate on average 11.2 hours for training one case. Fig. 4 reports the reward increase with respect to episodes. Overall, the evaluation results are promising. Test cases, despite their diverse functionality (e.g., `sha512sum` implements crypto hash algorithms while `wc` is a file processing program), all test cases manifest roughly consistent behaviors during training. The cumulative reward keeps increasing and the convergence happens around 60 episodes. Indeed, we report that with sufficient training, RL model can find well-performing obfuscation sequences for *all* the test cases.

6.3 Exploring Optimal Sequences

We collect the synthesized optimal obfuscation sequences for further studies. Fig. 5 reports the distribution of the average length of optimal sequences and the corresponding average slowdown on executables obfuscated by optimal sequences. For all the evaluated 20 programs, the average length of obfuscation sequences is 11.2 and the corresponding performance overhead is only 18.8%.

Our trained models successfully find well-performing obfuscation sequences in terms of similarity score and performance. As shown in the cumulative histogram in Fig. 5, 70.0% cases have less than 20.0% performance overhead, when applying the optimal obfuscation sequences. We also report that 75% optimal sequences contain more than 10 obfuscation passes, indicating a diverse sets of obfuscation traces are generated. The promising result demonstrates that the proposed technique synthesizes diverse and highly-efficient obfuscation sequences toward executable files.

Recall the implemented obfuscations can be put into three categories regarding levels of program structures they focus (cf. Table 1). To study the preference of well-trained models, we compute distribution in terms of these three categories from the produced optimal sequences in Table 2.

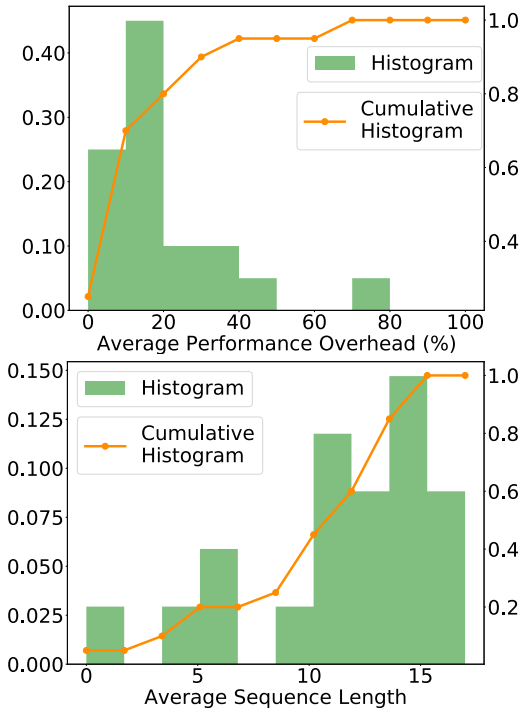


Fig. 5. Exploring optimal obfuscation sequences in terms of average cost and sequence length.

TABLE 2
Distribution of optimal sequences according to obfuscation category.

Instruction Level	Basic Block Level	Function Level
5.5%	83.2%	11.3%

The result shows that basic block-level obfuscations dominates the optimal sequences, because it can significantly obfuscate programs without introducing too much overhead. As introduced in Sec. 4.1, basic block-level transformations manifest a good balance between effectiveness and cost, by adding additional edges on the control graph to defeat the graph isomorphism-based similarity analysis in BinDiff, while does not overly change program structures. Function-level obfuscations, in particularly function inline, are also reasonably adopted given its effectiveness in complicating program control structures. In contrast, we find that control flow flattening, by largely flattening the control-flow graph of a function (see Fig. 2), overly imposing performance overhead and are usually excluded

from optimal sequences. Function reordering obfuscation is also undesired. As aforementioned, BinDiff computes the "similarity" of two binary executables, by first measuring the graph isomorphism-based similarities of functions. While function reordering obfuscation swaps the relative positions of two functions, it does *not* change the control-flow graph of each function, thus engendering trivial effect on BinDiff. Similarly, instruction-level obfuscation methods do not primarily change program structures, thus are rarely used in optimal obfuscation sequences.

6.3.1 Optimal Sequences Across Different Programs

We further investigate on how different obfuscation sequences are across different binaries. To this end, we study if there exist some common obfuscation sequences that are generally effective in most binaries in the dataset. Table 3 reports the top-three obfuscation sequences inducing the highest accumulated learning rewards for each test case. High accumulated rewards indicate low incurred overhead, and therefore, these top-three obfuscation sequences can be mostly desired by users. On the other hand, we also note that since almost all obfuscation passes randomly select code components for transformation (see Sec. 4.1), using the same sequence of obfuscations can still induce different obfuscated executable files. We now interpret Table 3 from the following aspects.

Overall, basic block-level obfuscation methods, in particularly `BFI` (branch function insertion), are frequently used in forming common obfuscation sequences. We interpret the results as highly consistent with our aforementioned distributions (i.e., basic block-level obfuscations occupy 83.2% optimal sequences). Basic block-level obfuscations generally outperform instruction- and function-level obfuscation methods given its balanced cost and effectiveness. In particular, `BFI`, by changing x86 jump instructions into function calls, exhibits noticeable effectiveness in defeating the similarity analysis of BinDiff. As previously mentioned, BinDiff conducts a graph isomorphism-based similarity analysis, where changes on the control flow graph and call graph would largely impede the analysis and reduce the similarity score. More importantly, `BFI` does not impose too much cost on the transformed binary code, hence is frequently picked by the RL agent when transforming most binary code. While `FIL` (function inline) is adopted by a number of programs (e.g., `sha512sum`, `base64`), this obfuscation method seems particularly effective to process `b2sum`, reducing the similarity score below 0.68 with only one pass. Our study shows that `FIL` inlines a frequently invoked function, `rotr64`, into its in total 384 callers. Also, after inlining this function, our method changes the function return opcode `ret` into `pop ecx; jmp ecx`, which can be mistakenly treated as function exit points by IDA-Pro. This would induce broken recovery of function boundary information, thus largely impeding BinDiff's graph isomorphism-based similarity analysis.

We note that while most optimal obfuscation sequences are distinct, we still observed some similar patterns. Note that while `coreutils` programs have generally distinguish functionalities, many `coreutils` programs are statically linked to a large `coreutils` library which subsumes some common functionality and utilities. Our observation shows that obfuscating functions from this static `coreutils` library could likely induce identical obfuscation sequences.

We also observed an outlier in Table 3. Optimal obfuscation sequences generated on `md5sum` seems inconsistent with others by frequently employing `OPA` (opaque predicate insertion) instead of `BFI`. This is anti-intuition, given that the inner workings of crypto hash algorithm `md5` and `sha1` are highly similar. Indeed, our observation on the `coreutils` implementation of `md5sum` and `sha1sum` show that they share mostly identical code blocks. At this step, we carefully analyzed and identified root causes inducing inconsistent obfuscation sequence generation on `md5sum` and `sha1sum`.

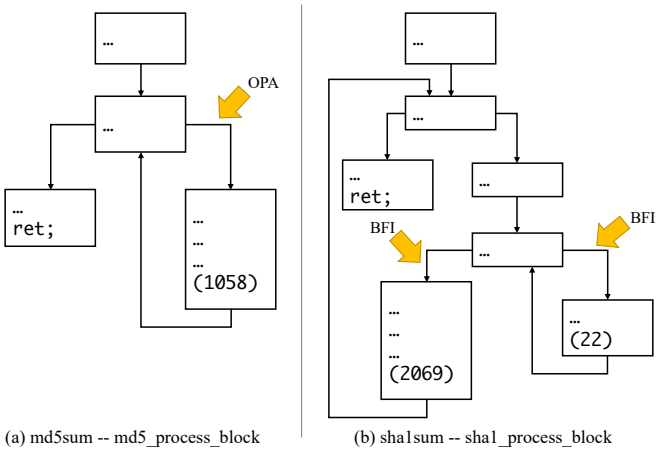


Fig. 6. Comparing obfuscation passes applied toward the `md5_process_block` function of `md5sum` vs. the `sha1_process_block` function of `sha1sum`.

Fig. 6 presents and compares the code components that induce the key difference of optimal sequences. Two code chunks in Fig. 6(a) and Fig. 6(b) represent the main loop of conducting hash computation in `md5sum` and `sha1sum`, respectively. In particular, `md5sum` leverages a loop, the bottom right basic block in Fig. 6(a) with 1058 instructions, to iteratively compute the crypto hash. In contrast, Fig. 6(b) shows that the corresponding hash loop of `sha1sum` is slightly more complex, in the sense that a large portion of loop iterations are undertaken by a small basic block with only 22 instructions. This notably reduces the execution cost of `sha1sum`, and as a result, RL model has to use relatively lightweight obfuscation methods for `sha1sum` to retain low performance penalty in the reward function (i.e., $-\frac{perf_{obf}-perf_{orig}}{perf_{orig}}$; see Eq. 1). Nevertheless, `md5sum` takes longer execution time (i.e., a larger $perf_{orig}$), and therefore, imposing relatively heavyweight obfuscation method (i.e., `OPA`) would thus incur less relative increase on the performance cost (since the divisor in $-\frac{perf_{obf}-perf_{orig}}{perf_{orig}}$ is already very large). Indeed, we report that the RL agent tends to translate the highlighted edge in Fig. 6(a) with `OPA`, while translating two highlighted edges in Fig. 6(b) with `BFI`.

6.3.2 Comparing with Randomly Constructed Sequences

To understand the quality of optimal sequences generated by RLOBF, we also compare the optimal sequences with randomly constructed sequences. To setup the comparison,

TABLE 3
Top 3 frequently-used optimal obfuscation sequences for each test case. The abbreviations are described in Table 1.

Programs	Optimal obfuscation sequences
b2sum	FIL FIL FIL
base32	OPA,BBS,BFI,BFI,BFI,BFI,BFI OPA,IR,GRA,IR,BFI,BFI,BFI,BFI OPA,IR,BBS,BFI,BFI
base64	CFA,CFA,BFI,BFI,BFI,BFI,BFI,FR,BFI,BFI,BFI,BFI,BFI,BFI,BFI CFA,OPA,CFA,BFI,BFI,BFI,BFI,BFI,BFI,BFI,BFI,BFI,BFI,BFI,BFI,BFI,BFI BBR,FIL,BFI,BFI,BFI,BFI,BFI
cat	CFA,IR,BFI,BFI,BFI,BFI,BFI,BFI,BFI,BFI CFA,IR,BFI,BFI CFA,IR,BFI
comm	OPA,IR,BFI OPA,IR,IR,BFI,BFI,BFI,BFI OPA,IR,IR,BFI,BFI,BFI,BFI,BFI,BFI
dir	GRA,BBS,BFI,BFI,BFI,BFI,BFI,BFI,BFI,BFI,BFI,BFI,BFI GRA,BBS,BFI,BFI,BFI,BFI,BFI,BFI GRA,BBS,BFI,BFI,BFI,BFI,BFI,BFI,BFI
expand	BBR,BFI,BFI,BFI,BFI,BFI IR,BBR,BFI,BFI IR,BBR,BFI,BFI
join	FIL,GRA,FIL,BFI,BFI,BFI,BBR,BFI,BFI,BFI,BFI FIL,GRA,FIL,BFI,BFI,BFI,BFI,BFI,BFI,BFI,BFI,BFI,BFI,BFI,BFI,BFI,BFI,BFI FIL,GRA,FIL,BFI,BFI,BFI,BFI,BFI,BFI,BFI,BFI,BFI,BFI,BFI,BFI
ls	CFA,BBS,BBS,BFI,BFI,FR,BFI,BFI,BFI CFA,CFA,BBS,BFI,BFI,BFI,BFI CFA,BBS,BBS,BFI,BFI,BFI,BFI,BFI,BFI,BFI
md5sum	BBR,OPA,OPA,OPA,OPA,OPA,OPA,OPA,OPA,OPA,OPA,OPA,OPA,OPA BBR,OPA,OPA,OPA,OPA,OPA,OPA,OPA,OPA,OPA,OPA,OPA,OPA,OPA BBR,OPA,OPA,OPA,OPA,OPA,BBS
ptx	GRA,BBS,BFI,BFI,BFI,BFI BBS,BBS,BFI,BFI,BFI,BFI BBS,BBS,BFI,BFI,BFI,BFI,BFI
sha1sum	GRA,CFA,BFI,IR,BFI,BFI,BFI,BFI OPA,BFI,BFI,IR,BFI,BFI,BFI,BFI,BFI,BFI OPA,CFA,IR,BFI,BBS,BFI,BFI
sha256sum	BBR,BBR,IR,BFI,BFI,BFI,BFI,BFI,BFI,BFI,BFI,BFI,BFI,BFI,BFI,BFI CFA,BFI,BFI,BFI,BFI,BFI,BFI,BFI,BFI,BFI,BFI,BFI BBR,BBR,CFA,BFI,BFI,BFI,BFI,BFI,BFI,BFI
sha512sum	BBR,FIL,FIL,FIL,FIL,FIL,FIL,FIL,FIL,FIL,FIL,FIL,FIL,FIL,FIL,FIL BBR,FIL,BFI,BFI,BFI,BFI,BFI,BFI,BFI,BFI,BFI,BFI,BFI,BFI,BFI,BFI,BFI,BFI,BFI,BFI BBR,BFI,BFI,BFI,BFI,BFI,BFI
shuf	CFA,BFI,BFI,CFA,BFI,BFI,BFI,BFI,BFI,BFI,BFI,BFI,BFI,BFI,BFI,BFI CFA,CFA,BFI,FR,BFI,BFI,BFI,BFI,BFI,BFI,BFI,BFI,BFI,BFI,BFI,BFI,BFI,BFI CFA,FR,CFA,BFI,BFI,BFI,BFI,BFI,BFI,BFI
sort	BFI,BFI,BBS,BFI,BFI,BFI,BFI,BFI,BFI,BFI,BFI,BFI,BFI BFI,BFI,IR,BFI,FR,BFI,BBS,BFI,BFI,BFI,BFI BFI,CFA,BFI,BFI,BFI,BFI,CFA,BFI,BFI,BFI,BFI,BFI,BFI,BFI,BFI
sum	CFA,BBR,CFA,OPA,OPA,BBR,OPA,OPA,OPA,BBR,BBR,BBR,BBR,BBR,BBR,BBR,BBR,BBR CFA,OPA,OPA,OPA,OPA,OPA,OPA,OPA,BBR,BBR,BBR,BBR,BBR CFA,OPA,BBR,BBR,BBR,OPA,OPA,BBR,BBR,BBR,BBR,BBR,BBR,BBR,BBR
tsort	IR,BFI,BFI,BFI,BFI IR,BFI,BFI,BFI IR,BFI,BFI
uniq	GRA,IR,BFI,BFI,BFI,BFI,BFI,BFI,BFI GRA,CFA,IR,BFI,BFI,BFI,BFI GRA,IR,BFI
wc	CFA,BBS,BFI,BFI,BFI,BFI,BFI,BFI,BFI,BFI,BFI,BFI,BFI,BFI,BFI CFA,BBS,BFI,BFI,BFI,BFI,BFI,BFI,BFI,BFI,BFI,BFI CFA,BBS,BFI,BFI,BFI

TABLE 4
Program CFG complexity before and after applying obfuscations.

Programs	# of Basic Blocks			# of CFG Edges			Cyclomatic Number		
	Original	Obfuscated	Increase (%)	Original	Obfuscated	Increase (%)	Original	Obfuscated	Increase (%)
b2sum	1385	4154	200.0	2993	8465	182.8	1610	4313	167.9
base32	1037	1372	32.4	1522	1935	27.1	487	564	15.7
base64	1006	1187	18.0	1470	1868	27.1	466	683	46.4
cat	882	961	9.0	1345	1749	30.0	465	789	69.9
comm	919	1011	10.1	1436	1934	34.7	519	925	78.1
dir	4309	5278	22.5	6890	8029	16.5	2583	2752	6.6
expand	889	1064	19.7	1247	1430	14.7	360	368	2.3
join	1227	1629	32.8	2007	12716	533.6	782	11088	1318.0
ls	4309	5604	30.1	6890	8217	19.3	2583	2614	1.2
md5sum	1059	1878	77.4	2378	6649	179.6	1321	4772	261.3
ptx	2138	2743	28.3	3344	3978	19.0	1208	1236	2.4
sha1sum	1062	1332	25.5	2382	2730	14.6	1322	1400	5.9
sha256sum	1070	1262	18.0	2390	2724	14.0	1322	1463	10.7
sha512sum	1076	1475	37.1	2401	6545	172.6	1327	5072	282.2
shuf	1548	1763	13.9	2560	2884	12.6	1014	1122	10.7
sort	3279	3841	17.2	5563	7026	26.3	2286	3186	39.4
sum	1079	2765	156.3	1526	5295	247.0	449	2531	463.8
tsort	877	962	9.8	1200	1680	40.0	325	719	121.2
uniq	1057	1152	9.0	1964	2340	19.2	909	1190	30.9
wc	1203	1519	26.3	2115	2546	20.4	914	1028	12.5
average	1570	2148	39.7	2681	4537	82.6	1112	2391	147.4

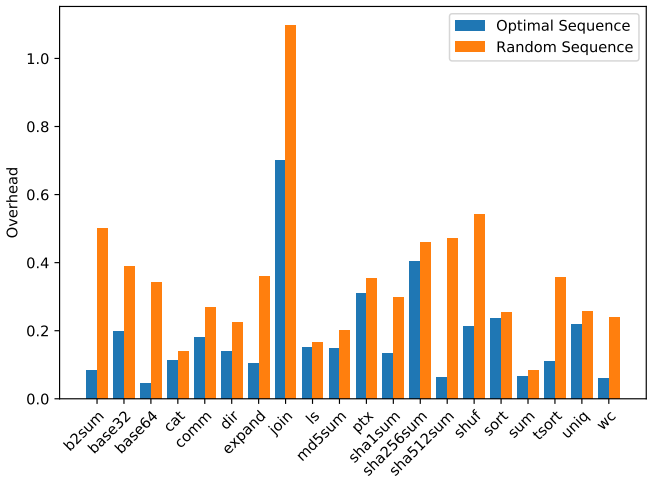


Fig. 7. Comparing overhead with randomly constructed sequences.

we randomly and iteratively apply obfuscation sequences toward an executable file until the similarity score reduces to the same threshold (i.e., 0.68). We then measure the performance cost induced by these randomly constructed obfuscation sequences, and compare with overhead incurred by optimal obfuscation sequences constructed by RLOBF. Fig. 7 reports the comparison results. Overall, RLOBF can consistently identify optimal obfuscation sequences to achieve the same amount of similarity reduction, while inducing much lower performance penalty. For instance, for `join`, we report that to achieve comparable reduction of similarity score, randomly constructed sequences introduces 1.6 times more execution slowdown. Even worse, for cases like `base64` and `sha512sum`, randomly constructed sequences induce about 7.6 times more overhead compared with obfuscation sequences generated by RLOBF. Overall, we interpret that this evaluation illustrates strong evidence that enabled by reinforcement learning techniques, RLOBF can promptly explore the search space and identify optimal obfuscation sequences, outperforming randomly constructed sequences where considerable runtime overhead could be introduced.

6.4 Complexity of Obfuscated Programs

To further study the complexity of our obfuscation result, we measure the complexity in terms of program control structure changes [26], [35], [36]. For this evaluation, we use three metrics to measure the program complexity: *Number of Basic Blocks*, *Number of CFG Edges*, and *Cyclomatic Number*. We write plugins for the commercial decompiler, IDA-Pro [37], to disassemble the executable files before and after using obfuscation and generate each corresponding control structure. Then, we traverse the control structures to record the number of basic blocks and edges for each program. Cyclomatic number [38] shows the overall complexity of CFG. It is defined as $e-n+2$, where e and n are the numbers of edges and basic blocks in the CFG, respectively.

Table 4 reports the CFG complexity evaluation. We measure programs obfuscated with synthesized optimal sequences, and its corresponding original inputs. We also report the increase percentage w.r.t. the original inputs. The obfuscated programs have largely outperformed their corresponding reference inputs, by achieving a much higher data in terms of each criterion. Obfuscated programs exhibit consistent increase of CFG complexity, compared to their original inputs. In particular, the improved basic block count can be up to 200.0% (for the `b2sum` case), with the average ratio is 39.7%. Similarly, `join` achieves the highest increase ratio in terms of cyclomatic number, and over half test cases have a cyclomatic number increase of 20%, indicating an encouraging increase of CFG complexity.

Table 4 shows that programs exhibit inconsistent increase in terms of CFG complexity. Despite the fact that the original program binaries have different number of basic blocks and edges, one key reason is that when analyzing the CFG of some obfuscated programs with IDA-Pro, due

to the applied obfuscation passes, IDA-Pro makes a large number of reverse engineering failures and *neglects* to count a number of basic blocks and edges in the obfuscated binary code. We give further discussion and quantitative assessment regarding this issue in Sec. 6.6.2 and Table 6.

6.5 Stealth of Obfuscated Programs

Besides complexity, *stealth* is another crucial property of obfuscated software. Stealth assesses the difficulty with which the obfuscation can be discovered. To evaluate stealth, we measure whether our obfuscation techniques introduce any abnormal statistical characters or code patterns that can be used to uncover obfuscation [12]. Particularly, we adopt a metric called instruction distribution [35], [39], which measures the ratio of different types of instructions. At this step, we group x86 instructions into 26 categories and calculate the distribution.

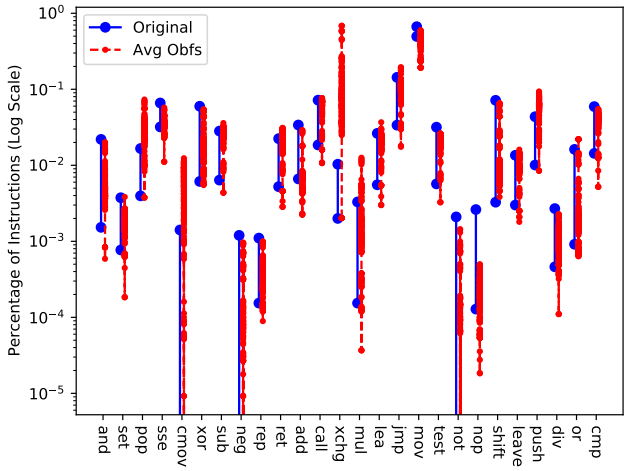


Fig. 8. Instruction distributions before and after obfuscations.

Fig. 8 reports the instruction distribution by taking all input programs and programs obfuscated with optimal sequences into account.[2] For each instruction category, "Original" shows the range of distributions in original programs, while each red dot denotes the distribution of a particular instruction category within an obfuscated program. In general, we interpret that obfuscated programs exhibit consistent trending with the input programs. In other words, obfuscated programs are stealth enough and attackers can hardly notice the usage of RLOBF.

A spike of `mul` (i.e., multiply) instructions can be found in the obfuscated programs. Similarly, `push` and `pop` instruction distributions are also increased after obfuscation. Accordingly, `mov` instructions have a decreased portion in the obfuscated programs. Overall, obfuscation reasonably increases certain instruction categories, for instance `mul` is employed to construct the opaque predicates, while `push` and `pop` instructions are frequently used to access new local variables. Similarly, the distribution of `xchg` is also increased, given its usage in garbage code insertion obfuscation (see Sec. 4.1). It is worth noting that C programs can be compiled into x86 assembly of *only* `mov` instructions [40]. As one future work, we envision the opportunity to further tune the distribution by substituting certain instructions (e.g., `mul` or `xchg`) with their semantics-equivalent `mov` instruction sequences.

2. Instruction distribution for each individual program can be found here [11]. Overall, all programs show consistent findings.

TABLE 5
ROP gadget elimination rate.

Programs	Elim. Rate%	Programs	Elim. Rate%
b2sum	93.4	base32	88.5
base64	88.6	cat	86.9
comm	87.0	dir	96.2
expand	87.4	join	88.9
ls	96.2	md5sum	87.9
ptx	91.5	sha1sum	88.9
sha256sum	90.7	sha512sum	94.0
shuf	90.5	sort	94.7
sum	89.5	tsort	85.4
uniq	89.1	wc	87.7

6.6 Security Impact

6.6.1 Resiliency to Code Reuse Attacks

To evaluate the security impact of our technique, we first measure its resiliency to code reuse attacks [26], [41], [42]. We evaluate how well our obfuscation technique can eliminate Return Oriented Programming (ROP) attack gadgets [9]. ROP attack is a de facto program exploitation which manipulates program call stacks and chains sequences of victim program's own code snippets (named ROP gadgets) to perform exploitations [9], [43]. To conduct such attacks, e.g., toward the Firefox browser on the victim's remote computer, the general procedure is to first extract memory addresses of ROP gadgets from a local Firefox browser and construct an attack payload, subsuming a sequence of ROP gadget addresses used to perform the attack. Then, the attacker sends this payload to the victim's computer and exploits the same Firefox broswer by executing each ROP gadget noted on the payload [26], [42], [44], [45], [46]. Software obfuscation helps to defeat ROP attacks by changing memory addresses of gadgets or breaking their instruction sequences in the obfuscated program, so that attackers cannot reuse these gadgets in their payload. Hence, the security impact of our obfuscation technique can be measured by counting how many ROP gadgets in the obfuscated code no longer stay in its original locations. We adopt a popular ROP gadget harvesting tool, ROPGadget [47], to search for ROP gadgets before and after our obfuscation.

Table 5 reports the ROP gadget elimination rate in our testbed. Let G_{orig} and G_{obf} denote two sets of ROP gadgets found from the input executable file and its corresponding obfuscated executable, respectively. Then, the ROP gadget elimination rate is defined as $1 - \frac{|G_{orig} \cap G_{obf}|}{\min\{|G_{orig}|, |G_{obf}|\}}$. The evaluation result is highly encouraging: ROP gadgets can be eliminated by up to 96.2% (the `dir` and `ls` cases), and at least 85.4% (the `tsort` case). The average elimination rate is 90.1%. In other words, less than 10.0% of the attack surface still remains, indicating a very low chance for attackers to exploit ROP gadgets in programs obfuscated by RLOBF.

Also, we note that some recent work has demonstrated the feasibility of conducting ROP attacks, without the preknowledge of ROP gadgets. In other words, ROP gadgets

TABLE 6
Obfuscation complexity imposed on C decompilation.

Programs	# of Functions			# of Unreachable Instructions			# of Decompilation Error		
	Original	Obfuscated	Increase (%)	Original	Obfuscated	Increase (%)	Original	Obfuscated	Increase (%)
b2sum	247	633.6	156.5	748	881.6	17.9	1	2.0	100.0
base32	210	288.6	37.4	757	2005.8	165.0	1	39.9	3890.0
base64	211	265.3	25.7	756	1529.2	102.3	1	11.1	1010.0
cat	196	224.5	14.5	675	1021.5	51.3	1	7.6	660.0
comm	195	223.6	14.7	700	1252.7	79.0	1	6.2	520.0
dir	575	647.1	12.5	1692	3088.2	82.5	1	112.0	11100.0
expand	196	266.5	36.0	661	1659.2	151.0	1	17.8	1680.0
join	231	298.7	29.3	679	2062.8	203.8	1	26.2	2520.0
ls	575	674.6	17.3	1692	3549.7	109.8	1	115.9	11490.0
md5sum	210	272.5	29.8	690	3336.8	383.6	1	1.8	80.0
ptx	284	353.9	24.6	905	2096.2	131.6	1	58.1	5710.0
sha1sum	210	284.9	35.7	690	2065.8	199.4	1	17.1	1610.0
sha256sum	214	281.0	31.3	788	2002.7	154.1	1	17.6	1660.0
sha512sum	214	293.5	37.1	788	25028.0	3076.1	2	18.8	840.0
shuf	305	403.7	32.4	1284	2843.4	121.4	1	40.0	3900.0
sort	528	621.0	17.6	1667	3183.7	91.0	1	53.4	5240.0
sum	212	283.4	33.7	756	2086.7	176.0	1	3.4	240.0
tsort	194	218.7	12.7	747	1235.8	65.4	1	6.5	550.0
uniq	220	264.8	20.4	721	1618.1	124.4	1	11.7	1070.0
wc	245	311.2	27.0	795	1641.6	106.5	1	41.9	4090.0
average	273.6	355.6	32.3	909.5	3209.5	279.6	1.1	30.5	2898.0

can be discovered from the remote victim software and chained to launch attacks on the fly [48], [49]. Nevertheless, we note that obfuscation is still the basis of advanced protection schemes against JIT-ROP [49], [50], [51], [52], [53], [54]. For instance, the mainstream defense method against JIT-ROP, Execute-no-Read (XnR) [52], [53], [54], changes the process memory space of protected software as non-readable but executable-only. Therefore, attackers can on longer inspect the protected process during runtime. To implement such scheme, the protected software needs to be obfuscated first, preventing attackers from resorting to use the pre-constructed ROP payloads.

6.6.2 Resiliency to C Decompilation

In addition to quantify the complexity of the obfuscated programs, we also explore the complexity and resiliency of obfuscation through the lens of adversarial reverse engineering. Overall, as one primary goal of software obfuscation is to impede hackers to conduct adversarial static analyses such as reverse engineering, we decide to use the IDA-Pro decompiler to decompile the obfuscated code and assess the quality of decompiled C code. For each `coreutils` test program, we randomly select five binary code obfuscated by optimal sequences, and further conduct decompilation with IDA-Pro.

A common decompilation evaluation metrics, named "structuredness" [55], counts the number of `goto` statements, to decide the complexity of the obfuscated programs. However, we note that when decompiling the binary code with our recent version of IDA-Pro (ver. 7.0), the number of `goto` statements in the C code seem to be very low, and does not primarily change with respect to obfuscation imposed. Instead, our finding shows that with obfuscation applied, IDA-Pro has primary difficulty of correctly identifying *function boundaries*, even if we do not trim off the symbol information from the binary code (i.e., we feed unstripped binary code to IDA-Pro).

We summarize two key observations when using IDA-Pro to decompile obfuscated binary code: 1) some instruction sequences, particularly control transfer targets of indirect jumps, does not belong to *any* function, and 2) IDA-Pro identifies much more "functions", by treating certain instruction sequences as function entry points in a mistaken way. We quantitatively measure and illustrate the first and second observations in Table 6. We also report the number of errors directly encountered when decompiling obfuscated code in the last three columns of Table 6. As shown in Table 6, obfuscation is indeed significant in impeding IDA-Pro's decompilation, given that IDA-Pro recovers average 32.3% more functions (only `BFI` would introduce one function for each pass), indicating that many instructions are mistakenly treated as "function exit points", incorrectly inducing new "function entry points" at the adjacent instructions. Similarly, the control transfer destinations of indirect jump instructions introduced by our obfuscation introduce more challenges in recovering intra-procedure control flow graphs. As a result, many instruction sequences that can be reached by only indirect jumps would not be taken into account when forming the control flow graph of obfuscated programs. We note that the difficulty of analyzing indirect jump targets indeed induces the underestimation of control flow complexity in the obfuscated code (see Table 4), since when counting the number of basic blocks and edges, we iterate every function recovered by IDA-Pro and then traverse its intra-procedural control flow graphs to count nodes and edges.

We collect errors thrown by IDA-Pro and report the average number of errors when decompiling an obfuscated binary code in Table 6; note that IDA-Pro typically would not

TABLE 7
Decompilation error messages.

Error message	#error in orig.	#error in obf.
"found positive sp value"	29	5008
"too big function"	1	12
"call analysis failed"	0	4946
"switch analysis failed"	0	8
"function frame is wrong"	0	2

"halt" or terminate the decompilation when ecountering errors, but would place a comment in the decompiled C code for each decompilation error. We compare error comments thrown by decompiling the original binary code and obfuscated binary code in Table 7. While it is generally difficult to analyze the root cause of these decompilation errors (IDA-Pro is a "blackbox" with no source code available), these error messages are mostly self-explainable. One major issue, "positive sp value", indicates defects in analyzing stack pointer register. Errors of this category likely occur during the local variable recovery stage. Another major issue, "call analysis failed", possible implies defects throw during the call graph recovery stage of IDA-Pro. As aforementioned, obfuscation methods introduce considerable indirect jump instructions, which presumably imposes high challenges of control flow analysis.

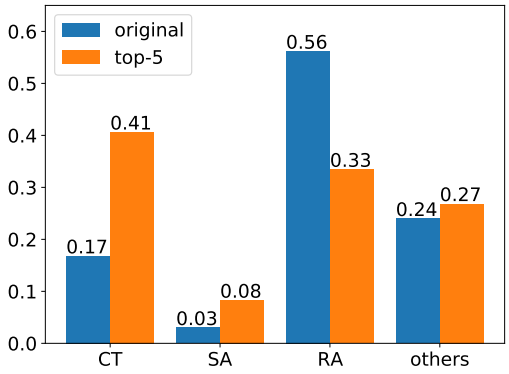


Fig. 9. Opcode type distribution in "top-5" contributors of model prediction and in original programs. For the ease of presentation, we classify x86 opcodes into four types regarding their holistic semantics. "CT" represents control transfer opcodes. "SA" subsumes opcodes for stack accesses. "RA" subsumes opcodes for register assignments. "Others" (e.g., arithmetic) subsume opcodes that are not related to control transfer, stack access, or register assignments.

6.7 Interpretation of Model Decisions

In this section, we conduct permutation feature importance analysis to interpret the decisions of well-trained DRL models [56]. Permutation feature importance estimates the importance of a given feature, by randomly shuffling the feature and measuring how much the model prediction can be influenced. Being agnostic toward the specific model implementations, this technique has been widely-used to "debug" black-box models. Here, we use a popular implementation of this technique, namely TextExplainer, to interpret the results of our trained models.[3]

TextExplainer takes a trained model and model inputs (in our case it is a sequence of opcodes) as its inputs. It calculates the importance of different opcodes w.r.t. decisions (i.e., selecting which obfuscation) made by the model. TextExplainer makes extensive permutation toward the opcode sequences, and our tentative test shows that if we feed opcode sequences of the entire program (typically around 20K opcodes in one input) into TextExplainer, it takes too long time to finish. Therefore, we instead feed function-level opcode sequences into TextExplainer. Accordingly, for this evaluation, we tweak RLOBF to let it take opcodes of a function for obfuscation.

3. https://eli5.readthedocs.io/en/latest/tutorials/black-box-text-classifiers.html

TABLE 8
Average performance overhead incurred by optimal obfuscation sequences under different similarity thresholds T. As reported in Sec. 6.1, $T = 0.68$ is our default setting for experiments.

Threshold T	**0.48**	**0.58**	**0.68**	**0.78**	**0.88**
`expand`	31.6%	21.6%	10.6%	10.1%	5.9%
`ls`	28.6%	27.8%	15.1%	10.2%	3.5%
`ptx`	65.8%	44.4%	31.0%	14.9%	12.6%
`sha1sum`	19.2%	13.9%	13.4%	6.0%	3.8%
average	36.3%	26.9%	17.5%	10.3%	6.3%

Recall given a sequence of opcodes, the trained model will select one obfuscation to use, aiming at achieving high effectiveness with low cost. TextExplainer assigns each opcode an importance score, implying their contribution to the prediction. To understand "top contributors", we feed opcode sequences of all functions (in total 3,360 functions from 20 cases) from our dataset, and collect the top-5 contributors to each prediction. We present the distribution in terms of opcode types in Fig. 9. To compare with, we also measure the distribution of opcode types within these function-level opcode sequences (on average each sequence has 67.2 opcodes).

As shown in Fig. 9, while opcode related to control transfers (e.g., `jmp`) has a relatively low portion in normal programs, they are critical in predictions. Opcodes related to control transfers primarily affect the program control structures, and since BinDiff performs graph isomorphism-based similarity analysis, a well-trained model would naturally focus on those control transfer opcodes to defeat BinDiff effectively. In contrast, while opcodes related to register-level operation (e.g., `mov` which moves data to a register) extensively exist in x86 assembly code, they contribute notably less in predictions. Another interesting finding is that, instructions in the "others" category have higher portion in top-5 contributors of model predictions. These instructions mainly perform arithmetic computation such as `add` or `sub`. With further investigation, we observe that many of these arithmetic opcodes are used to compute the opaque predicate conditions inserted by previous obfuscation passes (see Fig. 2 for an example of opaque predicts). When they are shuffled by TextExplainer, those opaque predicates may be relocated to other places, disabling the execution of certain code fragments while enabling the execution of junk code guarded by the opaque predicates. By largely affecting performance, "others" play an important role in the model decision. In summary, we interpret the model interpretation reports encouraging and intuitive observations: the trained model can pinpoint critical features on program control structures, thus effectively making program dissimilar.

6.8 Adjusting Similarity Threshold

For the prototype implementation of RLOBF, we decide to train the model from scratch to highlight the key contribution and novelty — synthesizing effective and low-cost obfuscation sequences with DRL. To do so, we follow a general assumption to simultaneously take code similarity and

execution slowdown as the learning reward. Our learning reward uses several parameters to weight contributions of code similarity and performance. As reported in Sec. 6.2, by applying the coefficient for smoothing, we make the cost and similarity factors contribute *comparably* in designing a generic learning reward.

Nevertheless, we also envision the opportunities to incorporate domain-specific considerations to fine-tune the learning reward and promote model training. For instance, to protect *security-sensitive* programs with high obfuscation strength, we could decrease our current similarity threshold 0.68 (see evaluation setup at Sec. 6.1). This way, we anticipate the RL agent can be trained to harvest the power of heavyweight obfuscation methods on protecting such security-sensitive programs, for which a relatively higher cost could be incurred. Similarly, to protect *cost-sensitive* software (e.g., games), we could increase the similarity threshold 0.68 (e.g., into 0.78) to stop applying obfuscation methods at an earlier stage.

Overall, envisioning the importance of fine-tuning parameters in real-world usage scenarios, this section conducts further evaluation to study how different similarity threshold can affect the synthesized optimal obfuscation sequences. As shown in Table 8, we decrease the similarity threshold from the default setting 0.68 to 0.58 and 0.48, emulating scenarios of protecting security-critical applications: users are seeking higher-level of protection and willing to trade more cost. Similarly, we increase the similarity threshold from 0.68 to 0.78 and 0.88, emulating scenarios of protecting cost-sensitive applications by trading obfuscation effectiveness for less performance penalty. We randomly select four `coreutils` programs from our program set and re-train RL agents for each test program under different settings. We record and report the average performance overhead incurred by optimal obfuscation sequences in Table 8. Overall, we interpret that all test cases exhibit consistent performance overhead in terms of different similarity threshold. For instance, for cost-sensitive scenarios, setting a higher similarity threshold (e.g., 0.78) can notably reduce about half extra performance overhead of `ptx` (from 31.0% to 14.9%). Similarly, for security-critical scenarios, our RL agent can find optimal obfuscation sequences to reduce similarity score from 0.68 to 0.48 by only trading 5.8% more performance overhead (from 13.4% to 19.2%). Also, we note that when similarity score is set as 0.48, a considerable number of optimal obfuscation sequences have over 15 passes, and some of them have even more than 20 passes. In contrast, all optimal obfuscation sequences synthesized under other similarity thresholds contain less than 20 passes. This is intuitive; aiming at lower similarity scores (e.g., 0.48) forces the RL agent to iteratively apply more obfuscation passes to the input program. In our released RLOBF, we make the similarity threshold as a configurable parameter, whose value can be decided by users under different real-world usage scenarios.

7 Discussion

Binary code similarity is taken to compute the learning reward of our RL model. To this end, we use BinDiff, an industrial strength binary similarity analysis tool to compute a program-wise similarity score for each learning iteration.

BinDiff conducts a graph isomorphism-based similarity analysis, which is shown as robust to certain level of program changes; the score is also efficient to compute. Nevertheless, we have observed the progressive development of binary similarity analysis techniques leveraging various semantics (in terms of data flow facts) features. Such semantics features are either computed from rigorous symbolic execution-based techniques [57], [58], or leverage advanced software embedding and graph neutral network models [59]. Nevertheless, we note that such advanced binary similarity techniques do *not* fit our scenario. Overall, such semantics-based techniques deem to be *obfuscation-resilience*, and hence would retain a close to 1.0 similarity score, no matter what kind of obfuscation is applied. As a result, the semantic similarity score cannot be used for calculating the reward in our method. Hence, we skip to leverage such advanced albeit more heavyweight semantics-based binary diffing tools in training RLOBF. We give further of these related research in Sec. 8.

8 Related Work

Code Obfuscation. Existing obfuscation techniques, in general, aim at transforming programs to impede either static or dynamic reverse engineering. To combat static reverse engineering, various techniques have been proposed and range from simply using XOR masks to obscure code snippets to heavily changing the program control-flow structures [26]. Similar to compiler optimizations, the scope of these obfuscation varies in granularity from single instructions to the entire program. We have introduced obfuscation schemes that complicate program control structures in Sec. 2.1. In addition to transformations toward program code section, data encoding techniques [3] translate program data sections into oblivious representations. Furthermore, malware authors frequently use obfuscation techniques, including various ad hoc transformations and (homemade) crypto algorithms, to pack their malware samples prior to distribution and to avoid detection by anti-malware scanners [7], [60].

To thwart dynamic reverse engineering such as symbolic and concolic execution, techniques have been proposed for either substantially inflating execution paths within the program or crafting constraints that are extremely difficult for an SMT solver to check [61], [62], [63]. Advanced obfuscation techniques are proposed to incorporate a process-level virtual machine that rewrites an input program into bytecode instructions that are customized for an interpreter [64], [65], [66], [67]. During runtime, the bytecode is emulated by a virtual machine interpreter. The obfuscated output can effectively hamper dynamic analysis with its mingled execution model and runtime system. These complex obfuscation methods usually either introduce high execution overhead or involve much manual work. In this work, we focus on the obfuscation methods which are easily automated and introduce measurable execution overhead.

DNN-based Software Comprehension. DNN has enabled major thrust in various areas such as machine translation. Recent research has shown its ability in software comprehension and analysis, enabled by a various program em-

bedding techniques [68], [69], [70]. To date, RNN and LSTM techniques have been used to address various code comprehension and instrumentation tasks, including program synthesis [71], [72], [73], software reverse engineering [74], loop invariant inference [75], malware detection [68], program repairing [76], [77], program interpretation [78], and for education purposes, such as automatic grading of programming assignments [79], [80]. Despite the progress adoption of learning-based technique in understanding software, however, hardening software with learning-based techniques is rarely touched to date. In this work, we are the first to propose a unified learning framework to generate high-quality and low-cost software obfuscation sequences to instrument commonly-used software and make it more resilient toward (adversarial) similarity analysis.

Binary Similarity. Comparing the similarity or difference between two binary executable files is one of the most popular scenarios in software security, e.g., detection of malware variants, plagiarism detection, differential analysis. As two widely used binary diffing tools, BinDiff [32] and DarunGrim [81] first check the isomorphism of the control flow graphs of binary files and then compare the syntactic similarity of basic blocks. Binslayer [82] promotes BindDiff by comparing program bipartite graph. discovRE [83] accelerates the control flow graph isomorphism mapping by extracting syntactical features. These binary analysis tools only compare syntax-level similarity, that is, the appearance of a program. Therefore, tools like BinDiff perfectly fit our reinforcement learning training process. It gives feedback showing how different the obfuscation result is.

Binary diffing analysis can also perform semantics-level comparison. BinHunt [57] introduces symbolic execution and SMT solver to checking the equivalence of basic blocks. BinJuice [84] compares an abstract semantics extracted from basic blocks. iBinHunt [58] extends BinHunt to an interprocedure analysis by application of multi-tag taint analysis. Binsim [85] compares similarity of two dynamic execution traces by checking system call sliced segment equivalence. Recent research works also explore the feasibility to leverage AI techniques in comprehending program semantics and performing binary code similarity analysis [59], [86], [87], [88], [89], [90], [91], [92]. Nevertheless, semantic-level binary similarity analysis does not fit our scenario since our obfuscation result does not change program semantics.

9 Conclusion

Software obfuscation techniques have been widely used to protect software from malicious analysis. To date, how to effectively compose obfuscation methods still remains an unsolved problem. In this research, we propose the design of RLObf for synthesizing practical and high-quality obfuscation sequences to reduce program similarity with only low cost. The proposed technique harnesses advanced DRL technique to learn optimal obfuscation sequences. We implemented the proposed technique as a practical tool to protect executable files. Our evaluation reports promising results. The employed DRL model can transform programs to diverse representations with modest cost.

References

[1] J. Nagra and C. Collberg, *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*. Pearson Education, 2009.

[2] C. Collberg, C. Thomborson, and D. Low, "Manufacturing cheap, resilient, and stealthy opaque constructs," in *Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, ser. POPL '98, 1998.

[3] C. Collberg and C. Thomborson, "Watermarking, tamper-proofing, and obfuscation—tools for software protection," *IEEE Transactions on Software Engineering*, vol. 28, no. 8, pp. 735–746, Aug. 2002.

[4] B. Barak, S. Garg, Y. T. Kalai, O. Paneth, and A. Sahai, "Protecting obfuscation against algebraic attacks," in *Proceedings of the 33rd Annual International Conference on the Theory and Applications of Cryptographic Techniques*, ser. EUROCRYPT '14, 2014, pp. 221–238.

[5] K. A. Roundy and B. P. Miller, "Binary-code Obfuscations in Prevalent Packer Tools," *ACM Computing Surveys*, vol. 46, no. 1, 2013.

[6] P. OKane, S. Sezer, and K. McLaughlin, "Obfuscation: The hidden malware," *IEEE Security and Privacy*, vol. 9, no. 5, 2011.

[7] M. Sikorski and A. Honig, *Practical Malware Analysis: The Hands-On Guide to Dissecting Malicious Software*. No Starch Press, 2012.

[8] C. Wang, J. Davidson, J. Hill, and J. Knight, "Protection of software-based survivability mechanisms," in *Proceedings of International Conference of Dependable Systems and Networks (DSN'01)*, 2001.

[9] H. Shacham, "The geometry of innocent flesh on the bone: Return-into-libc without function calls (on the x86)," in *CCS 2007*, 2007, pp. 552–561.

[10] "Snapshot of our codebase," https://github.com/whj0401/RLOBF, 2020.

[11] "Evaluation data," https://www.dropbox.com/sh/82203fmbxk1cpzo/AACsWk11j_-7oAuykfj7f8LZa?dl=0, 2020.

[12] C. Collberg, C. Thomborson, and D. Low, "A taxonomy of obfuscating transformations," Department of Computer Science, The University of Auckland, New Zealand, Tech. Rep., 1997.

[13] C. Linn and S. Debray, "Obfuscation of executable code to improve resistance to static disassembly," in *Proceedings of the 10th ACM Conference on Computer and Communications Security*, ser. CCS '03, 2003, pp. 290–299.

[14] A. Moser, C. Kruegel, and E. Kirda, "Limits of static analysis for malware detection," in *Proceedings of the 23rd Annual Computer Security Applications Conference*, ser. ACSAC '07, 2007.

[15] I. V. Popov, S. K. Debray, and G. R. Andrews, "Binary obfuscation using signals," in *Proceedings of 16th USENIX Security Symposium*, ser. USENIX Security '07, 2007.

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

[17] C. Szepesvári, "Algorithms for reinforcement learning," *Synthesis lectures on artificial intelligence and machine learning*, vol. 4, no. 1, pp. 1–103, 2010.

[18] C. J. C. H. Watkins and P. Dayan, "Q-learning," *Machine Learning*, vol. 8, no. 3, pp. 279–292, May 1992.

[19] V. Mnih, K. Kavukcuoglu, D. Silver, A. A. Rusu, J. Veness, M. G. Bellemare, A. Graves, M. Riedmiller, A. K. Fidjeland, G. Ostrovski *et al.*, "Human-level control through deep reinforcement learning," *Nature*, vol. 518, no. 7540, p. 529, 2015.

[20] D. Silver, A. Huang, C. J. Maddison, A. Guez, L. Sifre, G. Van Den Driessche, J. Schrittwieser, I. Antonoglou, V. Panneershelvam, M. Lanctot *et al.*, "Mastering the game of go with deep neural networks and tree search," *nature*, vol. 529, no. 7587, p. 484, 2016.

[21] V. Mnih, K. Kavukcuoglu, D. Silver, A. Graves, I. Antonoglou, D. Wierstra, and M. Riedmiller, "Playing atari with deep reinforcement learning," *arXiv preprint arXiv:1312.5602*, 2013.

[22] S. Wang, P. Wang, and D. Wu, "Composite software diversification," in *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 2017, pp. 284–294.

[23] U. Bayer, P. M. Comparetti, C. Hlauschek, C. Kruegel, and E. Kirda, "Scalable, behavior-based malware clustering," in *Proceedings of the 2009 Network and Distributed System Security Symposium*, ser. NDSS '09. Internet Society, 2009.

[24] J. Jang, M. Woo, and D. Brumley, "Towards automatic software lineage inference," in *Proceedings of the 22Nd USENIX Conference on Security*, ser. USENIX Security'13. USENIX Association, 2013, pp. 81–96.

[25] D. Brumley, P. Poosankam, D. Song, and J. Zheng, "Automatic patch-based exploit generation is possible: Techniques and implications," in *IEEE S&P'08*, 2008.

[26] P. Larsen, A. Homescu, S. Brunthaler, and M. Franz, "SoK: Automated software diversity," in *Proceedings of the 35th IEEE Symposium on Security and Privacy (S&P '14)*, 2014.

[27] M. Piano, "Infrastructure for Reassembleable Disassembling and Transformation," https://github.com/piax93/uroboros, 2018.

[28] D. W. Shuai Wang, Pei Wang, "A real disassembler," 2015.

[29] R. Wang, Y. Shoshitaishvili, A. Bianchi, A. Machiry, J. Grosen, P. Grosen, C. Kruegel, and G. Vigna, "Ramblr: Making reassembly great again," in *24th Annual Network & Distributed System Security Symposium*, 2017.

[30] B. Erick, L. Zhiqiang, and H. K. W., "Superset disassembly: Statically rewriting x86 binaries without heuristics," in *NDSS*, 2018.

[31] Y. Shoshitaishvili, R. Wang, C. Salls, N. Stephens, M. Polino, A. Dutcher, J. Grosen, S. Feng, C. Hauser, C. Kruegel, and G. Vigna, "Sok: (state of) the art of war: Offensive techniques in binary analysis," 2016.

[32] Zynamics, "BinDiff," http://www.zynamics.com/bindiff.html, 2018.

[33] S. Hex-Rays, "IDA Pro: a cross-platform multi-processor disassembler and debugger," 2014.

[34] T. Dullien and R. Rolles, "Graph-based comparison of executable objects," *SSTIC*, vol. 5, pp. 1–3, 2005.

[35] H. Chen, L. Yuan, X. Wu, B. Zang, B. Huang, and P.-c. Yew, "Control flow obfuscation with information flow tracking," in *Proceedings of the 42nd Annual IEEE/ACM International Symposium on Microarchitecture*, ser. MICRO 42. New York, NY, USA: ACM, 2009, pp. 391–400.

[36] P. Wang, S. Wang, J. Ming, Y. Jiang, and D. Wu, "Translingual obfuscation," in *2016 IEEE European Symposium on Security and Privacy (EuroS&P)*. IEEE, 2016, pp. 128–144.

[37] "Hex-Rays Decompiler: Manual," https://www.hex-rays.com/products/decompiler/manual/failures.shtml.

[38] T. J. McCabe, "A complexity measure," *IEEE Trans. Softw. Eng.*, vol. 2, no. 4, pp. 308–320, Jul. 1976.

[39] I. V. Popov, S. K. Debray, and G. R. Andrews, "Binary obfuscation using signals," in *Proceedings of 16th USENIX Security Symposium on USENIX Security Symposium*. Berkeley, CA, USA: USENIX Association, 2007, pp. 19:1–19:16.

[40] xoreaxeaxeax, "The single instruction C compiler," https://github.com/xoreaxeaxeax/movfuscator, 2018.

[41] V. Pappas, M. Polychronakis, and A. D. Keromytis, "Practical software diversification using in-place code randomization," in *Moving Target Defense II*. Springer, 2013, pp. 175–202.

[42] ——, "Smashing the gadgets: Hindering return-oriented programming using in-place code randomization," in *Security and Privacy (S&P), 2012 IEEE Symposium on*. IEEE, 2012, pp. 601–615.

[43] E. Buchanan, R. Roemer, H. Shacham, and S. Savage, "When good instructions go bad: generalizing return-oriented programming to RISC," in *Proceedings of the 15th ACM conference on Computer and Communications Security*, 2008, pp. 27–38.

[44] R. Wartell, V. Mohan, K. W. Hamlen, and Z. Lin, "Binary stirring: Self-randomizing instruction addresses of legacy x86 binary code," in *Proceedings of the 19th ACM Conference on Computer and Communications Security (CCS '12)*, 2012.

[45] C. Kil, J. Jim, C. Bookholt, J. Xu, and P. Ning, "Address space layout permutation (aslp): Towards fine-grained randomization of commodity software," in *Computer Security Applications Conference, 2006. ACSAC '06. 22nd Annual*, Dec 2006, pp. 339–348.

[46] L. V. Davi, A. Dmitrienko, S. Nürnberger, and A.-R. Sadeghi, "Gadge me if you can: Secure and efficient ad-hoc instruction-level randomization for x86 and arm," in *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security*, ser. ASIA CCS '13. New York, NY, USA: ACM, 2013, pp. 299–310.

[47] J. Salwan, "ROPgadget tool," *http://shell-storm.org/project/ROPgadget*, 2012.

[48] K. Z. Snow, F. Monrose, L. Davi, A. Dmitrienko, C. Liebchen, and A.-R. Sadeghi, "Just-In-Time Code Reuse: On the effectiveness of fine-grained address space layout randomization," in *Proceedings of the 2013 IEEE Symposium on Security and Privacy*, ser. SP '13. Washington, DC, USA: IEEE Computer Society, 2013, pp. 574–588.

[49] G. Maisuradze, M. Backes, and C. Rossow, "What cannot be read, cannot be leveraged? revisiting assumptions of JIT-ROP defenses," ser. USENIX Security '16, 2016, pp. 139–156.

[50] M. Backes and S. Nürnberger, "Oxymoron: Making fine-grained memory randomization practical by allowing code sharing," in *Proceedings of the 23rd USENIX Security Symposium (USENIX Security 14)*. San Diego, CA: USENIX Association, Aug. 2014, pp. 433–447.

[51] L. Davi, C. Liebchen, A.-R. Sadeghi, K. Z.Snow, and F. Monrose, "Isomeron: Code randomization resilient to (Just-In-Time) Return-Oriented Programming," in *22nd Annual Network & Distributed System Security Symposium (NDSS)*, 2015.

[52] M. Backes, T. Holz, B. Kollenda, P. Koppe, S. Nürnberger, and J. Pewny, "You can run but you can't read: Preventing disclosure exploits in executable code," in *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '14. New York, NY, USA: ACM, 2014, pp. 1342–1353.

[53] S. Crane, C. Liebchen, A. Homescu, L. Davi, P. Larsen, A. R. Sadeghi, S. Brunthaler, and M. Franz, "Readactor: Practical code randomization resilient to memory disclosure," ser. IEEE S&P '15, 2015, pp. 763–780.

[54] S. J. Crane, S. Volckaert, F. Schuster, C. Liebchen, P. Larsen, L. Davi, A.-R. Sadeghi, T. Holz, B. De Sutter, and M. Franz, "It's a trap: Table randomization and protection against function-reuse attacks," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*, 2015, pp. 243–255.

[55] E. J. Schwartz, J. Lee, M. Woo, and D. Brumley, "Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring," in *Proceedings of the 22nd USENIX Security Symposium (Washington DC, USA, 2013), USENIX Association*, 2013.

[56] L. Breiman, "Random forests," *Machine Learning*, vol. 45, no. 1, pp. 5–32, Oct 2001.

[57] D. Gao, M. K. Reiter, and D. Song, "BinHunt: Automatically finding semantic differences in binary programs," in *Proceedings of the 4th International Conference on Information and Communications Security*, ser. ICICS '08, 2008.

[58] J. Ming, M. Pan, and D. Gao, "iBinHunt: Binary hunting with inter-procedural control flow," in *Proceedings of the 15th Annual International Conference on Information Security and Cryptology*, ser. ICISC '12, 2012.

[59] S. H. Ding, B. M. Fung, and P. Charland, "Asm2Vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization," in *IEEE Symposium on Security and Privacy*, 2019.

[60] P. Szor, *The Art of Computer Virus Research and Defense*. Addison-Wesley Professional, February 2005.

[61] Z. Wang, J. Ming, C. Jia, and D. Gao, "Linear obfuscation to combat symbolic execution," in *European Symposium on Research in Computer Security*. Springer, 2011, pp. 210–226.

[62] S. Banescu, C. Collberg, V. Ganesh, Z. Newsham, and A. Pretschner, "Code obfuscation against symbolic execution attacks," in *Proceedings of the 32nd Annual Conference on Computer Security Applications*. ACM, 2016, pp. 189–200.

[63] B. Yadegari and S. Debray, "Symbolic execution of obfuscated code," in *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security*. ACM, 2015, pp. 732–744.

[64] R. Rolles, "Unpacking virtualization obfuscators," in *Proceedings of the 3rd USENIX Conference on Offensive Technologies*, ser. WOOT'09, 2009.

[65] O. Tech., "Code Virtualizer: Total obfuscation against reverse engineering," http://oreans.com/codevirtualizer.php, 2019.

[66] VMProtect, "VMProtect software protection," http://vmpsoft.com, 2019.

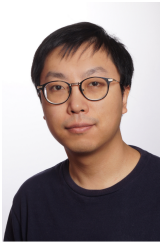
[67] C. Collberg, "The Tigress C Diversifier/Obfuscator," https://tigress.wtf, 2020.

[68] T. Ben-Nun, A. S. Jakobovits, and T. Hoefler, "Neural code comprehension: A learnable representation of code semantics," in *Advances in Neural Information Processing Systems*, 2018, pp. 3585–3597.

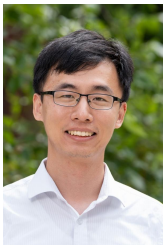
[69] J. Henkel, S. K. Lahiri, B. Liblit, and T. Reps, "Code vectors: understanding programs through embedded abstracted symbolic traces," in *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. ACM, 2018, pp. 163–174.

[70] E. Patterson, I. Baldini, A. Mojsilovic, and K. R. Varshney, "Teaching machines to understand data science code by semantic enrichment of dataflow graphs," *arXiv preprint arXiv:1807.05691*, 2018.
[71] M. Balog, A. L. Gaunt, M. Brockschmidt, S. Nowozin, and D. Tarlow, "Deepcoder: Learning to write programs," in *Proceedings of 4th International Conference on Learning Representations*, 2016.
[72] C. Liang, J. Berant, Q. Le, K. D. Forbus, and N. Lao, "Neural symbolic machines: Learning semantic parsers on freebase with weak supervision," in *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, vol. 1, 2017, pp. 23–33.
[73] E. Parisotto, A. rahman Mohamed, R. Singh, L. Li, D. Zhou, and P. Kohli, "Neuro-symbolic program synthesis," 2016.
[74] E. C. R. Shin, D. Song, and R. Moazzezi, "Recognizing functions in binaries with neural networks," in *In Proceedings of the 24th USENIX Security Symposium*, 2015, pp. 611–626.
[75] X. Si, H. Dai, M. Raghothaman, M. Naik, and L. Song, "Learning loop invariants for program verification." in *Advances in Neural Information Processing Systems (NeurIPS)*, 2018.
[76] K. Wang, R. Singh, and Z. Su, "Dynamic neural program embeddings for program repair," in *6th International Conference on Learning Representations*, 2018.
[77] R. Gupta, S. Pal, A. Kanade, and S. Shevade, "Deepfix: Fixing common c language errors by deep learning," in *Thirty-First AAAI Conference on Artificial Intelligence*, 2017.
[78] L. Zhang, G. Rosenblatt, E. Fetaya, R. Liao, W. E. Byrd, M. Might, R. Urtasun, and R. Zemel, "Neural guided constraint logic programming for program synthesis," 2018.
[79] S. Bhatia, P. Kohli, and R. Singh, "Neuro-symbolic program corrector for introductory programming assignments," in *Proceedings of the 40th International Conference on Software Engineering*, ser. ICSE '18, 2018, pp. 60–70.
[80] R. Gupta, A. Kanade, and S. Shevade, "Deep reinforcement learning for syntactic error repair in student programs," in *Proceedings of the thirty-third AAAI conference on Artificial Intelligence*, ser. AAAI 2019, 2019.
[81] "DarunGrim: A patch analysis and binary diffing tool," http://www.darungrim.org/.
[82] M. Bourquin, A. King, and E. Robbins, "Binslayer: Accurate comparison of binary executables," in *Proceedings of the 2nd ACM SIGPLAN Program Protection and Reverse Engineering Workshop (PPREW '13)*, 2013.
[83] S. Eschweiler, K. Yakdan, and E. Gerhards-Padilla, "discovRE: Efficient cross-architecture identification of bugs in binary code," in *Proceedings of the 23nd Annual Network and Distributed System Security Symposium (NDSS'16)*, 2016.
[84] A. Lakhotia, M. D. Preda, and R. Giacobazzi, "Fast location of similar code fragments using semantic 'juice'," in *PPREW'13*.
[85] J. Ming, D. Xu, Y. Jiang, and D. Wu, "BinSim: Trace-based Semantic Binary Diffing via System Call Sliced Segment Equivalence Checking," in *Proceedings of the 26th USENIX Conference on Security Symposium (USENIX Security'17)*, 2017.
[86] X. Xu, C. Liu, Q. Feng, H. Yin, L. Song, and D. Song, "Neural network-based graph embedding for cross-platform binary code similarity detection," in *Proceedings of the 2017 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '17. ACM, 2017, pp. 363–376.
[87] F. Zuo, X. Li, P. Young, L. Luo, Q. Zeng, and Z. Zhang, "Neural machine translation inspired binary code similarity comparison beyond function pairs," in *Proceedings of the 2019 Network and Distributed Systems Security Symposium (NDSS)*, 2019.
[88] U. Alon, M. Zilberstein, O. Levy, and E. Yahav, "Code2Vec: Learning distributed representations of code," *Proc. ACM Program. Lang.*, vol. 3, no. POPL, pp. 40:1–40:29, Jan. 2019.
[89] B. Liu, W. Huo, C. Zhang, W. Li, F. Li, A. Piao, and W. Zou, "αdiff: Cross-version binary code similarity detection with DNN," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE 2018, 2018, pp. 667–678.
[90] J. Gao, X. Yang, Y. Fu, Y. Jiang, and J. Sun, "Vulseeker: A semantic learning based vulnerability seeker for cross-platform binary," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE 2018, 2018, pp. 896–899.
[91] Y. Duan, X. Li, J. Wang, and H. Yin, "Deepbindiff: Learning program-wide code representations for binary diffing," 2020.
[92] Q. Feng, R. Zhou, C. Xu, Y. Cheng, B. Testa, and H. Yin, "Scalable graph-based bug search for firmware images," in *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '16, 2016, pp. 480–491.

Huaijin Wang received his Bachelor's degree in Software Engineering from the Nanjing University, Jiangsu, China, in 2018. He is currently a Ph.D. student at the Department of Computer Science and Engineering at the Hong Kong University of Science and Technology, Hong Kong SAR. His research interests include software security, reverse engineering, and program analysis.



Shuai Wang received the Ph.D. degree in informatics from the Pennsylvania State University, State College, PA, USA, in 2018. He is an Assistant Professor in the Department of Computer Science and Engineering at the Hong Kong University of Science and Technology, Hong Kong SAR. His research interests include cybersecurity and software engineering. He is a member of the IEEE.



Dongpeng Xu received his Ph.D. in Information Sciences and Technology at the Pennsylvania State University. He is an assistant professor in the Department of Computer Science at the University of New Hampshire. His research interest is software security, especially program analysis on binary code, malware analysis and detection, program protection, software testing, program similarity analysis, and model checking.

Xiangyu Zhang is a PhD student at University of New Hampshire. He received his M.S. degree in Electrical and Computer Engineering from Auburn University and B.E. in Electronic Information Engineering from Taiyuan University of Technology. His research interests are Cybersecurity, wireless networking, mobile social computing, crowd-sourcing, and machine learning.

Xiao Liu received her Ph.D. degree in informatics from the Pennsylvania State University, State College, PA, USA, in 2019. She is a Research Scientist at the Facebook Inc., Menlo Park, CA, USA. Her research interests lie in the areas of computer security, software engineering, and AI.