Data Augmentation in Learning-Based Binary Code Analysis

Lennart von Stülpnagel*
BIFOLD & TU Berlin
Berlin, Germany

Felix Weissberg*
BIFOLD & TU Berlin
Berlin, Germany

Jonas Möller
BIFOLD & TU Berlin
Berlin, Germany

Benjamin Greb
BIFOLD & TU Berlin
Berlin, Germany

Thorsten Eisenhofer
CISPA Helmholtz Center for
Information Security
Saarbrücken, Germany

Konrad Rieck
BIFOLD & TU Berlin
Berlin, Germany

Abstract

Understanding and analyzing binary code is fundamental to many security-critical tasks, such as reverse engineering and vulnerability discovery. As machine learning has become integral to these tasks, techniques from other domains have been transferred to binary code analysis. In particular, inspired by computer vision, data augmentation has recently been proposed to improve performance in code analysis tasks. However, prior work considers only a limited set of transformations and downstream tasks, making it difficult to draw broader conclusions about when and how augmentation is effective. In this work, we thus take a step back and systematically study the role of data augmentation in binary code analysis. To this end, we introduce a unified framework that encompasses both semantic-preserving and semantic-altering code transformations for augmentation. We evaluate the impact of augmentation across multiple tasks, including function similarity, parameter analysis, and vulnerability discovery. Our findings reveal a key difference from computer vision: in binary code analysis, augmentation is not a silver bullet. While code transformations can improve performance when training data is scarce, the gains diminish as more data becomes available. We conclude that augmentation is only beneficial under specific conditions and therefore derive practical recommendations for its application.

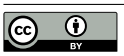
CCS Concepts

• **Security and privacy → Software and application security**; **Software reverse engineering**.

Keywords

Binary code analysis, data augmentation, machine learning

ACM Reference Format:
Lennart von Stülpnagel, Felix Weissberg, Jonas Möller, Benjamin Greb, Thorsten Eisenhofer, and Konrad Rieck . 2026. Data Augmentation in Learning-Based Binary Code Analysis. In *19th Workshop on Artificial Intelligence and Security (AISec '26), November 15–19, 2026, The Hague, Netherlands.* ACM, New York, NY, USA, 12 pages. *https://doi.org/10.1145/3847352.3848106*



AISec '26, The Hague, Netherlands

ACM ISBN 979-8-4007-3031-3/2026/11
https://doi.org/10.1145/3847352.3848106

1 Introduction

Binary code analysis is essential for understanding closed-source software, supporting a broad range of tasks, from detecting vulnerable or malicious code in binaries to reconstructing program semantics during reverse engineering [33, 35, 56]. At a finer level of granularity, tasks such as function boundary detection [3], parameter identification [10], and extraction of compilation details [43] serve as essential building blocks for analyzing and understanding binary code. While traditional tools for these tasks require substantial manual engineering, machine learning has transformed this dynamic by enabling solutions to be learned directly from data. As a result, learning-based methods have become a crucial instrument to improve and automate binary code analysis.

Substituting these manual tasks, however, is not straightforward, as machine learning critically depends on both the quality and the quantity of the available data. One method that has consistently led to performance improvements, by increasing the size and diversity of training data, is *data augmentation*. While its application is straightforward in many domains, such as computer vision [27, 46], where common augmentations include rotations, brightness adjustments, or cropping, a similarly well-defined set of transformations is not immediately available for binary code. For example, there is no direct analogue to rotating an image or adjusting its brightness when working with assembly instructions. Motivated by the success of this technique in other machine learning domains, a growing body of research has begun to explore its potential for machine learning-based binary code analysis. As a starting point, most approaches define their own task-specific augmentations for binary code. However, these approaches draw inspiration from diverse domains, leading to different names, implementations, and constraints for their augmentation.

Wong et al. [55] study data augmentation for malware classification, deriving semantics-preserving transformations from mutations observed in malware samples and on code obfuscation patterns. Shroyer et al. [47], by contrast, adapt NLP-inspired transformations that delete, duplicate, or reorder instructions to improve code similarity models. These approaches differ in motivation, definition, and even semantic preservation: the former preserves semantics by design, while the latter offers no such guarantee, for example, when instructions are removed. This heterogeneity across augmentation strategies and evaluation tasks makes it difficult to assess the effectiveness of data augmentation for learning-based binary code analysis.

*Authors contributed equally.

In this paper, we take a step back and present a systematic analysis of data augmentation for learning-based binary code analysis in five different downstream tasks. As a foundation for this study, we introduce a unified framework for data augmentation in this domain. At its core, the framework is based on edit operations on code, such as the insertion, deletion, and substitution of instructions. Building on these three primitives, we consolidate and express a diverse range of transformations from prior works that aim at increasing code diversity for learning-based binary code analysis.

To systematically analyze augmentation in this domain, we conduct a series of experiments. First, we measure how the effectiveness of data augmentation varies with dataset size on four common binary code analysis tasks: function parameter count and type analysis [10], function similarity measurement [12, 37], and compiler identification [43]. For a fair comparison, we adopt models and training procedures from prior work and modify only the training data through augmentation. Our results show that augmentation can improve performance when training data is scarce, but that the gains diminish as more data becomes available.

Second, we examine the impact of individual transformations on the performance of different tasks. We find that each task is highly sensitive to at least one specific transformation, while the same transformation has little effect on other tasks, suggesting that the selection and configuration of augmentation strategies should be adapted to the target task.

Third, we compare augmentations that strictly preserve execution semantics with those that allow small semantic changes. Interestingly, our results indicate that strict semantic preservation is not always required for effective augmentation. We relate this observation to a gap between the execution semantics and the semantics that are relevant for the prediction target of the learning task.

With our unified augmentation framework and systematic evaluation across multiple binary code analysis tasks, we provide a foundation for researchers and practitioners to better understand and apply data augmentation in this domain. Our results show that no single augmentation strategy dominates across tasks, and we therefore contribute to a more nuanced view on data augmentation in the field. In summary, we make the following contributions:

(1) *Unified augmentation framework.* We introduce a unified framework for augmenting binary code, built on the primitives of instruction insertion, deletion, and substitution. This framework enables the design and exploration of a broad range of binary code augmentations.
(2) *Empirical Study.* We present the first systematic study of augmentation for binary code analysis. We implement common code transformations and evaluate their impact on diverse analysis tasks.
(3) *Actionable recommendations.* We derive a set of practical recommendations for the application of data augmentation in machine learning-based binary code analysis.

To foster further research in machine learning for binary code analysis and to support the adoption of data augmentation in the field, we make our framework[1] and our datasets[2] publicly available to the community.

[1] *github.com/mlsec-group/laica*
[2] *huggingface.co/collections/mlsec-group/laica-datasets*

2 Binary Code Analysis

The goal of binary code analysis is to partially reverse the compilation process and infer properties of a program directly from its compiled form. Such an analysis is necessary whenever the source code is unavailable. For instance, a common task in reverse engineering is to infer a function's boundary or signature [4, 6, 10] or to reconstruct control-flow and data-flow information [18, 21]. Moreover, binary code analysis is also essential for security auditing, enabling the identification of vulnerable code in proprietary software [35, 42], and for detecting malicious components in the context of malware analysis [31, 59].

2.1 Binary Code Analysis

During compilation, much of a program's high-level semantics, such as data structures, control abstractions, and variable names, is lost. As a result, binary code analysis typically begins by recovering the remaining information still embedded in the binary by disassembling it [10, 33, 59]. The resulting representation provides a more tractable format for more advanced analyses, including binary lifting [20, 51], control-flow analysis [18, 21], and decompilation [8, 58], compared to the raw binary data.

For our analysis, we thus focus on disassembled code and use the term binary code to refer to the resulting assembly instructions. Formally, we define binary code as a sequence

$$s = \langle a_1, \ldots, a_n \rangle ,$$

of assembly instructions a_i. This sequence s may represent code in varying levels of granularity, ranging from basic blocks to functions or even entire binaries. Unless stated otherwise, we default to a function-level granularity, which aligns with prior work on binary code analysis [10, 12, 49].

Under this definition, we can formalize a generic analysis task as a recovery function f that extracts a property p from a given sequence of instructions s

$$f : s \mapsto p.$$

For example, p may represent a categorical property, such as the number of function parameters, the presence of malicious code, or a specific control-flow pattern. Similarly, p may also represent continuous attributes, such as the similarity to other functions or the likelihood of a vulnerability. In the following, we refer to the set of all sequences that possess a given property p, or fall within a similar range, as $\mathbb{S}_p$. This set $\mathbb{S}_p$ provides the foundation for designing augmentations that may slightly perturb the semantics of a binary code while preserving the targeted property p.

2.2 Learning-Based Binary Code Analysis

Similar to other domains, learning-based approaches have become increasingly prominent in binary code analysis, often outperforming traditional techniques that rely on handcrafted rules or complex program analyses [10, 41]. These approaches aim to train a learning model f_θ that approximates the desired recovery function f by minimizing a regularized loss L over a labeled dataset $\mathcal{D} = \{(s_i, p_i)\}_{i=1}^n$,

$$\theta^* = \arg\min_\theta \; L(f_\theta, \mathcal{D}).$$

This formulation essentially recasts binary code analysis as a standard learning problem that is structured around two core components: (a) the construction of training datasets and (b) the optimization of model parameters through machine learning.

The majority of existing work on learning-based approaches for binary code analysis has concentrated on the second component, including novel model architectures [1, 26, 57], training paradigms [40, 41, 49], and instruction embeddings [12, 30]. However, more recently, the role of data and its impact on performance have received increasing attention, catching up with other areas of machine learning, where data curation and preparation have long been central pillars of progress [50, 54].

2.3 Data Augmentation

One of the most widely used strategies for preparing and enhancing data in machine learning is its *augmentation*. The goal is to synthetically increase the diversity of the training data, enabling models to generalize more effectively to a given target property. In the image domain, for instance, this is achieved through standard image preprocessing transformations such as rotation, brightness adjustment, or cropping, each introducing controlled variance while preserving the underlying label of the samples [44, 50].

Binary code analysis has no standard approach to augmentation, mainly because assembly admits no natural transformations comparable to the rotations and crops used in computer vision. We therefore start from the transformations themselves. We define a transformation as a function T that maps an input sequence of assembly instructions s to an output sequence z, with both sequences belonging to the same property class:

$$T : \mathbb{S} \to \mathbb{S},\ s \mapsto z,\ \text{with } s, z \in \mathbb{S}_p.$$

Building on this notion, we can then define a code augmentation as a sequence of such transformations, composed to introduce controlled variability into the training data while preserving the targeted property p,

$$\begin{aligned} A &: \mathbb{S} \to \mathbb{S},\ s \mapsto (T_1 \circ \cdots \circ T_n)(s) \\ \text{with} \quad & s \in \mathbb{S}_p \Rightarrow A(s) \in \mathbb{S}_p\,. \end{aligned}$$

In essence, each transformation must preserve the semantic label of the original sequence: if $(s_i, p_i) \in \mathcal{D}$, then the augmented sequence $A(s_i)$ must retain the same label p_i.

For image classification, this requirement is generally easy to verify. After all, an image of a rotated bird still shows a bird. In contrast, the effects of modifying a sequence of binary code $s = \langle a_1, \ldots, a_n \rangle$ are more subtle. In fact, flipping a single bit is enough to change a `JZ` instruction to a `JNZ`, potentially introducing or eliminating a vulnerability, which may fundamentally alter the program's semantics and, consequently, its properties.

2.4 Current Augmentation Practices

A growing number of works have begun to implement transformations that enable data augmentation for machine learning-based binary code analysis. Wong et al. [55] introduce a binary mutator for malware classification that implements several transformations such as inserting junk code, obfuscating and substituting instructions, or inserting opaque predicates, while ensuring semantic equivalence through binary rewriting.

Shroyer and Swany [47] employ semantics preserving transformations for code similarity analysis on binary code, using a decompiler and applying transformations such as replacing arithmetic operations by equivalent alternatives, or extracting conditional branches to separate functions, using a C obfuscator. Additionally, they apply NLP-inspired transformations such as deleting, duplicating, and transposing instructions, that do not necessarily preserve execution semantics. Surprisingly, they observe improved model performance even when employing these transformations to their code similarity task.

McLaughlin and del Rincon [38] investigate data augmentation using solely non semantics-preserving transformations such as dropping parts of the input, or replacing it randomly. Using these transformations they observe improved model performance for opcode sequence-based malware detection.

Notably, many of these transformations are not unique to the data augmentation domain. Software diversity employs equivalent instruction substitution, instruction reordering, and junk code insertion to diversify binaries as a defense against code-reuse attacks [28] and binary obfuscation relies on overlapping techniques to alter code appearance while preserving behavior [32].

Across these works, we observe vastly different approaches to defining and describing operations. Usually this is done in prose, employing the vocabulary from the domains the ideas stem from. This makes it particularly difficult to identify similar augmentations across publications, as well as recognizing differences in those that might sound identical. Additionally, while some approaches are rooted in software diversity [28] and constrain these augmentations such that only strictly semantically equivalent samples are derived, others adopt relaxed implementations that do not guarantee to preserve semantics.

3 A Unified Augmentation Framework

Although existing augmentation approaches for binary code analysis differ in origin and detail, they share a common structure. Each operates on instruction sequences through a small set of primitive operations, namely the insertion, deletion, and substitution of instructions. What distinguishes one approach from another is not the primitives themselves, but the constraints placed on them and the way they are combined.

Based on this shared structure, we introduce an augmentation framework centered on *constrained edit operations*: instruction insertion, deletion, and substitution. These operations are inspired by the edit distance formalism [29], which offers a theoretical basis for converting one sequence into another through a finite series of edits [19, 52]. When applied to binary code, this abstraction enables a principled strategy for generating diverse code variants.

3.1 Edit Operations over Instructions

To formally introduce these edit operations, we adopt a notation from the literature on sequence processing [19]. Let $s[i]$ denote the i-th instruction in a sequence, and $s[i, j]$ the contiguous slice from the i-th to the j-th instruction, inclusive. Then, the insertion operation INS can be defined over binary code as follows

$$\text{INS}(s, i, z) := s[1, i] \mid z \mid s[i + 1, n],$$

where the operation introduces a new sequence z into s after the i-th instruction using a concatenation $|$. Similarly, the deletion operation DEL removes a contiguous slice of l instructions starting at position i:

$$\text{DEL}(s, i, l) := s[1, i-1] \mid s[i+l, n].$$

Finally, the substitution operation SUB replaces a slice of instructions from position i in s with a new sequence z. While this operation can also be defined explicitly, we opt for a simpler formulation using functional composition:

$$\text{SUB}(s, i, l, z) := \text{INS}(\text{DEL}(s, i, l), i, z).$$

3.2 Constraining Edit Operations

The flexibility of edit operations comes with a caveat: arbitrary modifications risk degrading the learning signal, so each edit must leave the sequence informative for the model. A natural way to enforce this is to require transformations to preserve the program's execution semantics, as is standard in the domains of software diversity and code obfuscation where transformed binaries are strictly required to preserve the execution semantics.

However, prior work demonstrates that augmentations can improve model performance even when semantic fidelity is not strictly preserved [38, 47]. To allow both of these transformation types to be expressed in our framework, we define a set of three constraints that limit the application scope of operations and allow for making explicit which aspects of semantics are preserved. Rather than requiring full execution equivalence, these constraints allow us to selectively relax semantic preservation, specifying precisely which semantic aspects a given transformation must retain and which ones it is permitted to alter.

Constraint 1: Semantic equivalence. First, we use $s \sim z$ to denote semantic equivalence between two instruction sequences with respect to their primary computational effect, abstracting away from architectural side effects. For example, $\langle$mov eax, 0$\rangle \sim \langle$xor eax, eax$\rangle$, although the two instructions differ in their effects on processor flags. The relation is defined over instruction sequences of arbitrary length and compares them as a whole rather than instruction by instruction. At the level of individual instructions, it captures local equivalences such as in the example above. When applied to entire functions or binaries, it coincides with semantic equivalence in the classical sense.

Constraint 2: Data independence. Second, we define data independence using the operator $\perp$. A sequence of instructions $\langle a_1, a_2 \rangle$ is data independent, written $a_1 \perp a_2$, if the execution of a_1 has no impact on the data dependencies of a_2. Specifically, we require that the following conditions hold: First, there must be *no read-after-write* conflicts. That is, a_2 must not read from any register or memory location that is written by a_1. Second, there must be *no write-after-read* conflicts, so that a_2 must not write to any location that a_1 reads from. Third, there must be *no write-after-write* conflicts, where a_2 must not write to any register or memory location that is also written by a_1. This definition applies to successive instructions in a sequence, independent of the order in which they are ultimately executed.

Constraint 3: Execution order. Finally, we use the operator $\prec$ to describe the execution order of two instructions. In a sequence s, instructions are laid out in ascending address order, but execution may not follow this order due to control flow transfers like jumps or branches. We use $a_i \prec a_j$ to denote that a_j is guaranteed to execute after a_i, regardless of whether the instructions are contiguous in the sequence. The same applies to instruction sequences: $s_i \prec s_j$ signifies that every instruction in s_j is executed after every instruction in s_i.

3.3 Code Transformations

Prior work has explored a wide range of transformations. However, these transformations are often named and defined in an inconsistent way, as they are inspired by different domains, and their constraints are frequently not stated explicitly. For example, both Wong et al. [55] and McLaughlin and del Rincon [38] describe a transformation that inserts instructions into a binary, but only the former enforces semantic preservation. Our framework addresses these issues by describing each transformation in terms of the edit operations it applies and the constraints it satisfies. This allows precise definitions and enables direct comparison across different approaches.

We collect augmentation methods from prior work and express them within our framework in Table 1. As illustrated by the example of basic block reordering, the framework is flexible enough to represent complex transformations. In particular, it can describe the reordering of basic blocks b_i and b_j through appropriate combinations of operations and constraints. In total, we identify eight individual augmentations that can be grouped into one of three classes depending on the operation that prevails in its definition.

Inserting transformations. The simplest insertion transformation is *NOP insertion*, which places a no-operation instruction at an arbitrary position. A generalization from the software diversity literature, *instruction insertion*, inserts a sequence z at an arbitrary position. Prior work has applied it both with the constraint $z \sim \langle a_{\text{nop}} \rangle$, ensuring semantic neutrality [55], and without it [38, 47]. A related technique, *dead code*, injects instructions that are unreachable at runtime also with [55] and without [47] semantic constraints.

Substituting transformations. The most direct form is *instruction substitution*, which replaces a sequence s with an alternative z. Under the strict constraint $s \sim z$, the replacement is semantically equivalent [55]. Without this constraint, substitution can deliberately introduce noise [38, 47]. Restricting substitution to data-independent instruction pairs while constraining execution order yields *instruction swapping*, which exchanges the positions of two instructions. Both the constrained form [55] and an unconstrained variant [47] appear in prior work. A coarser variant, *basic block reordering*, permutes entire basic blocks within a sequence and is a standard technique in software diversity [28].

Deleting transformations. The final class removes instructions from sequences. *Instruction erasing* deletes instructions from arbitrary positions within a sequence, typically without semantic constraints [38, 47]. *Truncation* similarly deletes instructions without semantic constraints, but is restricted to the beginning or end of the sequence.

Table 1: Overview of code transformations.

Description	Transformation T	Constraint C	Related Work
Inserting transformations			
NOP insertion	$\textsc{ins}(s, i, z)$	$z = \langle a_{\text{nop}} \rangle$	[22, 25, 55]
Instruction insertion	$\textsc{ins}(s, i, z)$	$z \sim \langle a_{\text{nop}} \rangle$	[22, 32, 38, 47, 55]
Dead code	$\textsc{ins}(s, i, z)$	$z_1 \prec s_{i+1}$	[32, 55]
Substituting transformations			
Instruction substitution	$\textsc{sub}(s, i, l, z)$	$z \sim s_i, l = 1$	[38, 39, 47, 55]
Instruction swapping	$\textsc{sub}(\textsc{sub}(s, i, l_i, s_j), j, l_j, s_i)$	$s_i \perp s_j, l_i = l_j = 1$	[39, 47, 55]
Basic block reordering	$\textsc{sub}(\textsc{sub}(s, i, l_i, b_j), j, l_j, b_i)$	$s_i \prec s_{i+l_i}, s_j \prec s_{j+l_j}$	[11, 53]
Deleting transformations			
Instruction erasing	$\textsc{del}(s, i, l)$	$i \leq n - l$	[38, 47]
Truncation	$\textsc{del}(\textsc{del}(s, n - l, l), i, l)$	$i = 1$	[38, 47]

3.4 From Transformation to Augmentation

So far, each transformation describes a single local modification of a sequence. Although such transformations can be used directly for data augmentation, applying one at a single site only changes the sample slightly, thus yielding limited diversity. A naive remedy is to apply a transformation to every region of a sequence that satisfies its constraints. However, this deterministic strategy produces only one possible outcome per sample: the most strongly modified variant the transformation can achieve.

To balance these two extremes, we adopt a probabilistic approach inspired by the randomized schemes used in software diversity [28], which allows the strength of a transformation to be varied gradually. Each transformation T is therefore parameterized by a probability π that controls the fraction of valid locations at which the transformation is applied. For example, setting $\pi = 0.5$ for the instruction swapping transformation modifies, on average, half of the instruction pairs that can be swapped without violating the data independence constraint.

This scheme applies to most considered transformations, but some require an adapted formulation. The truncation transformation, for instance, removes instructions at the beginning or end of an instruction sequence. Here, the relevant factor is not the number of code locations at which the transformation is applied, but the number of instructions it affects. For such transformations, the strength is defined as a quantity, and the parameter π is used to sample this quantity at random.

For five of the eight transformations, π determines which locations in s are selected. Specifically, in this case, a transformation is applied to a location that satisfies the constraint with a probability π. For the remaining three transformations π governs the length of the edited sequence. That is, for truncation and instruction erasing, the number of instructions to delete, and for the dead code transformation, the length of the added sequence. In each of these cases, the respective count is drawn from a pre-defined range according to the parameter π.

Building on this probabilistic application for a single transformation T, we define the overall augmentation using multiple transformations $T_1, \ldots, T_k$ as

$$A(s, \Pi) = \bigcirc_{i=1}^{k} T_{i,\Pi_i}(s),$$

where $\Pi \in [0, 1]^k$ is a probability vector, and Π_i denotes the probability associated with transformation T_i. These probabilities control the extent of augmentation and additionally serve as a mechanism for preserving desired properties: transformations that are more likely to disrupt the target property can be applied with a lower probability.

4 Evaluation

Equipped with a unified definition of common augmentations used in learning-based binary code analysis and software diversity, we can now turn to a critical examination of their application in practice. To this end, we analyze the effect of data augmentation through a series of four experiments:

- E1 *Binary code augmentation.* We begin by exploring data augmentation across a wide range of common analysis tasks and as a function of the available training data.
- E2 *Impact analysis.* Next, we analyze the individual contribution of each transformation to model performance, isolating its impact for a given task.
- E3 *Preserving semantics.* We then investigate the effects of relaxing the definition of semantic equivalence, comparing transformations under a strict and a relaxed interpretation.
- E4 *Vulnerability detection.* Finally, we conduct a case study on vulnerability detection, one particularly demanding binary analysis task where the target property of the learning problem is closely linked to the code semantics.

4.1 Experimental Setup

In our first three experiments, we study how data augmentation affects model performance across different tasks and as a function of the available training data. Investigating this scaling behavior requires labeled data of sufficient size and quality, which rules out tasks such as vulnerability detection or malware classification, where reliable labels are costly to obtain at scale. We therefore first focus on tasks for which large, high-quality labeled datasets can be generated automatically.

To this end, we build on the framework proposed by Maier et al. [34] to construct labeled binary code datasets from Debian packages for multiple downstream tasks. The framework extracts the ELF binaries and recovers function boundaries from their DWARF debugging sections, decoupling our augmentation analysis from potential errors of function boundary detection heuristics. The functions are subsequently disassembled using the Capstone engine[3]. For each of our tasks, we adopt the model and training procedure used in prior work and modify only the training data through our augmentation methods.

Task 1: Function similarity. We first address binary function similarity, a core task in reverse engineering [16, 17, 23]. Following prior work, we define two functions as similar if they originate from the same source code but differ due to compiler configurations [34, 37, 57]. This task uses the siamese model of Maier et al. [34], which builds on the architecture proposed by Massarelli et al. [37], using Asm2Vec embeddings [12] to model function similarity across compilation settings.

Task 2: Compiler identification. The second task is to determine which compiler was used to generate a given binary. Prior work has shown that this task can be effectively addressed with machine learning [9, 15, 43]. We adopt the model of Pizzolotto et al. [43], which employs embedding and dense layers to distinguish between binaries compiled with Clang and GCC.

Task 3: Parameter count inference. The third task is parameter count inference, that is, predicting how many parameters a given function accepts. Prior work casts this problem as multi-class classification [10], where each function is assigned a label corresponding to its expected number of parameters. We adopt the model proposed by Chua et al. [10], which comprises three stacked GRU layers, dropout regularization, and a dense output layer for classification.

Task 4: Parameter type inference. The fourth task is parameter type inference: classifying the type of the first parameter a function processes. Prior studies have demonstrated the effectiveness of learning-based methods for type inference [40, 45, 61, 62]. We employ the same architecture as for parameter count inference, adapting only the output layer to account for parameter types.

For all tasks, we follow prior work [34] and build our dataset from Debian packages, splitting it at the level of source packages. Functions from the same package therefore never appear in two splits, so shared binaries and internal dependencies cannot leak between them. Some overlap between packages may remain, but such cases are rare and reflect actual code reuse found in real software. The full split is summarized in Table 2. We compile each package for the x86-64 instruction set at optimization levels O0 through O3. The compiler configuration depends on the task: compiler identification needs both Clang and GCC binaries, while function similarity, parameter count, and type inference use Clang only. From the resulting binaries, we extract approximately 600k functions comprising 60M instructions in total.

[3]https://www.capstone-engine.org

Table 2: Overview of dataset used for low-level analysis tasks.

Dataset	Debian Package	Functions	Instructions
Training	`binutils`	465 833	53 373 623
	`findutils`	4 375	407 149
		470 208	53 780 772
Validation	`coreutils`	46 843	3 310 264
	`diffutils`	2 415	240 293
		49 258	3 550 557
Testing	`util-linux`	45 533	3 462 469
	`inetutils`	8 814	849 123
	`sg3-utils`	3 703	660 270
	`usbutils`	440	60 458
		58 490	5 032 320
Total		577 956	62 363 649

4.2 Augmentation Configuration

We implement the eight augmentations described within our framework in Section 3 using an approximate binary rewriting approach. Specifically, we disassemble the binary code, perform the transformation, and re-assemble it. The details of our implementation are outlined in Appendix B. Since our implementation enforces only the specified constraints, aspects of execution semantics that fall outside their scope are not guaranteed to be preserved. For example, jump targets in unconstrained regions might be mis-aligned after transformation.

Further, our augmentation framework applies each transformation probabilistically and allows explicit control over the application probabilities. Choosing these probabilities is crucial: they govern the trade-off between increased data diversity and the preservation of properties relevant to the analysis task. This trade-off between increased data diversity and property preservation in a machine learning-based binary code analysis system is reflected in the utility of the model.

Therefore, we conduct a hyperparameter study using sequential model based global optimization [7]. This approach relies on surrogate models of the objective function based on previously observed evaluations. This surrogate model is then used to propose promising values for our transformation parameters Π. These values are subsequently evaluated using the more expensive true objective, namely training a model and measuring its utility. This procedure reduces the number of required training runs and enables a more efficient parameter search compared to brute force or grid search methods.

For the implementation, we use the hyperparameter optimization framework Optuna [2] and continue the search until at least 1,000 parameter configurations have been evaluated, following the recommendations for this type of study. For each task, we use 7k randomly sampled data points from the respective training dataset, which keeps individual training runs tractable within the search budget while preserving the relative ranking of configurations. The configuration achieving the highest validation performance is then used in the experiments reported in the following sections.

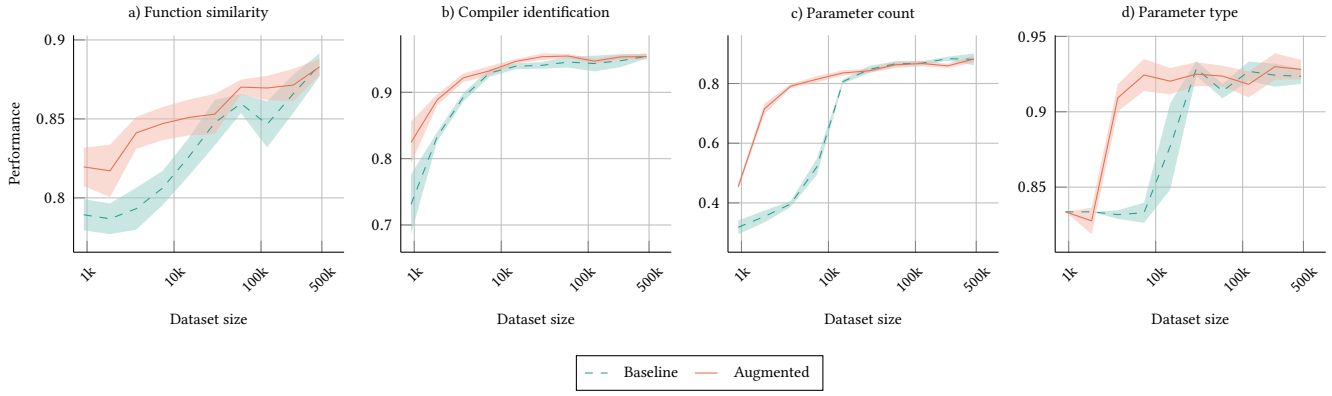


Figure 1: Average performance with 95% confidence intervals for models trained with and without data augmentation across dataset sizes. Performance axes are scaled independently per task.

4.3 Binary Code Augmentation (E1)

We begin by assessing data augmentation across the four binary code analysis tasks. Since the benefit of augmentation is expected to depend on the amount of training data available, we evaluate each task across a range of dataset sizes. We use successive fractions $1, \frac{1}{2}, \frac{1}{4}, \ldots, \frac{1}{512}$ of the full training sets, ranging from approximately 470k samples at the largest scale to about 1k at the smallest.

Setup. For this experiment, we consider the four tasks and eight transformations from Section 3. Augmentations are tuned using the optimization described in Section 4.2. For the models trained with augmented data, we additionally tune the number of synthetic samples introduced during the training process. To ensure statistical robustness, all training procedures are repeated ten times.

Results. Figure 1 shows the mean performance for the four tasks, together with the corresponding 95% confidence intervals computed over ten repetitions. Across all tasks, reaching full model performance requires a minimum number of training samples, which depends on the specific task. Data augmentation can reduce this requirement, but it cannot replace the need to collect new data when the available training set is too small. Nevertheless, augmentation leads to substantial performance improvements in low-data regimes. In contrast, when a large amount of training data is available, the performance gap between augmented and non-augmented settings diminishes. This suggests that, although data augmentation cannot replace the collection of new data, it can reduce the associated costs, as generating synthetic samples is typically much cheaper.

When examining the influence of the task on the magnitude of improvement due to data augmentation, we observe substantial variation across tasks. For function similarity, the gains remain modest, reaching up to 4 percentage points, whereas for parameter count inference they are as high as 39 percentage points. Notably, however, we do not observe a statistically significant decrease in performance for any task or dataset size. Overall, these results indicate that augmentation provides a consistent benefit in low-data regimes without measurable risk, supporting their use as a standard component for learning-based binary code analysis.

4.4 Impact Analysis (E2)

So far we have tuned augmentations per task, on the assumption that augmentation effectiveness is task-specific and that no single augmentation is best everywhere. To test this assumption, we measure the relative impact of each transformation T_i within the entire set $\mathcal{T}$. We use fANOVA [24], which decomposes the variance in model performance into contributions from individual transformations and their interactions. Formally, we define the *importance* ψ_i of a transformation T_i as

$$\psi_i := \frac{\mathrm{Var}_{T_i}[\mathbb{E}_{\mathcal{T}_{|i}}[\mathrm{perf}(\mathcal{T})]]}{\mathrm{Var}[\mathrm{perf}(\mathcal{T})]}.$$

Here, $\mathcal{T}_{|i}$ denotes that T_i is held fixed while the remaining transformations vary. The function $\mathrm{perf}(\mathcal{T})$ captures the performance of a model when trained on data augmented with transformations $\mathcal{T}$. A high ψ_i thus means that the choice of T_i alone explains much of the spread in performance across augmentation configurations. Note that ψ_i measures influence rather than benefit: a transformation that consistently degrades performance receives a high importance as well.

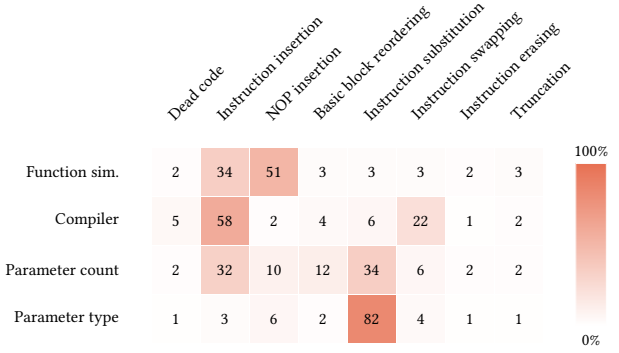


Figure 2: Impact of the individual transformations on each downstream task. Values give the fraction of performance variance attributable to a transformation from 0% to 100%.

Results. We report first-order effects, normalized per task so that the importances of all transformations sum to 100% in Figure 2. Under this normalization, a transformation at 0% explains none of the performance variance for that task, whereas one at 100% explains all of it. The results show strong task-specific effects. For example, for parameter type inference, *instruction substitution* dominates with an importance of 82%, indicating that the model performance is particularly sensitive to this type of transformation. In contrast, the same transformation reaches only 3% on the function similarity task.

More generally, every task is highly sensitive to at least one transformation (importance > 30%) that is nearly irrelevant to another (< 5%). This indicates that augmentation strategies should be chosen per task. For three transformations, *dead code insertion*, *instruction erasing*, and *truncation*, importance stays low on every task. The model is thus largely indifferent to whether they are applied. Indifference is especially informative for *instruction erasing*, since it alters program semantics substantially and still leaves performance unchanged. Semantic fidelity therefore appears to matter less for augmentation than the diversity a transformation introduces.

4.5 Preserving Semantics (E3)

The previous analysis suggests that strict semantic preservation is not always required. This is consistent with the findings from prior work [38, 47], motivating a more detailed comparison. Therefore, we re-implement three transformations, instruction substitution, basic block reordering, and instruction insertion, as compiler passes for Clang. These passes operate on the LLVM-IR, which avoids many of the pitfalls of directly manipulating binary code and allows the strict variant to preserve execution semantics as closely as possible. This is feasible because we have access to the source code and the build process, an option typically not available in binary analysis.

Setup. We repeat our initial experiment on the restricted sets of the three shared transformations and compare the performance between both sets. We exclude the compiler identification task for this experiment as the LLVM-based augmentations are specific to Clang. We optimize the parameters of the augmentations operating under relaxed semantic preservation as described in Section 4.2. Optimizing the more strictly semantic preserving transformations based on LLVM-passes is not practically feasible, as each augmentation pass requires the compilation of every package in the training set. Allocating the same amount of resources during optimization would, hence, result in only a fraction of optimization steps compared to the augmentation with our approach. Therefore, to put the two approaches on a more equal footing, we use the parameters from the optimization of our relaxed augmentations for both. After optimization, we train the models ten times for each set of optimized transformations to ensure consistent and reliable results.

Results. Figure 3 presents the performance of the two augmentation strategies across the three tasks. When sufficient data is available, the two strategies perform largely indistinguishable, suggesting that strict semantic preservation provides no measurable benefit over the relaxed variant at larger dataset sizes. Notably, this is the case for all three tasks considered even though they require different levels of semantic preservation. For example, slightly changing the semantics of a function might not change its signature

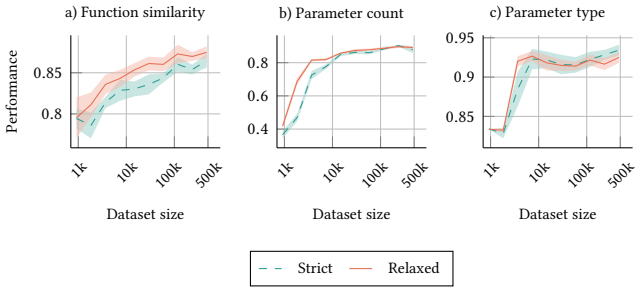


Figure 3: Average performance on each task for relaxed and strictly semantics-preserving transformations, over increasing dataset sizes up to the full dataset. The two variants perform comparably across all tasks and dataset sizes.

but may have a large impact on its similarity to other functions. Moreover, we observe that when only little data is available, the relaxed variant performs on par with or slightly better than the strictly semantics-preserving augmentation in several cases, a finding that at first glance runs counter to the intuition that stricter correctness guarantees should yield better augmented training data.

We offer two complementary explanations for these results. First, the parameters used for both strategies were optimized exclusively for the relaxed variant, which could introduce a bias against the strict variant in this comparison. Second, execution semantics and the target property may be considered disentangled, as they are not the same concept and do not vary together in general. By coupling the augmentation criterion to a property the model is not asked to learn, strict semantic preservation conflates two dimensions and unnecessarily restricts the augmentation space. Removing a return statement, for example, does not change the number of parameters that a function accepts. Such transformations may even be beneficial when they push the model away from spurious shortcuts.

Therefore, optimizing augmentations for the target property might offer a better alternative to universally requiring strict semantic preservation. Where full semantic equivalence does matter, the optimization selects only preserving transformations, or drops augmentation entirely in the worst-case. Furthermore, relaxed transformations might offer an additional benefit, as they are generally simpler to implement and cheaper to apply in practice.

4.6 Vulnerability Detection (E4)

The results of the previous experiment indicate that strict preservation of execution semantics is not a necessary condition for effective data augmentation across the considered tasks. Although this observation is consistent with prior work, it remains somewhat surprising. To assess the extent to which this observation holds, we now consider a setting in which the target property is more directly tied to program behavior. In particular, we study vulnerability detection, where subtle changes in execution semantics can influence whether a piece of code should be classified as vulnerable. This setting provides a natural test case to assess whether relaxed augmentations remain effective when semantic properties are expected to play a more central role.

Dataset. Since no well-established dataset exists for binary code vulnerability detection, prior approaches rely either on synthetic data or on narrowly scoped real-world datasets, which often cover fewer than ten projects with annotated vulnerabilities [5, 33, 35]. Vulnerability datasets for source code are considerably more mature. The most comprehensive and high-quality resource currently available, PrimeVul [13], covers 725 projects and includes 5,175 documented vulnerabilities. Building on PrimeVul, we manually compile a subset of the 22 projects with the largest share of vulnerabilities across dataset splits, covering 1,378 vulnerabilities in total. Heuristically mapping the source code vulnerabilities to their binary code counterparts leaves us with a total of 783 vulnerabilities. We keep the temporal split of the original source code dataset and give a breakdown of each in Table 3.

Setup. To evaluate the augmentations in this setting, we adopt a model from previous work, specifically the Bi-LSTM architecture proposed by Aumpansub et al. [5] and a learned instruction representation. As before, we first optimize the augmentation specifically for this task using a 20-hour optimization budget, and repeat model training ten times to ensure robustness.

Results. In the absence of augmentation, the model achieves an average F1 score of 15.7%. With augmentation, performance improves to 22%, an absolute gain of 6.3 percentage points and a 40% relative improvement. While an F1 score of 22% indicates that the task remains far from being solved, these results are consistent with vulnerability detection performance on the source code level of the full PrimeVul dataset [13]. More importantly, despite the more strongly pronounced link between code semantics and the target property of the analysis task, augmentations that operate on a relaxed definition of semantic preservation still improve model performance. While surprising at first glance, taking a closer look into the individual augmentations reveals a potential explanation.

To this end, we repeat our impact analysis of E2 for this task. We find that, compared to the tasks considered in our previous experiments, the vulnerability detection task shows approximately a fourfold increase in sensitivity to truncation-based augmentation and a tenfold increase in sensitivity to random instruction removal. This suggests that vulnerability detection is more sensitive to changes in code semantics. However, as the hyperparameter study assigns low probabilities to these augmentations, the complete set of augmentations still leads to an overall improvement in model performance. While other augmentations also affect execution semantics, for example due to misaligned jump targets after inserting dead code or reordering basic blocks, these more subtle semantic modifications do not appear to impact the model performance negatively.

5 Discussion & Recommendations

Our study provides a systematic examination of data augmentation in the context of learning-based binary code analysis. Prior work has often considered augmentation as a generally beneficial strategy. In contrast, our results suggest a more nuanced picture. Data augmentation is not a universally effective solution. Instead, its impact depends strongly on the specific task, the type of transformation applied, and the amount of available training data.

Table 3: Overview of the binary-level vulnerability dataset derived from PrimeVul. We sample vulnerabilities separately from the train, validation, and test splits of the original dataset, so its temporal split is preserved.

Dataset	#Projects	#Versions	#CWEs	#CVEs
Training	21	1,413	72	1,097
Testing	18	232	33	204
Validation	19	149	33	125
Total	22	1,790	77	1,378

More specifically, our evaluation shows that data augmentation consistently improves performance in low-data regimes, although the magnitude of this improvement varies across tasks. We further observe that these gains do not rely on strictly semantics-preserving transformations. Rather, effective augmentation requires selecting transformations that match the characteristics of the target task and model. Based on these findings, we derive three concrete recommendations for effectively integrating data augmentation into learning-based binary code analysis:

R1 *Use augmentation as a supplement.* Data augmentation for binary code can substantially improve model performance. While it introduces additional cost to generate synthetic samples, it is typically cheaper than collecting and labeling new data. Moreover, we observe no statistically significant decrease in model performance across the tasks and dataset sizes we consider. We therefore recommend augmentation as a standard component of learning-based binary code analysis, which should be omitted only when the additional cost is prohibitive.

R2 *Default to relaxed augmentation.* Our experiments indicate that, somewhat counter-intuitively, a very strict semantic preservation is not always required for effective data augmentation. Transformations based on relaxed semantic constraints achieve performance comparable to those that enforce strict preservation on the tasks we compare, suggesting that approximate fidelity is often sufficient and the additional diversity triumphs over slight semantic deviations. This widens the range of usable transformations to those that are easier to implement and cheaper to apply without reducing effectiveness. We therefore recommend defaulting to relaxed constraints and resorting to strict preservation only where relaxed variants fail.

R3 *Tune augmentation to the task.* Our analysis shows that the effectiveness of individual augmentations varies markedly across tasks. Each task exhibits strong sensitivity to at least one of the transformations that may be largely irrelevant in others. To maximize the benefit of data augmentation in binary code analysis, transformations should be carefully tailored to the specific characteristics and requirements of the target task.

6 Related Work

Data augmentation is a mature technique in machine learning, with well-established practices in the domains of computer vision and natural language processing. In binary code analysis, by contrast, augmentation remains a young and fragmented field where basic questions are still open. The existing efforts adopt different vocabularies, propose overlapping but incompatible transformations, and reach inconsistent conclusions on points as fundamental as whether semantic preservation is required. This paper takes a step toward systematizing this picture by drawing on closely related work in software diversity and code obfuscation, and by relating our findings to comparable work on source-code augmentation.

Software diversity and obfuscation. Many of the transformations that appear in binary code augmentation have direct precedents in the software diversity and binary obfuscation literature, where they were introduced for different purposes. Software diversity uses randomized transformations such as NOP and instruction insertion [22, 25], instruction substitution and swapping [39], and basic block reordering [11, 53] to maximize implementation entropy and resist code-reuse attacks. Binary obfuscation employs overlapping techniques, including dead code insertion and instruction substitution, to hinder reverse engineering [32]. Both fields treat strict semantic preservation as essential, since the transformed binary must execute correctly in production. Our framework draws on the same primitives but, as the analysis tasks in our setting do not require runtime equivalence, can relax this constraint, opening a broader space of augmentations than these prior fields could exploit.

Data augmentation for source code. A more developed picture exists at the source code level. Yu et al. [60] apply strictly semantics-preserving transformations to Java source code and report consistent gains across multiple analysis tasks, even when small bugs are inadvertently introduced. Dong et al. [14] take the opposite approach and adopt NLP-style transformations such as synonym replacement and random swaps, which need not preserve semantics, and likewise observe consistent improvements. These two findings, mirrored by our own observations at the binary level, suggest that strict semantic preservation is not a prerequisite for effective augmentation across code analysis settings. Source-level methods, however, depend on the availability of source code and on representations and models tailored to source-level analysis, which limits their direct applicability to binary analysis.

Consolidation in binary code analysis. Our work follows a broader line of recent efforts that bring methodological clarity to fragmented subfields of binary and code analysis. Marcelli et al. [36] address the binary function similarity problem, re-implementing existing learning-based approaches in a common framework and show that many methods reporting state-of-the-art results perform comparably once evaluated on the same dataset. Maier et al. [34] take a similar perspective on pre-trained embeddings for binary analysis, finding that end-to-end learning matches or exceeds the performance of specialized embeddings when labeled data is available. Steenhoek et al. [48] apply the same lens to deep-learning-based vulnerability detection, observing that strong reported results often do not transfer across datasets.

A common pattern unites these efforts: approaches that appear incomparable as reported are re-examined side by side under consistent conditions, yielding a more nuanced picture than the one suggested by individual contributions. Our work continues this line for the case of data augmentation in binary code analysis.

7 Conclusion

Data augmentation is a well-established tool in machine learning, yet its application to binary code analysis has so far been fragmented and narrowly scoped. In this work, we introduce a unified framework for binary-level augmentation built on a small set of constrained edit operations. The framework subsumes the transformations used in prior work and makes the design choices behind them, in particular the constraints on semantic preservation, explicit and comparable. Through a systematic evaluation across five binary code analysis tasks, we examine when and how augmentation is effective in this domain. Our results offer a more nuanced picture than the prevailing view of augmentation as a generally beneficial strategy.

Augmentation reliably improves performance when training data is scarce, but the gains diminish as more data becomes available, and they vary substantially across tasks and transformations. Strict semantic preservation, often treated as a prerequisite for sound augmentation, might not always be required: relaxed transformations match the performance of strict ones across the tasks we compare. They remain effective even for vulnerability detection, a task whose target property is closely tied to program semantics, further supporting that strict runtime equivalence is not always necessary. From these findings, we derive practical recommendations for the use of augmentation in binary code analysis, including the use of relaxed transformations as a cheaper alternative and the task specific tuning of augmentation strategies.

More broadly, our work shows that the value of revisiting an established technique like data augmentation lies less in its mere adoption than in understanding when and how it pays off. The inconsistent application of augmentation in prior binary code analysis work has obscured both its strengths and its limits. By providing a unified framework and a systematic empirical study, we make these trade-offs visible and place augmentation on a firmer footing for future research and practice.

Open Science

To support further research on data augmentation for binary code analysis, we make all code artifacts of this work publicly available at *github.com/mlsec-group/laica*. Further, we release all datasets at *huggingface.co/collections/mlsec-group/laica-datasets*.

This includes our augmentation framework with the implementation of all eight transformations, the code and configurations to reproduce the experiments in Section 4, and the binary-level vulnerability dataset derived from PrimeVul [13].

Acknowledgments

This work was funded by the German Federal Ministry of Research, Technology and Space (BMFTR) under the grant "AIgenCY" (16KIS2012) and the European Research Council (ERC) under the consolidator grant MALFOY (101043410).

References

[1] Sunwoo Ahn, Seonggwan Ahn, Hyungjoon Koo, and Yunheung Paek. 2022. Practical Binary Code Similarity Detection with BERT-based Transferable Similarity Learning. In *Proc. of the Annual Computer Security Applications Conference (ACSAC)*.

[2] Takuya Akiba, Shotaro Sano, Toshihiko Yanase, Takeru Ohta, and Masanori Koyama. 2019. Optuna: A Next-generation Hyperparameter Optimization Framework. In *Proc. of the ACM SIGKDD International Conference on Knowledge Discovery & Data Mining (KDD)*.

[3] Jim Alves-Foss and Jia Song. 2019. Function Boundary Detection in Stripped Binaries. In *Proc. of the Annual Computer Security Applications Conference (ACSAC)*.

[4] Dennis Andriesse, Asia Slowinska, and Herbert Bos. 2017. Compiler-Agnostic Function Detection in Binaries. In *Proc. of the IEEE European Symposium on Security and Privacy (EuroS&P)*.

[5] Amy Aumpansub and Zhen Huang. 2022. Learning-based Vulnerability Detection in Binary Code. In *Proc. of the International Conference on Machine Learning and Computing (ICMLC)*.

[6] Tiffany Bao, Jonathan Burket, Maverick Woo, Rafael Turner, and David Brumley. 2014. BYTEWEIGHT: Learning to Recognize Functions in Binary Code. In *Proc. of the USENIX Security Symposium*.

[7] James Bergstra, Rémi Bardenet, Yoshua Bengio, and Balázs Kégl. 2011. Algorithms for Hyper-Parameter Optimization. In *Proc. of the Conference on Neural Information Processing Systems (NIPS)*.

[8] David Brumley, JongHyup Lee, Edward J. Schwartz, and Maverick Woo. 2013. Native x86 Decompilation Using Semantics-Preserving Structural Analysis and Iterative Control-Flow Structuring. In *Proc. of the USENIX Security Symposium*.

[9] Yu Chen, Zhiqiang Shi, Hong Li, Weiwei Zhao, Yiliang Liu, and Yuansong Qiao. 2018. HIMALIA: Recovering Compiler Optimization Levels from Binaries by Deep Learning. In *Proc. of the Intelligent Systems Conference (IntelliSys)*.

[10] Zheng Leong Chua, Shiqi Shen, Prateek Saxena, and Zhenkai Liang. 2017. Neural Nets Can Learn Function Type Signatures From Binaries. In *Proc. of the USENIX Security Symposium*.

[11] Bart Coppens, Bjorn De Sutter, and Jonas Maebe. 2013. Feedback-Driven Binary Code Diversification. *ACM Transactions on Architecture and Code Optimization (TACO)* 9, 4 (2013).

[12] Steven H. H. Ding, Benjamin C. M. Fung, and Philippe Charland. 2019. Asm2Vec: Boosting Static Representation Robustness for Binary Clone Search against Code Obfuscation and Compiler Optimization. In *Proc. of the IEEE Symposium on Security and Privacy (S&P)*.

[13] Yangruibo Ding, Yanjun Fu, Omniyyah Ibrahim, Chawin Sitawarin, Xinyun Chen, Basel Alomair, David A. Wagner, Baishakhi Ray, and Yizheng Chen. 2025. Vulnerability Detection with Code Language Models: How Far Are We?. In *Proc. of the International Conference on Software Engineering (ICSE)*.

[14] Zeming Dong, Qiang Hu, Yuejun Guo, Zhenya Zhang, Maxime Cordy, Mike Papadakis, Yves Le Traon, and Jianjun Zhao. 2025. Boosting Source Code Learning With Text-Oriented Augmentation: An Empirical Study. *Empirical Software Engineering* 30 (2025).

[15] Yufei Du, Omar Alrawi, Kevin Z. Snow, Manos Antonakakis, and Fabian Monrose. 2023. Improving Security Tasks Using Compiler Provenance Information Recovered At the Binary-Level. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*.

[16] Sebastian Eschweiler, Khaled Yakdan, and Elmar Gerhards-Padilla. 2016. discovRE: Efficient Cross-Architecture Identification of Bugs in Binary Code. In *Proc. of the Network and Distributed System Security Symposium (NDSS)*.

[17] Qian Feng, Minghua Wang, Mu Zhang, Rundong Zhou, Andrew Henderson, and Heng Yin. 2017. Extracting Conditional Formulas for Cross-Platform Bug Search. In *Proc. of the ACM Asia Conference on Computer and Communications Security (AsiaCCS)*.

[18] Wenbo Guo, Dongliang Mu, Xinyu Xing, Min Du, and Dawn Song. 2019. DEEPVSA: Facilitating Value-set Analysis with Deep Learning for Postmortem Program Analysis. In *Proc. of the USENIX Security Symposium*.

[19] Dan Gusfield. 1997. *Algorithms on Strings, Trees, and Sequences: Computer Science and Computational Biology.* Cambridge University Press.

[20] Niranjan Hasabnis and R. Sekar. 2016. Lifting Assembly to Intermediate Representation: A Novel Approach Leveraging Compilers. In *Proc. of the International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS)*.

[21] Jingxuan He, Pesho Ivanov, Petar Tsankov, Veselin Raychev, and Martin T. Vechev. 2018. Debin: Predicting Debug Information in Stripped Binaries. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*.

[22] Andrei Homescu, Steven Neisius, Per Larsen, Stefan Brunthaler, and Michael Franz. 2013. Profile-Guided Automated Software Diversity. In *Proc. of the IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*.

[23] Xin Hu, Tzi-cker Chiueh, and Kang G. Shin. 2009. Large-scale malware indexing using function-call graphs. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*.

[24] Frank Hutter, Holger Hoos, and Kevin Leyton-Brown. 2014. An efficient approach for assessing hyperparameter importance. In *Proc. of the International Conference on Machine Learning (ICML)*.

[25] Todd Jackson, Andrei Homescu, Stephen Crane, Per Larsen, Stefan Brunthaler, and Michael Franz. 2013. Diversifying the Software Stack Using Randomized NOP Insertion. In *Moving Target Defense II (Advances in Information Security)*. Springer New York.

[26] Nan Jiang, Chengxiao Wang, Kevin Liu, Xiangzhe Xu, Lin Tan, Xiangyu Zhang, and Petr Babkin. 2025. Nova: Generative Language Models for Assembly Code with Hierarchical Attention and Contrastive Learning. In *Proc. of the International Conference on Learning Representations (ICLR)*.

[27] Alex Krizhevsky, Ilya Sutskever, and Geoffrey E Hinton. 2012. ImageNet Classification with Deep Convolutional Neural Networks. In *Advances in Neural Information Processing Systems 25 (NeurIPS)*.

[28] Per Larsen, Andrei Homescu, Stefan Brunthaler, and Michael Franz. 2014. SoK: Automated Software Diversity. In *Proc. of the IEEE Symposium on Security and Privacy (S&P)*.

[29] Vladimir I. Levenshtein. 1966. Binary codes capable of correcting deletions, insertions, and reversals. In *Soviet physics doklady*, Vol. 10. 707–710.

[30] Xuezixiang Li, Yu Qu, and Heng Yin. 2021. PalmTree: Learning an Assembly Language Model for Instruction Embedding. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*.

[31] Xiang Ling, Lingfei Wu, Wei Deng, Zhenqing Qu, Jiangyu Zhang, Sheng Zhang, Tengfei Ma, Bin Wang, Chunming Wu, and Shouling Ji. 2022. MalGraph: Hierarchical Graph Neural Networks for Robust Windows Malware Detection. In *Proc. of the IEEE Conference on Computer Communications (INFOCOM)*.

[32] Cullen Linn and Saumya K. Debray. 2003. Obfuscation of Executable Code To Improve Resistance to Static Disassembly. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*.

[33] Zhenhao Luo, Pengfei Wang, Baosheng Wang, Yong Tang, Wei Xie, Xu Zhou, Danjun Liu, and Kai Lu. 2023. VulHawk: Cross-architecture Vulnerability Detection with Entropy-based Binary Code Search. In *Proc. of the Network and Distributed System Security Symposium (NDSS)*.

[34] Alwin Maier, Felix Weißberg, and Konrad Rieck. 2024. On the Role of Pre-trained Embeddings in Binary Code Analysis. In *Proc. of the ACM Asia Conference on Computer and Communications Security (AsiaCCS)*.

[35] Alessandro Mantovani, Luca Compagna, Yan Shoshitaishvili, and Davide Balzarotti. 2022. The Convergence of Source Code and Binary Vulnerability Discovery - A Case Study. In *Proc. of the ACM Asia Conference on Computer and Communications Security (AsiaCCS)*.

[36] Andrea Marcelli, Mariano Graziano, Xabier Ugarte-Pedrero, Yanick Fratantonio, Mohamad Mansouri, and Davide Balzarotti. 2022. How Machine Learning Is Solving the Binary Function Similarity Problem. In *Proc. of the USENIX Security Symposium*.

[37] Luca Massarelli, Giuseppe Antonio Di Luna, Fabio Petroni, Leonardo Querzoni, and Roberto Baldoni. 2019. SAFE: Self-Attentive Function Embeddings for Binary Similarity. In *Proc. of the Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA)*.

[38] Niall McLaughlin and Jesus Martinez Del Rincon. 2022. Data Augmentation for Opcode Sequence Based Malware Detection. In *Proc. of the Cyber Research Conference - Ireland (Cyber-RCI)*.

[39] Vasilis Pappas, Michalis Polychronakis, and Angelos D. Keromytis. 2012. Smashing the Gadgets: Hindering Return-Oriented Programming Using In-place Code Randomization. In *Proc. of the IEEE Symposium on Security and Privacy (S&P)*.

[40] Kexin Pei, Jonas Guan, Matthew Broughton, Zhongtian Chen, Songchen Yao, David Williams-King, Vikas Ummadisetty, Junfeng Yang, Baishakhi Ray, and Suman Jana. 2021. StateFormer: Fine-Grained Type Recovery From Binaries Using Generative State Modeling. In *Proc. of the ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*.

[41] Kexin Pei, Jonas Guan, David Williams-King, Junfeng Yang, and Suman Jana. 2021. XDA: Accurate, Robust Disassembly with Transfer Learning. In *Proc. of the Network and Distributed System Security Symposium (NDSS)*.

[42] Jannik Pewny, Behrad Garmany, Robert Gawlik, Christian Rossow, and Thorsten Holz. 2015. Cross-Architecture Bug Search in Binary Executables. In *Proc. of the IEEE Symposium on Security and Privacy (S&P)*.

[43] Davide Pizzolotto and Katsuro Inoue. 2021. Identifying Compiler and Optimization Level in Binary Code From Multiple Architectures. *IEEE Access* 9 (2021), 163461–163475.

[44] Adrian Rosebrock. 2017. *Deep Learning for Computer Vision with Python.* PyImageSearch.

[45] Lukas Seidel, Sam L. Thomas, and Konrad Rieck. 2026. Practical Type Inference: High-Throughput Recovery of Real-World Structures and Function Signatures. In *Proc. of the Sixteenth ACM Conference on Data and Application Security and Privacy (CODASPY)*.

[46] Connor Shorten and Taghi M Khoshgoftaar. 2019. A Survey on Image Data Augmentation for Deep Learning. *Journal of Big Data* 6 (2019), 1–48.

[47] Alexander Shroyer and D. Martin Swany. 2022. Data Augmentation for Code Analysis. In *Proc. of the International Conference on Intelligent Data Science Technologies and Applications (IDSTA)*.
[48] Benjamin Steenhoek, Md Mahbubur Rahman, Richard Jiles, and Wei Le. 2023. An Empirical Study of Deep Learning Models for Vulnerability Detection. In *Proc. of the International Conference on Software Engineering (ICSE)*.
[49] Zian Su, Xiangzhe Xu, Ziyang Huang, Zhuo Zhang, Yapeng Ye, Jianjun Huang, and Xiangyu Zhang. 2024. CodeArt: Better Code Models by Attention Regularization When Symbols Are Lacking. *Proc. of the ACM on Software Engineering (FSE)* 1 (2024), 562–585.
[50] Richard Szeliski. 2010. *Computer Vision: Algorithms and Applications*. Springer.
[51] Freek Verbeek, Joshua A. Bockenek, Zhoulai Fu, and Binoy Ravindran. 2022. Formally verified lifting of C-compiled x86-64 binaries. In *Proc. of the ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI)*.
[52] Robert A. Wagner and Michael J. Fischer. 1974. The String-to-String Correction Problem. *Journal of the ACM (JACM)* (1974).
[53] Richard Wartell, Vishwath Mohan, Kevin W. Hamlen, and Zhiqiang Lin. 2012. Binary Stirring: Self-Randomizing Instruction Addresses of Legacy x86 Binary Code. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*.
[54] Jason Wei and Kai Zou. 2019. EDA: Easy Data Augmentation Techniques for Boosting Performance on Text Classification Tasks. In *Proc. of the Conference on Empirical Methods in Natural Language Processing (EMNLP)*.
[55] Michael D. Wong, Edward Raff, James Holt, and Ravi Netravali. 2023. Marvolo: Programmatic Data Augmentation for Deep Malware Detection. In *Proc. of the Machine Learning and Knowledge Discovery in Databases: Research Track - European Conference (ECML PKDD)*.
[56] Dongpeng Xu, Jiang Ming, and Dinghao Wu. 2017. Cryptographic Function Detection in Obfuscated Binaries via Bit-Precise Symbolic Loop Mapping. In *Proc. of the IEEE Symposium on Security and Privacy (S&P)*.
[57] Xiaojun Xu, Chang Liu, Qian Feng, Heng Yin, Le Song, and Dawn Song. 2017. Neural Network-based Graph Embedding for Cross-Platform Binary Code Similarity Detection. In *Proc. of the ACM Conference on Computer and Communications Security (CCS)*.
[58] Khaled Yakdan, Sebastian Eschweiler, Elmar Gerhards-Padilla, and Matthew Smith. 2015. No More Gotos: Decompilation Using Pattern-Independent Control-Flow Structuring and Semantic-Preserving Transformations. In *Proc. of the Network and Distributed System Security Symposium (NDSS)*.
[59] Jiaqi Yan, Guanhua Yan, and Dong Jin. 2019. Classifying Malware Represented as Control Flow Graphs using Deep Graph Convolutional Neural Network. In *Proc. of the annual IEEE/IFIP international conference on dependable systems and networks (DSN)*.
[60] Shiwen Yu, Ting Wang, and Ji Wang. 2022. Data Augmentation by Program Transformation. *Journal of Systems and Software* 190 (2022).
[61] Kunsong Zhao, Zihao Li, Jianfeng Li, He Ye, Xiapu Luo, and Ting Chen. 2023. DeepInfer: Deep Type Inference from Smart Contract Bytecode. In *Proc. of the ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE)*.
[62] Chang Zhu, Ziyang Li, Anton Xue, Ati Priya Bajaj, Wil Gibbs, Yibo Liu, Rajeev Alur, Tiffany Bao, Hanjun Dai, Adam Doupé, Mayur Naik, Yan Shoshitaishvili, Ruoyu Wang, and Aravind Machiry. 2024. TYGR: Type Inference on Stripped Binaries using Graph Neural Networks. In *Proc. of the USENIX Security Symposium*.

A Generative AI Usage

Generative AI tools were used during the preparation of this work to assist in organizing the research code into a structured software library and to support the code documentation. All AI-assisted code and documentation were reviewed, tested, validated, and, where necessary, corrected by the authors to ensure their accuracy and consistency with the implementation. Generative AI tools were also used to improve the grammar, spelling, and fluency of the manuscript. All AI-assisted revisions were reviewed and edited by the authors, who take full responsibility for the accuracy, correctness, and integrity of the final manuscript and all associated research artifacts. No AI tools were used to generate research data, conduct data analysis, or produce results underlying the conclusions of this work beyond the programming assistance described above.

B Implementation Details

Our framework defines transformations through edit operations and constraints, leaving the implementation open. Instruction substitution, for example, can draw from a table of semantically equivalent alternatives, use binary rewriting, or take any other route that satisfies the same constraints. In the following, we describe our own implementation, summarized in Table 4. We satisfy the constraints heuristically to keep the runtime overhead low.

NOP insertion, truncation, and instruction erasing satisfy their constraints by construction. Instruction substitution resorts to a set of semantically equivalent replacements for common mnemonics and samples replacement sequences with register, memory, and immediate variants. Instruction insertion samples from a predefined set of idempotent one- and two-instruction sequences parameterized by registers, addresses, and immediates. Dead code insertion generates a random instruction sequence and places an unconditional jump immediately before it, targeting the first instruction after the sequence. Instruction swapping applies a shallow register-level check: two instructions count as data-dependent and thus not swapped, if one writes a register the other accesses. Basic block reordering recovers basic blocks and permutes their positions.

Table 4: Overview of the implementation of the constraints.

Constraint	Implementation
Instruction substitution	
$s_i \sim z$ $l = 1$	Replacements are drawn from a table of specified equivalent alternatives, so z satisfies the constraint by construction.
Instruction insertion	
$z \sim \langle a_{\text{nop}} \rangle$	Sequences z are sampled from a predefined set of sequences equivalent to a no-operation and are therefore semantics-preserving by construction.
Dead code	
$z_1 \prec s_{\text{i+1}}$	The sequence z is prepended with a a_{jmp} instruction, targeting $s_{\text{i+1}}$ and thus satisfies the constraint by construction.
NOP insertion	
$z = \langle a_{\text{nop}} \rangle$	a_{nop} instructions are inserted at random places in the sequences.
Truncation	
$i = 1$	Instructions are deleted from the start and end of s.
Instruction erasing	
$i \leq n - l$	Instructions are deleted at a random index in s.
Instruction swapping	
$s_i \perp s_j$ $l_i = l_j = 1$	We analyze each pair's register usage $\langle a_i, a_{i+1} \rangle$ and swap only those without data hazards.
Basic block reordering	
$s_i \prec s_{i+l_i}$ $s_j \prec s_{j+l_j}$	These constraints always hold within basic blocks, which we recover through a control-flow graph reconstruction of each function.