Exploiting Code Symmetries for Learning Program Semantics

Kexin Pei [1 2] **Weichen Li** [1] **Qirui Jin** [3] **Shuyang Liu** [4] **Scott Geng** [5]
Lorenzo Cavallaro [6] **Junfeng Yang** [1] **Suman Jana** [1]

Abstract

This paper tackles the challenge of teaching code semantics to Large Language Models (LLMs) for program analysis by incorporating code symmetries into the model architecture. We introduce a group-theoretic framework that defines code symmetries as semantics-preserving transformations, where forming a code symmetry group enables precise and efficient reasoning of code semantics. Our solution, SYMC, develops a novel variant of self-attention that is provably equivariant to code symmetries from the permutation group defined over the program dependence graph. SYMC obtains superior performance on five program analysis tasks, outperforming state-of-the-art code models, including GPT-4, without any pre-training. Our results suggest that code LLMs that encode the code structural prior via the code symmetry group generalize better and faster.

1. Introduction

Automated program analysis using Large Language Models (LLMs) has become widely popular for software engineering and security tasks (Liu et al., 2023; Maniatis & Tarlow, 2023), but it remains unclear whether code LLMs can stay robust and generalize to new code (Henke et al., 2022; Rabin et al., 2021; Gao et al., 2023b;a; Yefet et al., 2020; Bundt et al., 2022; Zhang et al., 2023). This paper aims to enhance LLMs by establishing and preserving fundamental code symmetries, drawing inspiration from translation and rotation symmetries that typically hold in vision.

Code symmetry. Intuitively, symmetry of code refers to any transformation applied to a code block that preserves the semantics (i.e., input-output behavior) of the original code. Consider a (sequential) code fragment `x=2;y=4`. Reordering the instructions to `y=4;x=2` does not change the semantics of the code. Of course, any code analysis task that depends solely on the semantics of the code (e.g., bug detection) needs to preserve these symmetries by staying invariant to the transformations. Otherwise, if a bug detector flips its prediction from correct to buggy due to such simple semantics-preserving permutations, developers will lose confidence in the tool (Bessey et al., 2010). Formally, given a code block c and a set of symmetries G, an LLM m should ensure $\forall g \in G, m(g(c)) = m(c)$. Incorporating such an invariant property in the model has proven an effective approach in many domains to enforce domain-specific rules and improve generalization (Cohen & Welling, 2016).

Limitations of existing approaches. A popular way to train LLMs to be robust to code symmetries is via large-scale pre-training, where the pre-training dataset likely includes many semantically equivalent code samples (Roziere et al., 2023). While this approach improves the generalization, it is inherently a best-effort attempt and does not guarantee invariance for the pre-trained model. Without the guarantee, the trained model lacks assurance when deployed for program analysis (Ullah et al., 2023), e.g., the malware can be easily transformed to evade detection. In fact, we find that state-of-the-art LLMs break desired invariances at an alarmingly high rate, e.g., 18% in CodeLlama in predicting function names (Table 1) even for simple code symmetries like a two-statement permutation.

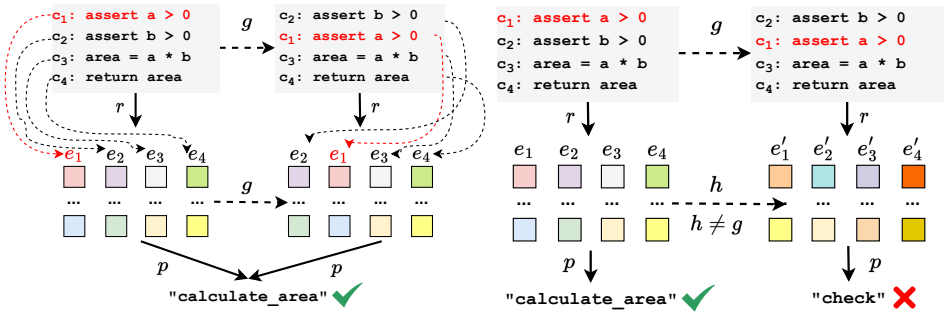
A more direct approach is to explicitly enumerate the code symmetries via data augmentation. However, it is prohibitively expensive due to the sheer number of possible symmetries and their compositions. Similar to pre-training, even exhausting the code symmetries in the augmented training set does not provide any guarantee that the trained model stays invariant to the observed code symmetries. An alternative strategy involves approximating code symmetry structures as priors within the model's architecture, e.g., data and control flow graphs, using Graph Neural Nets (GNNs) (Allamanis et al., 2017). Such approaches are argued to offer a better generalization, e.g., as evidenced in Table 1 where GGNN has the lowest violation rate among the baselines. However, graph architectures often restrict a model's expressiveness relative to LLM architectures (Ying et al., 2021). Moreover, the existing common practice of encoding code structures into GNNs does not explicitly preserve code symmetries and thus lacks guarantee

[1]Columbia University [2]The University of Chicago [3]University of Michigan [4]Huazhong University of Science and Technology [5]University of Washington [6]University College London. Correspondence to: Kexin Pei<kpei@cs.uchicago.edu>, Suman Jana <suman@cs.columbia.edu>.

Proceedings of the 41st International Conference on Machine Learning, Vienna, Austria. PMLR 235, 2024.

Table 1: Invariance violation rate across different code models (darker colors indicate more violations).

	Violation
SYMC (Ours)	**0%**
code2vec	61%
code2seq	52%
CodeLlama	18%
CodeT5	16%
DOBF	41%
GGNN	7%
GPT-4	43%
GraphCodeBERT	31%
WizardCoder	14%



(a) **SYMC (Ours)** (b) Existing code LLMs

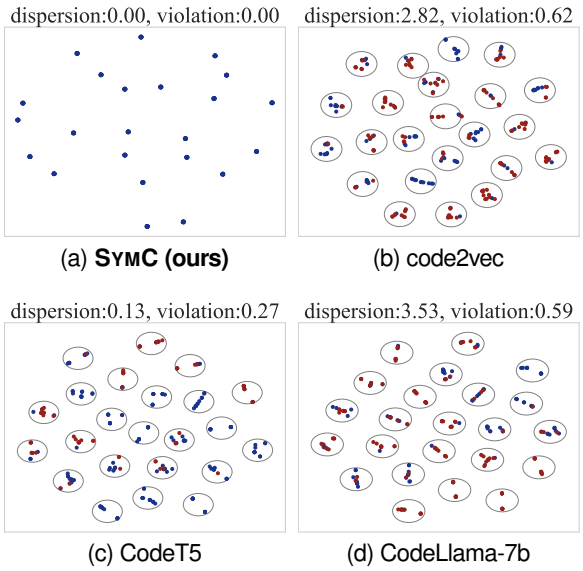
Figure 1: (a) SYMC as a G-invariant function name predictor where G is a group of semantics-preserving statement permutations g. (b) Code LLMs not preserving the symmetries in G and thus mispredict the label.

that the code symmetries are indeed preserved (see §6).

Our approach. In this paper, we investigate how to modify expressive architecture like Transformers to provably impose semantic priors like code symmetries while still preserving the learning capacity. We introduce a group-theoretic framework to precisely define code symmetries in terms of semantics-preserving statement permutations and create LLM architectures that inherently preserve these symmetries *by construction*. Importantly, such a group-theoretic framework expresses semantic priors in a syntax-agnostic way while being amenable to be encoded in neural architectures (Cohen & Welling, 2016; Romero & Cordonnier, 2020). This makes symmetries an ideal option to represent program semantics as the inductive bias for code LLM architectures.

Using this framework, we present SYMC, a variant of LLM architecture designed to *provably guarantee* invariance to semantics-preserving statement permutations. This is achieved through a G-equivariant code representation (r) followed by a G-invariant prediction (p), with G determined based on the graph automorphisms of the code block's interpretation graph (a generalization of program dependence graph).

Figure 1a shows a concrete example of the benefit of SYMC when deployed for function name prediction. The code snippets illustrate a semantics-preserving statement reordering. SYMC enforces its output to stay invariant via keeping its learned representation G-equivariant, where the code representation (e_1,e_2,e_3,e_4) is transformed into (e_2,e_1,e_3,e_4), followed by a G-invariant prediction module. By contrast, Figure 1b shows an existing code model (Jin et al., 2022) that does not preserve permutation symmetry. In this case, the code representation $(e_1,...,e_4)$ is transformed into a completely different set of embeddings $(e'_1,...,e'_4)$, leading to a changed prediction. In fact, the t-SNE visualization in Figure 2 shows that the learned code embeddings of existing code models are highly dispersed when the code is under semantics-preserving permutations, and a large fraction of the permuted samples have their labels mispredicted.



(a) **SYMC (ours)** (b) code2vec

(c) CodeT5 (d) CodeLlama-7b

Figure 2: Each cluster represents the learned embeddings of a code block and its semantics-preserving permuted versions. The cluster's dispersion (variances to the mean) indicates that the permutation changes the embeddings, while the color turning from blue to red indicates the changed predictions. We take the mean of SYMC's embedding so it becomes permutation-invariant (§3.5).

Result summary. We evaluate SYMC on five program analysis tasks against 12 source code and binary analysis baselines. For semantics-preserving permutations, SYMC is guaranteed to stay invariant while the state-of-the-art code LLMs, e.g., CodeLlama, violate the invariance by 31.4% on average. As a result, SYMC outperforms the extensively pre-trained baselines by up to 67.5%, even though SYMC is only trained

from scratch *without requiring any pre-training*. For unseen semantics-preserving source code transformations beyond permutations, SYMC surpasses the state-of-the-art code baselines by up to 30.8%, while maintaining 104.8× smaller model size. On sophisticated code transformations introduced by compiler optimizations and obfuscations, SYMC outperforms the extensively pre-trained binary analysis baseline by 30.7%.

2. Preliminaries

This section briefly describes the symmetry groups. See Appendix A for a more formal description.

A *symmetry group* $(G,\circ)$ consists of a non-empty set G of transformations and a binary operator $\circ : G \times G \to G$, where $\circ$ operates on two elements in G, e.g., $x,y \in G$, and produces a new transformation $z = x \circ y, z \in G$. The binary operator has to be associative, invertible, and there exists an identity $\exists \mathbf{1} \in G, \forall x \in G, x \circ \mathbf{1} = \mathbf{1} \circ x$.

The elements of a G are abstract transformations that become concrete when they *act* on some set X, i.e., they transform $x \in X$ into $x' \in X$ while keeping some properties of x *invariant*. Formally, an action $\bullet$ of a G is a binary operation defined on a set of objects X, i.e., $\bullet : G \times X \to X$, where it is also associative and has an identity.

It is common in the group theory literature to use $\circ$ to denote both *action* and *composition*, when it is clear from the context (Higgins et al., 2018). It is also customary to interchange $g(x)$ and $g \circ x$. Therefore, we treat $g \bullet (h \bullet x)$, $g \circ (h \circ x)$, and $g(h(x))$ as the same in the rest of this paper.

A symmetry group comes with two properties, namely *invariance* and *equivariance*, that formalize the concept of preservation of some properties when a set X is acted upon by G. Let f be a function that maps each element $x \in X$ to a corresponding element y in the set Y, indicating the property's value for that particular element. f is called G**-invariant** if $\forall g \in G, \forall x \in X, f(g \circ x) = f(x)$. f is called G**-equivariant** if $\forall g \in G, \forall x \in X, f(g \circ x) = g \circ f(x)$.

3. Method

This section describes the construction of group-equivariant self-attention layers and the group-invariant code model.

3.1. Invariance & Equivariance for Code Models

Code representation units. We establish formal definitions of the code space as a collection of code blocks, which serve as the input space for representation learning.

Definition 3.1. A **code representation unit** (e.g., procedure) c consists of n instructions from an instruction set I, i.e., $c \in I^n$. The **code space** I^n is the set of all code representation units of interest.

A typical Code Representation Unit (CRU) is a method with well-defined interfaces, ensuring controlled interaction with other methods, without arbitrary control transfers. Below, we provide formal definitions for learning program representation and predictive learning.

We establish formal properties for code analysis models with explicit representation learning r and predictive learning p based on G-equivariance/invariance.

Definition 3.2 (G**-equivariant code representation learning).** Let G be a symmetry group consisting of *semantics-preserving transformations* applied to a CRU $c \in I^n$. A representation function $r : I^n \to \mathbb{R}^{d \times n}$ is G-equivariant if for every $g \in G$ and $c \in I^n$, we have $g \circ r(c) = r(g \circ c)$ (d denotes the dimension of the embedding to which each instruction is mapped).

Note that here the input space of r (I^n) and its output space ($\mathbb{R}^{d \times n}$) are both sets of size n, where each instruction I is mapped to a R^d vector by the representation function. This consideration is necessary to ensure the symmetry group can act on both sets appropriately.

Definition 3.3 (G**-invariant code predictive learning).** Let G be a symmetry group consisting of *semantics-preserving transformations* applied to program representation vector $c \in I^n$. A predictive learning function $p : \mathbb{R}^{d \times n} \to \mathbb{R}^L$ is G-invariant if $\forall g \in G, \forall e \in \mathbb{R}^{d \times n}$, $p(g \circ e) = p(e)$.

Stacking p on top of r, $p \circ r$, leads to a G-invariant model according to Lemma 5.

3.2. Semantics-Preserving Program Symmetries

A code symmetry is a program transformation that preserves the input-output behavior of a CRU when interpreted by the program interpretation function f. The program interpretation function takes a CRU $c \in I^n$ as input. We use $\mathcal{I}$ to represent the set of all input values to execute CRU, and produce output values represented by the set $\mathcal{O}$.

Definition 3.4. A **semantics-preserving code symmetry** g is a transformation acting on $c \in I^n$ ($g : I^n \to I^n$) such that $\forall in \in \mathcal{I}, \forall out \in \mathcal{O}, f(g \circ c, in) = f(c, in) = out$.

Definition 3.5. A semantics-preserving **program symmetry group** G is a set of semantics-preserving program symmetries that also satisfy the group axioms.

3.3. $Aut(\mathcal{IG})$: A Program Symmetry Group

In this paper, we focus on a specific symmetry group that maintains the structural integrity of CRUs by utilizing their inherent compositional structure. However, note that this approach is not the only way to form code symmetry groups and does not encompass all possible code symmetries. We leave further exploration in these directions to future work. Next, we describe the compositional structure of the program interpreter f operating on a CRU, enabling us to define the program interpretation

graph that links CRUs to their input-output behavior.

Compositional structure of program interpreter f. The interpreter function f (defined in §3.2) can be represented as a composition of individual per-instruction interpreter functions $\{f_1,...,f_n\}$. Each $f_i : \mathcal{I}_i \rightarrow \mathcal{O}_i$ interprets a single instruction c_i from the instruction set I (Definition 3.1), takes the input values $in_i \in \mathcal{I}_i$, and produce the output values $out_i \in \mathcal{O}_i$. The output of f_i can include both data flow elements (e.g., variables or memory locations with values assigned by f_i) and control flow elements (e.g., addresses of next interpreter functions $f_j \in f$ assigned by f_i). Consequently, we can express f as the composition of different individual interpreters, i.e., $f_n \circ ... \circ f_1$, where later instructions act on the output of previous instructions.

Program interpretation graph ($\mathcal{IG}$). Programs often involve different control flow paths, such as if-else statements, leading compositions between individual interpreter functions to a directed graph instead of a linear sequence. This graph is referred to as the program interpretation graph. For a given CRU c, there can be multiple execution paths, each exercising different subsets of $\{f_1,...,f_n\}$.

To construct the interpretation graph $\mathcal{IG} = (V,E)$, we consider all feasible execution paths of c. In $\mathcal{IG}$, each node $V_i \in V$ corresponds to f_i, and each directed edge $E_{i,j} \in E$ (connecting V_i to V_j) represents at least one execution path where f_j takes the output of f_i as input, i.e., $E_{i,j} = (out_i, in_j)$.

Automorphism group of interpretation graph. Our objective is to find a group of symmetries that act on c while preserving its input and output behavior as interpreted by f in terms of $\mathcal{I}$ and $\mathcal{O}$ (Definition 3.4). Intuitively, as $\mathcal{IG}$ represents all execution paths of c, any transformations that preserve $\mathcal{IG}$ should also preserve the execution behavior of c. Therefore, we aim to uncover a group of symmetries that preserve $\mathcal{IG}$ (Theorem 1), and such a group can guide us to construct code analysis model that can stay invariant to all symmetries of the group (§3.4).

To achieve this, we consider a specific set of symmetries called the *automorphisms* of $\mathcal{IG}$, denoted as $Aut(\mathcal{IG})$. An automorphism is a group of symmetries $\sigma \in Aut(\mathcal{IG})$ that act on the interpretation graph $\mathcal{IG} = (V,E)$. Intuitively, graph automorphisms can be thought of as permutations of nodes that do not change the connectivity of the graph. $Aut(\mathcal{IG})$ is formally defined as follows:

Definition 3.6 ($\mathcal{IG}$ Automorphism)**.** $\mathcal{IG}$ automorphism is a group of symmetries $\sigma \in Aut(\mathcal{IG})$ acting on an interpretation graph $\mathcal{IG} = (V,E)$, where σ is a bijective mapping: $\sigma : V \rightarrow V$, such that for every edge $E_{i,j} \in E$, i.e., connecting f_i and f_j, there is a corresponding edge $(\sigma(f_i), \sigma(f_j)) \in E$.

We now show how the automorphism $\sigma \in Aut(\mathcal{IG})$ preserves all input and output behavior of $\{f_1, ..., f_n\}$ in the space of $\mathcal{I}$ and $\mathcal{O}$. As mentioned earlier, graph automorphism is a permutation on the set of nodes in $\mathcal{IG}$ such that the edges $E_{i,j} = (out_i, in_j)$ are preserved in the transformed $\mathcal{IG}'$. As each $f_i \in \{f_1,..,f_n\}$ operates on $c_i \in c$, we have the following (see Appendix B for the proof):

Theorem 1. The set of automorphisms $\sigma \in Aut(\mathcal{IG})$ forms a program symmetry group.

3.4. $Aut(\mathcal{IG})$-Equivariant Code Representation

Existing program analyses using Transformer typically involve an embedding layer followed by applying l self-attention layers A^l. A prediction head F is then placed on top of A^l for downstream analysis tasks. We can thus consider the representation learning r as the composition of the embedding layer and A^l, with F as the predictive learning p (§3.1). We now present the development of a new self-attention layer that is $Aut(\mathcal{IG})$-equivariant.

Self-attention. The standard self-attention computation can be succinctly represented as $w_v \cdot s(w_k^T \cdot w_q)$, where w_v, w_k, and w_q are learnable parameters for transforming value, key, and query, respectively, and $s(\cdot)$ represents scaling by $\sqrt{d}$ and applying Softmax (see Appendix A).

It is easy to show that the existing self-attention layer is equivariant to permutations (Appendix B). However, we want to make the self-attention layers equivariant *only* to $Aut(\mathcal{IG})$, not *all permutations*. In the following, we describe how to build $Aut(\mathcal{IG})$-equivariant self-attention.

Biasing self-attention with a distance matrix. To build $Aut(\mathcal{IG})$-equivariant self-attention layers, denoted as $\mathcal{GA}$, we add a customized distance matrix $d_{\mathcal{IG}}$ to $\mathcal{GA}$: $GA(e) = w_v e \cdot s(w_k e^T \circ w_q e + d_{\mathcal{IG}})$. Such a distance matrix is a superset of the adjacency matrix of $\mathcal{IG}$, encoding a richer topology structure of the graph. We relax the definition of distance matrix $d_{\mathcal{IG}}$ here to be no longer symmetrical, as long as it satisfies the following two properties: (1) $d_{\mathcal{IG}}$ stays invariant when $\sigma \in Aut(\mathcal{IG})$ acts on $\mathcal{IG}$: $d_{\mathcal{IG}} = \sigma(d_{\mathcal{IG}})$, and (2) $d_{\mathcal{IG}}$ commutes with permutation matrix p_σ ($\sigma \in Aut(\mathcal{IG})$).

We will describe a concrete instantiation of $d_{\mathcal{IG}}$ in §4.2. Based on the two properties, we have the following Theorem (with its proof in Appendix B.

Theorem 2. Self-attention $GA(e) = w_v e \cdot s(w_k e^T \cdot w_q e + d_{\mathcal{IG}})$ is $Aut(\mathcal{IG})$-equivariant.

As the embedding layer is trivially $Aut(\mathcal{IG})$-equivariant, composing it with $Aut(\mathcal{IG})$-equivariant self-attention layers remains $Aut(\mathcal{IG})$-equivariant (Lemma 4).

3.5. $Aut(\mathcal{IG})$-Invariant Predictor

We describe two prediction modules that are inherently $Aut(\mathcal{IG})$-invariant, so stacking them on top of the $Aut(\mathcal{IG})$-equivariant self-attention layers leads to an $Aut(\mathcal{IG})$-invariant code model (Lemma 5).

Token-level. Token-level predictor is often employed when

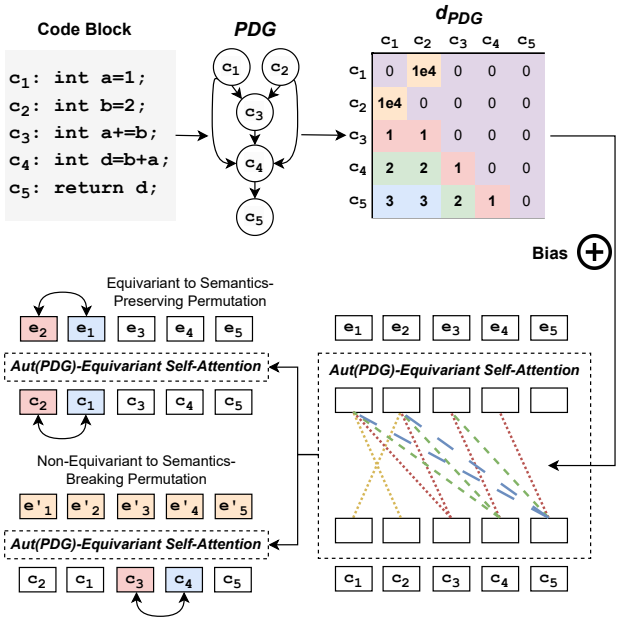


Figure 3: Simplified SYMC architecture, which takes as input the code block and its program dependence graph (PDG) to construct the $Aut(PDG)$-equivariant self-attention head.

each input token needs a label, e.g., predicting memory region per instruction (§5). As the automorphism acts on the input sequence e but not individual tokens, i.e., the value of the embedding vectors, the automorphism σ does not apply to the query vector q_i (§3.4). Therefore, we have Lemma 1. See Appendix B for complete proof.

Lemma 1. The biased self-attention computing the embedding $e'_i = GA(e_i)$ is $Aut(\mathcal{IG})$-invariant.

Pooling-based. Another popular $Aut(\mathcal{IG})$-invariant predictor involves pooling the embedding sequence $e' = GA(e)$, e.g., using max or mean. Pooling operators are invariant to permutations, thus to $Aut(\mathcal{IG})$, e.g., the mean pooling $\mu(e') = (\Sigma_{i=1}^{n} e'_i)/n$ is not sensitive to the order of $(e'_1,...,e'_n)$. Pooling-based predictor is often employed when we aim to predict the property for the entire input sequence, e.g., predicting the function signature, detecting function similarity, etc. (§5).

4. SYMC Implementation

This section elaborates on the design choices to implement $Aut(\mathcal{IG})$-invariant code analysis. Figure 3 shows the simplified steps of biasing the self-attention layers to be equivariant to the semantics-preserving permutation group.

4.1. Relaxing $\mathcal{IG}$ to Program Dependence Graph

§3.4 demonstrated how to build $Aut(\mathcal{IG})$-equivariant self-attention layers. However, directly constructing $\mathcal{IG}$ is computationally impractical as we need to enumerate all possible execution paths. To address this, we consider *program dependence graph* (PDG), a sound over-approximation to $\mathcal{IG}$ that explicitly captures the control/data dependencies and can be computed statically and efficiently. PDG (V_{PDG}, E_{PDG}) is a super graph of $\mathcal{IG}$ due to the conservative data flow analysis, sharing the same vertices but having a superset of edges $(E_{PDG} \supseteq E_{\mathcal{IG}})$. Enforcing PDG to be a super graph of $\mathcal{IG}$ is crucial because a subgraph's automorphism group is a subgroup of the super graph's automorphism group $(Aut(PDG) \supseteq Aut(\mathcal{IG}))$. Thus, if the self-attention layer is $Aut(PDG)$-equivariant, it is guaranteed to be $Aut(\mathcal{IG})$-equivariant.

PDG construction. We construct PDG edges based on data and control dependencies between instructions. We consider three types of data dependencies: read-after-write, write-after-read, and write-after-write. Additionally, control dependencies are included to determine the execution order. These dependencies create a partial order of instructions, preventing permutations that would violate the edge directions and potentially alter the program's input-output behavior.(§3.3).

To identify the control and data dependencies, we employ a computationally efficient, conservative static analysis approach. This method is efficient and thus scalable to large datasets, as demonstrated in Appendix E.2 and Figure 11. For instance, we treat any two statements that access memory as dependent. However, this conservative approach may overlook potential symmetries. We aim to integrate more accurate analyses in the future, such as alias analysis through abstract interpretation or dynamic analysis, to reduce overapproximation while managing the trade-off of increased overhead. Interleaving standard dependency analysis with training/inference to minimize analysis overhead presents a promising direction for future exploration.

Implementing the static analysis to identify symmetries presents an additional challenge, as symmetries are defined at the level of tokens as seen by LLMs, whereas compilers and other program analysis tools typically perform data dependency analysis at the Intermediate Representation (IR) level. It requires significant engineering effort to correlate IR-level dependencies with source-level tokens and subtokens for the self-attention layers. Hence, for our initial prototype, we developed our analysis method based on the source-level ASTs (as described in §D.1) to simplify the mapping from the analysis results back to the source code tokens and subtokens. We note that integrating more sophisticated dependency analysis routines from existing compiler frameworks to identify symmetries is a promising area for future work.

4.2. Encoding Graph Structure

This section presents a concrete instance of the distance matrix defined on PDG, which enables us to prove $Aut(PDG)$-equivariance for the resulting self-attention layers.

Distance matrix. Let d denote the distance matrix of PDG

where d_{ij} represents the distance between nodes V_i and V_j. Each entry d_{ij} is a 2-value tuple (p_{ij}, n_{ij}), indicating the longest path from the lowest common ancestor of V_i and V_j, denoted as T_{ij}, to V_i and V_j, respectively. We call p_{ij} the positive distance and n_{ij} the negative distance.

We incorporate d into the Multi-Head Self-Attention (MHA) to ensure $Aut(PDG)$-equivariance with specific modifications to the attention heads to handle positive and negative distances. Particularly, the first half of the attention heads $MHA^i(e)$, for $i \in [1, h/2]$, are combined with the matrix dp formed by the positive distances in d (denoted as $dp_{ij} = p_{ij}$). The second half of the attention heads $MHA^i(e)$, for $i \in [h/2+1, h]$, are combined with the matrix dn formed by the negative distances in d (denoted as $dn_{ij} = n_{ij}$). The modified attention heads are defined as: (1) $MHA^i(e) = w_v e \cdot s(w_k e^T \cdot w_q e + dp), i \in [1, h/2]$, (2) $MHA^i(e) = w_v e \cdot s(w_k e^T \cdot w_q e + dn), i \in [h/2+1, h]$.

It is easy to show d satisfies the two properties defined in §3.4 (see Appendix B). We thus have:

Lemma 2. The distance matrix d of PDG remains invariant under the action of $\sigma \in Aut(PDG)$.

Lemma 3. The distance matrix d of PDG commutes with permutation matrix p_σ of the automorphism $\sigma \in Aut(PDG)$: $d \cdot p_\sigma = p_\sigma \cdot d$.

Based on these two properties, we can prove each head in MHA is $Aut(PDG)$-equivariant, following the same proof steps to Theorem 2. Therefore, according to Lemma 4, MHA composed by multiple $Aut(PDG)$-equivariant heads is also $Aut(PDG)$-equivariant.

5. Experimental Setup

This section briefly describes the evaluation tasks and code transformations. We put the detailed description, e.g., baselines, evaluation dataset and metrics, etc., in Appendix D.

Task selection. We consider *program analysis* tasks that take as input the program code and output different program properties (see below). These tasks require comprehending code semantics and behavior, so the model performing these tasks is expected to stay invariant to code symmetries. We *do not consider code generation tasks*, where the input to the program is natural language or other formal specifications. We leave the study of enforcing code symmetry constraints in the output as the future work.

Specifically, we consider (1) *function name prediction*, which performs an "extreme summarization" of the function behavior. (2) *defect prediction*, which detects whether a code block has an error. (3) *function similarity detection*, which predicts if a pair of functions are semantically similar; (4) *function signature prediction*, which predicts the types (`int`, `float`, etc.) of function arguments; and (5) *memory region prediction*, which predicts the memory region (stack, heap, etc.) that each memory-accessing instruction can possibly access. For (3)-(5), we focus on analyzing stripped binaries considering is broad applications in security, e.g., vulnerability detection and security retrofitting.

Code transformations. We consider a set of real-world semantics-preserving transformations beyond PDG automorphisms to evaluate SYMC's generalization by staying $Aut(PDG)$-equivariant. Instruction permutation occasionally forms the basis for these transformations. In particular, we consider two categories of binary code transformations: (1) *compiler optimizations* from GCC-7.5 and Clang-8, some of which reorder instructions for scheduling purposes (`-fdelayed-branch`, `-fschedule-insns`); and (2) *compiler-based obfuscations*, where we consider 5 obfuscations following Jin et al. (2022), such as control flow flattening, indirect branching, etc.

In addition to binary transformations, we consider six source code transformations following Rabin et al. (2021) and Wang et al. (2022): *variable rename* (VR) - changes the identifier names; *statement permute* (SP) – semantics-preserving permutations; *loop exchange* (LX) – switches `for` and `while`; *boolean exchange* (BX) – flips the boolean variables and negates all their uses by tracking their def-use chain; *unused statement* (US) – injects unused string declaration into a randomly chosen basic block; *switch to if* (SI) – transforms `switch` from/to `if` statements.

6. Evaluation

6.1. Invariance and Generalization

Evaluating $Aut(PDG)$-invariance. As SYMC is provably invariant to $Aut(PDG)$, we aim to study how other baselines perform under varying percentages of semantics-preserving statement permutations. Table 2 shows that all baselines, even having much larger model sizes and extensively pre-trained, are susceptible to slight permutations, i.e., with their prediction changed by 31.4% on average for function name prediction. On defect prediction, SYMC outperforms the pre-trained baselines by 11.4% on average; even we already fine-tuned the open-sourced ones, e.g., CodeT5, using the same samples and steps as SYMC, while SYMC is trained from scratch. Table 3 shows SYMC outperforms the state-of-the-art baseline, PalmTree, by 67.5% and remains robust across all semantics-preserving permutations.

Generalization to unseen transformations. Figure 4 shows that SYMC's performance on new samples introduced by unseen semantics-preserving transformations beyond permutations, outperforming the model specialized for this task, e.g., code2seq, by 30.8%. It also outperforms the two LLMs, GPT-4 and WizardCoder, by 16.1% and 1.63%, respectively. However, we observe that Code Llama outperforms SYMC by 10%, although SYMC achieves better results on statement permutation (SP). We attribute this to Code Llama's better understanding of natural language due

Table 2: Comparing SYMC to baselines against semantics-preserving permutations. We include the F1 score for both prediction tasks before and after the testing samples are permuted. The violation rate measures how many samples get their labels changed after permutation.

Model	Size	Before	After	Violate
Function Name Prediction				
SYMC (ours)	**68.4M**	**36.3**	**36.3**	**0%**
code2seq	6.3M	25.5	24.7	61%
code2vec	348M	17.7	19.6	52%
CodeLlama	7B	31.7	31.4	18%
CodeT5	770M	25.4	25.4	16%
DOBF	428M	16.3	20.1	41%
GGNN	53M	1.6	1.6	7%
GPT-4	N/A	30.3	30.7	43%
GraphCodeBERT	481M	20.8	20.6	31%
WizardCoder	3B	33.9	34.6	14%
Defect Prediction				
SYMC (Ours)	**67.7M**	**68.8**	**68.8**	**0%**
CodeBERT	476M	62.2	61.7	4.1%
CodeLlama	7B	51.03	51.39	3.4%
CodeT5	770M	63.3	60	6%
DOBF	428M	62.4	61.5	2.7%
GPT-4	N/A	51.56	50	13.5%
GraphCodeBERT	481M	61.7	61.7	1.3%
UnixCoder	504M	67.1	67.1	2.9%
WizardCoder	3B	49.24	49.24	6.8%

to its extensive pre-training on both code and text, which is especially beneficial for function name prediction. Nonetheless, SYMC retains the edge of having a much smaller model size (104.8×) without the need of any pre-training.

Besides the source-level transformations, we compare SYMC to baselines on more sophisticated code transformations introduced by unseen compiler optimizations and obfuscations. Figure 5 shows that SYMC outperforms PalmTree (see Appendix D) across all binary analysis tasks (we exclude memory region prediction as the dataset does not have this categorization) by 33.8% and 30.7% on seen and unseen transformations, respectively. While the compiler optimizations and obfuscations often involve more sophisticated transformations not directly related to instruction permutations, SYMC maintains its superior generalization.

6.2. Training Efficiency

Besides the improved robustness and generalization, SYMC is efficient in avoiding expensive training efforts, e.g., some may take up to 10 days (Jin et al., 2022). As shown in §6.1, SYMC, without any pre-training, outperforms the pre-trained baselines.

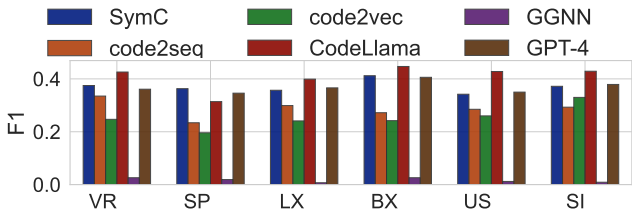


Figure 4: The performance (F1) of SYMC and baselines against different unseen code transformations defined in §5.

Therefore, in this section, we aim to study SYMC's performance under the limited training resources. Figure 6 shows that SYMC's performance (on memory region prediction) remains the highest when we reduce the model size and training iterations, outperforming PalmTree by 36.9% and 21.4%, respectively. Even in the most strict scenario, SYMC remains 38.2% and 15.3% better in both settings.

6.3. Ablations

Equivariance vs. Invariance. We compare the $Aut(PDG)$-*equivariant* self-attention layers to the $Aut(PDG)$-*invariant* ones, an alternative design choice to implement $Aut(PDG)$-*invariant* code models. Figure 7a shows that setting layers invariant early hinders prediction performance. SYMC with equivariant layers has an average 0.73 F1 across all training iterations and outperforms the second-best setting by 60.7%. This observation confirms the empirical findings that making earlier layers equivariant instead of invariant leads to better performance (Higgins et al., 2018).

Adding pre-training. We explore the impact of pre-training SYMC with masked language modeling (Devlin et al., 2018). We compare SYMC (without pre-training by default) to pre-trained versions (and then fine-tuned them for memory region prediction) with varying pre-training iterations. Figure 7b shows that pre-training with even one epoch results in a significantly improved F1 score, e.g., by 10.8%, with much faster convergence. However, additional pre-training epochs show diminishing returns, likely due to the limited training samples, e.g., the F1 score only improves by 3.2% with pre-training five epochs compared to 1 epoch.

Non-$Aut(PDG)$-equivariant baselines. We aim to confirm the performance of SYMC comes from preserving the symmetry, as opposed to the increased capacity from the distance matrix in the self-attention layers. To test this hypothesis, we compare SYMC with the baselines using the same exact architecture, but only changing the distance matrix whose entries are (1) kept exactly the same (i.e., fully permutation equivariant baseline), and (2) derived from the relative distance between the monotonically increasing positions (i.e., non-equivariant relative positional embedding).

Table 3: Comparing SYMC to binary analysis baselines against semantics-preserving permutations. Function signature and memory region prediction are measured in F1. Function similarity detection is measured using AUC (area under the ROC curve). Similar to Table 2, the magnitude of the violation rate is highlighted in red.

Model	Function Signature			Memory Region			Function Similarity		
	Before	After	Violate	Before	After	Violate	Before	After	Violate
SYMC	**0.88**	**0.88**	**0%**	**0.86**	**0.86**	**0%**	**0.96**	**0.96**	**0%**
PalmTree	0.59	0.41	24%	0.57	0.43	18%	0.72	0.69	31%
PalmTree-O	0.49	0.41	6%	0.57	0.44	11%	0.8	0.72	35%
PalmTree-N	0.19	0.41	86%	0.32	0.2	32%	0.71	0.72	38%

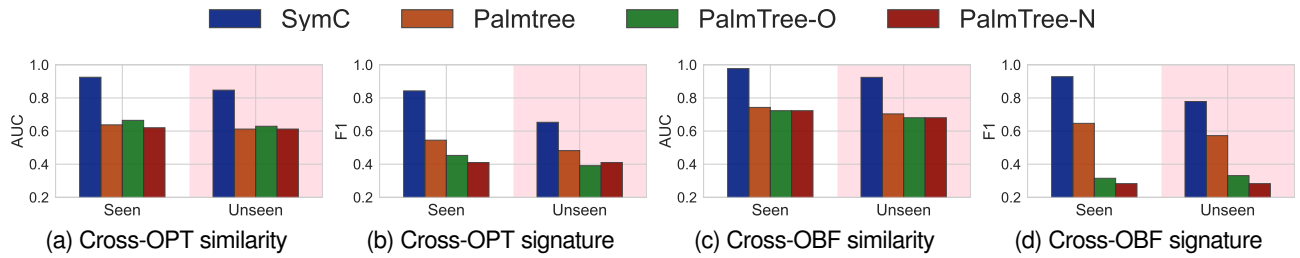


Figure 5: Evaluation on unseen optimization and obfuscation (marked in pink). We also include the testing results on *seen* optimizations and obfuscations (but the testing samples are non-overlapping with the training) on the left.

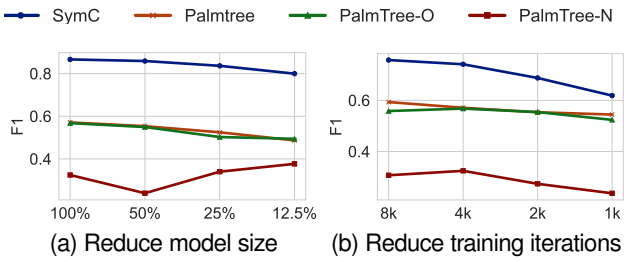


Figure 6: Comparing SYMC and baselines when we (a) reduce the model weights, and (b) reduce the number of training iterations, and observe how that affects the performance.

Figure 7c shows the comparison of testing F1 score by training the three models on memory region prediction tasks. Note that both baselines adopt the same shape of the distance matrix as SymC, so their number of parameters is identical with SYMC as well. The results demonstrate that SymC outperforms the fully permutation equivariant one and the non-equivariant one by 14.5% and 36.4%, respectively.

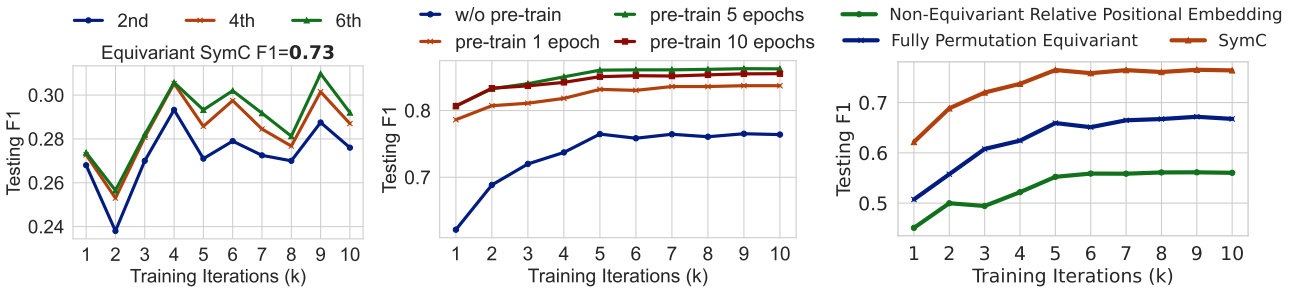
7. Limitations and Future Work

Overhead. While SYMC incurs additional overhead in generating PDG for each sample, we have demonstrated that our graph construction is relatively cheap (Appendix E.2, Figure 11), and we can interleave it with the training/inference to further improve the efficiency.

In essence, SYMC imposes the equivariance constraints *regardless of training*. This implies that a slight finetuning to help the model simply adapt to the symmetry constraints while specializing for downstream tasks would be likely enough (Basu et al., 2023). This is exactly why SYMC can obtain strong results by training the model from scratch using the typical fine-tuning tasks (not next token prediction or masked language modeling).

Other code symmetries. Our current framework focuses on the permutation group, but it extends to all transformations as long as they form a group. For example, variable name renaming is also a permutation over the entire vocabulary. SymC can also be readily applied to token permutation, e.g., `a=a+1` to `a=1+a` is a symmetry with a proper type inference, e.g., ensuring `a` is not a string and `+` is not a string concatenation operator.

However, the expressivity of the code symmetry can be restrictive to certain transformations. For example, insertion and deletion operations (e.g., deadcode elimination) are not invertible. Therefore, a new formalism, i.e., semigroup (Hille & Phillips, 1996) which relaxes the requirement of invertibility (§2), is required. Moreover, it is often an expensive manual effort to identify the symmetry group structure for arbitrary code transformations. For example, it is unclear how a compiler optimization pass, e.g., gcc O3, can be expressed using a sequence of insertion, deletion, and permutation operations. It would be an interesting future direction to uncover the unknown code symmetry from these transformations (Huh, 2024).



(a) Equivariant vs. invariant (b) Pre-training SYMC (c) Non-$Aut(PDG)$-equivariant baselines

Figure 7: (a) Setting SYMC's self-attention layers *invariant* in earlier layers. (b) Pre-training SYMC with varying pre-training epochs. (c) Comparing the $Aut(PDG)$-equivariant layers with the fully permutation equivariant ones (distance matrix with identical values) and non-equivariant ones (default relative position embeddings (Raffel et al., 2020; Wang et al., 2021)).

8. Related Work

Code representation learning. Previous research aims to automate software development tasks through code representation learning (Ding et al., 2023; Feng et al., 2020; Guo et al., 2022; Ahmad et al., 2021). Typical methodologies involve employing new model architectures and pre-training objectives with the goal of learning code representation that can be reused for other downstream program analysis and understanding tasks (Hellendoorn et al., 2019; Bieber et al., 2020; Pei et al., 2022; 2021; Allamanis et al., 2017; Sun et al., 2020; Peng et al., 2021; Kim et al., 2021; Guo et al., 2020). However, unlike our approach, these approaches do not provide any guarantees that the underlying models can be robust against any semantics-preserving transformations including simple permutations.

An alternative popular setting is to ground the program with its *execution behavior* so it learns semantics-aware code representations (Wen et al., 2024; Pei et al., 2022; Ding et al., 2023; Nye et al., 2021; Souza & Pradel, 2023; Wang & Su, 2020; Bieber et al., 2022; Ye et al., 2022). One caveat of incorporating execution in SYMC is that the dynamic traces are often incomplete to expose all possible code behaviors. Therefore, it might suffer from identifying false symmetries. However, as conservative static analysis can miss many true symmetries, dynamic traces can assist the analysis in canonicalizing transformations that are hard to reason about statically. In our current formulation based on PDG, we only consider symmetries for *all* input, but it can be relaxed to a subset of inputs, which is an exciting future work.

Symmetry in deep learning. Symmetry plays a crucial role in creating efficient neural architectures in various domains (Reiser et al., 2022; Wang et al., 2020; Bogatskiy et al., 2020; Perraudin et al., 2019; Cohen & Welling, 2016; Gordon et al., 2019; Dehmamy et al., 2021). Different architectures, such as CNNs, GNNs, and Transformers, leverage symmetry as the inductive bias to build model architectures robust to geometric transformations such as translations, rotations, permutations, etc. (Lee et al., 2019; Cohen & Welling, 2016; Esteves et al., 2018; Hutchinson et al., 2021; Gordon et al., 2019; Romero & Cordonnier, 2020). SYMC sets the first step to formalize code semantics learning using symmetry groups.

Recent studies have explored various strategies beyond symmetry groups to encode the geometric properties of input, aiming to enhance the model's robustness, generalization, and sample efficiency. These efforts include integrating symbolic operations into the neural architecture (Chaudhuri et al., 2021; Li et al., 2023; Kim et al., 2017; Hu et al., 2016) and designing specialized loss functions (Fort et al., 2019; Basu et al., 2023; Huh, 2024). Neural-symbolic approaches are particularly appealing for modeling programs, as programs typically encompass symbolic elements, such as syntactic and semantic rules, and fuzzy elements, such as natural language comments and function/variable names. However, developing differentiable components for symbolic rules might require extensive effort from the domain expert (Yi et al., 2018; Garcez et al., 2022). SYMC represents a new alternative to express program semantics in a way that is amenable to engineering the neural architecture, thanks to the rich literature in the domain of geometric deep learning (Bronstein et al., 2017). By representing program semantics as code symmetry groups, our approach holds the promise of enhancing inference efficiency by bypassing the interpretation of symbolic rules, which often face scalability challenges in traditional program analysis.

9. Conclusion

We studied code symmetries' impact on code LLM architectures for program reasoning. We introduced a novel self-attention variant that guarantees equivariance against code symmetries based on the permutation group. We implement SYMC and evaluate against various semantics-preserving transformations across different program analysis tasks, demonstrating the improved generalization in reasoning and analyzing programs.

Acknowledgement

We thank the anonymous reviewers for their constructive comments and feedback, which significantly improved this paper. This work was supported in part by NSF grants CNS-2154874, CNS-1564055, ONR grant N00014-17-1-2788, an NSF career award, a Google ASPIRE Award, multiple Google Cyber NYC awards, Columbia SEAS/EVPR Stimulus award, Columbia SEAS-KFAI Generative AI and Public Discourse Research award, and Accenture.

Impact Statement

Security-critical program analysis tools like those used for vulnerability detection and malware analysis have increasingly relied upon advanced machine learning, such as Large Language Models (LLMs), for improved efficiency, precision, and automation. However, incorrect predictions from learned code analysis impede the deployment and introduce more vulnerabilities in the deployed systems. This paper develops a new language to represent program semantics in a way that is amenable to being enforced in LLM architectures. Different from the philosophy of LLMs where all the rules are expected to be learned in an entirely data-driven manner, we want to emphasize the formal aspects of our approach to provably guarantee that the LLMs follow the precisely defined rules. Overall, this paper aims to advance generative AI technologies such as LLMs to make them trustworthy for high-assurance programming systems.

References

Ahmad, W. U., Chakraborty, S., Ray, B., and Chang, K.-W. Unified pre-training for program understanding and generation. *arXiv preprint arXiv:2103.06333*, 2021.

Allamanis, M., Peng, H., and Sutton, C. A convolutional attention network for extreme summarization of source code. In *International conference on machine learning*, pp. 2091–2100. PMLR, 2016.

Allamanis, M., Brockschmidt, M., and Khademi, M. Learning to represent programs with graphs. *arXiv preprint arXiv:1711.00740*, 2017.

Alon, U., Brody, S., Levy, O., and Yahav, E. code2seq: Generating sequences from structured representations of code. *arXiv preprint arXiv:1808.01400*, 2018.

Alon, U., Zilberstein, M., Levy, O., and Yahav, E. code2vec: Learning distributed representations of code. *Proceedings of the ACM on Programming Languages*, 3(POPL):1–29, 2019.

Arp, D., Quiring, E., Pendlebury, F., Warnecke, A., Pierazzi, F., Wressnegger, C., Cavallaro, L., and Rieck, K. Dos and don'ts of machine learning in computer security. In *31st USENIX Security Symposium (USENIX Security 22)*, pp. 3971–3988, 2022.

Basu, S., Sattigeri, P., Ramamurthy, K. N., Chenthamarakshan, V., Varshney, K. R., Varshney, L. R., and Das, P. Equi-tuning: Group equivariant fine-tuning of pretrained models. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 37, pp. 6788–6796, 2023.

Bessey, A., Block, K., Chelf, B., Chou, A., Fulton, B., Hallem, S., Henri-Gros, C., Kamsky, A., McPeak, S., and Engler, D. A few billion lines of code later: using static analysis to find bugs in the real world. *Commun. ACM*, 53:66–75, February 2010.

Bieber, D., Sutton, C., Larochelle, H., and Tarlow, D. Learning to execute programs with instruction pointer attention graph neural networks. *Advances in Neural Information Processing Systems*, 33:8626–8637, 2020.

Bieber, D., Goel, R., Zheng, D., Larochelle, H., and Tarlow, D. Static prediction of runtime errors by learning to execute programs with external resource descriptions. *arXiv preprint arXiv:2203.03771*, 2022.

Biggs, N., Biggs, N. L., and Norman, B. *Algebraic graph theory*. Number 67. Cambridge university press, 1993.

Bogatskiy, A., Anderson, B., Offermann, J., Roussi, M., Miller, D., and Kondor, R. Lorentz group equivariant neural network for particle physics. In *International Conference on Machine Learning*, pp. 992–1002. PMLR, 2020.

Bronstein, M. M., Bruna, J., LeCun, Y., Szlam, A., and Vandergheynst, P. Geometric deep learning: going beyond euclidean data. *IEEE Signal Processing Magazine*, 34(4): 18–42, 2017.

Bundt, J., Davinroy, M., Agadakos, I., Oprea, A., and Robertson, W. Black-box attacks against neural binary function detection. *arXiv preprint arXiv:2208.11667*, 2022.

Chaudhuri, S., Ellis, K., Polozov, O., Singh, R., Solar-Lezama, A., Yue, Y., et al. Neurosymbolic programming. *Foundations and Trends® in Programming Languages*, 7(3): 158–243, 2021.

Chua, Z. L., Shen, S., Saxena, P., and Liang, Z. Neural nets can learn function type signatures from binaries. In *26th USENIX Security Symposium*, 2017.

Cohen, T. and Welling, M. Group equivariant convolutional networks. In *International conference on machine learning*, pp. 2990–2999. PMLR, 2016.

Dehmamy, N., Walters, R., Liu, Y., Wang, D., and Yu, R. Automatic symmetry discovery with lie algebra convolutional network. *Advances in Neural Information Processing Systems*, 34:2503–2515, 2021.

Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

Ding, Y., Steenhoek, B., Pei, K., Kaiser, G., Le, W., and Ray, B. Traced: Execution-aware pre-training for source code. *arXiv preprint arXiv:2306.07487*, 2023.

Esteves, C., Allen-Blanchette, C., Makadia, A., and Daniilidis, K. Learning so (3) equivariant representations with spherical cnns. In *Proceedings of the European Conference on Computer Vision (ECCV)*, pp. 52–68, 2018.

Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., et al. Codebert: A pre-trained model for programming and natural languages. *arXiv preprint arXiv:2002.08155*, 2020.

Fernandes, P., Allamanis, M., and Brockschmidt, M. Structured neural summarization. *arXiv preprint arXiv:1811.01824*, 2018.

Fort, S., Hu, H., and Lakshminarayanan, B. Deep ensembles: A loss landscape perspective. *arXiv preprint arXiv:1912.02757*, 2019.

Gao, F., Wang, Y., and Wang, K. Discrete adversarial attack to models of code. *Proceedings of the ACM on Programming Languages*, 7(PLDI):172–195, 2023a.

Gao, S., Gao, C., Wang, C., Sun, J., Lo, D., and Yu, Y. Two sides of the same coin: Exploiting the impact of identifiers in neural code comprehension. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pp. 1933–1945. IEEE, 2023b.

Garcez, A. d., Bader, S., Bowman, H., Lamb, L. C., de Penning, L., Illuminoo, B., Poon, H., and Zaverucha, C. G. Neural-symbolic learning and reasoning: A survey and interpretation. *Neuro-Symbolic Artificial Intelligence: The State of the Art*, 342(1):327, 2022.

Gordon, J., Lopez-Paz, D., Baroni, M., and Bouchacourt, D. Permutation equivariant models for compositional generalization in language. In *International Conference on Learning Representations*, 2019.

Guo, D., Ren, S., Lu, S., Feng, Z., Tang, D., Liu, S., Zhou, L., Duan, N., Svyatkovskiy, A., Fu, S., et al. Graphcodebert: Pre-training code representations with data flow. *arXiv preprint arXiv:2009.08366*, 2020.

Guo, D., Lu, S., Duan, N., Wang, Y., Zhou, M., and Yin, J. Unixcoder: Unified cross-modal pre-training for code representation. *arXiv preprint arXiv:2203.03850*, 2022.

Guo, W., Mu, D., Xing, X., Du, M., and Song, D. DEEPVSA: Facilitating value-set analysis with deep learning for postmortem program analysis. In *28th USENIX Security Symposium (USENIX Security 19)*, 2019.

Hellendoorn, V. J., Sutton, C., Singh, R., Maniatis, P., and Bieber, D. Global relational models of source code. In *International conference on learning representations*, 2019.

Henke, J., Ramakrishnan, G., Wang, Z., Albarghouth, A., Jha, S., and Reps, T. Semantic robustness of models of source code. In *2022 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pp. 526–537. IEEE, 2022.

Higgins, I., Amos, D., Pfau, D., Racaniere, S., Matthey, L., Rezende, D., and Lerchner, A. Towards a definition of disentangled representations. *arXiv preprint arXiv:1812.02230*, 2018.

Hille, E. and Phillips, R. S. *Functional analysis and semigroups*, volume 31. American Mathematical Soc., 1996.

Hu, Z., Ma, X., Liu, Z., Hovy, E., and Xing, E. Harnessing deep neural networks with logic rules. *arXiv preprint arXiv:1603.06318*, 2016.

HuggingFace and ServiceNow. BigCode is an open scientific collaboration working on the responsible development and use of large language models for code. https://www.bigcode-project.org/, 2022.

Huh, D. Discovering symmetry group structures via implicit orthogonality bias. *arXiv preprint arXiv:2402.17002*, 2024.

Hutchinson, M. J., Le Lan, C., Zaidi, S., Dupont, E., Teh, Y. W., and Kim, H. Lietransformer: Equivariant self-attention for lie groups. In *International Conference on Machine Learning*, pp. 4533–4543. PMLR, 2021.

Ji, S., Xie, Y., and Gao, H. A mathematical view of attention models in deep learning. *Texas A&M University: College Station, TX, USA*, 2019.

Jin, X., Pei, K., Won, J. Y., and Lin, Z. Symlm: Predicting function names in stripped binaries via context-sensitive execution-aware code embeddings. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1631–1645, 2022.

Just, R., Jalali, D., and Ernst, M. D. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 international symposium on software testing and analysis*, pp. 437–440, 2014.

Kim, S., Zhao, J., Tian, Y., and Chandra, S. Code prediction by feeding trees to transformers. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*, pp. 150–162. IEEE, 2021.

Kim, Y., Denton, C., Hoang, L., and Rush, A. M. Structured attention networks. *arXiv preprint arXiv:1702.00887*, 2017.

Knuth, D. Permutations, matrices, and generalized young tableaux. *Pacific journal of mathematics*, 34(3):709–727, 1970.

Lachaux, M.-A., Roziere, B., Szafraniec, M., and Lample, G. Dobf: A deobfuscation pre-training objective for programming languages. *Advances in Neural Information Processing Systems*, 34:14967–14979, 2021.

Lee, J., Lee, Y., Kim, J., Kosiorek, A., Choi, S., and Teh, Y. W. Set transformer: A framework for attention-based permutation-invariant neural networks. In *International Conference on Machine Learning*, pp. 3744–3753. PMLR, 2019.

Li, X., Yu, Q., and Yin, H. Palmtree: Learning an assembly language model for instruction embedding. In *2021 ACM SIGSAC Conference on Computer and Communications Security*, 2021.

Li, Z., Huang, J., and Naik, M. Scallop: A language for neurosymbolic programming. *Proceedings of the ACM on Programming Languages*, 7(PLDI):1463–1487, 2023.

Liu, D., Metzman, J., and Chang, O. AI-Powered Fuzzing: Breaking the Bug Hunting Barrier. https://security.googleblog.com/2023/08/ai-powered-fuzzing-breaking-bug-hunting.html, 2023.

Luo, Z., Xu, C., Zhao, P., Sun, Q., Geng, X., Hu, W., Tao, C., Ma, J., Lin, Q., and Jiang, D. Wizardcoder: Empowering code large language models with evol-instruct. *arXiv preprint arXiv:2306.08568*, 2023.

Maniatis, P. and Tarlow, D. Large sequence models for software development activities. https://ai.googleblog.com/2023/05/large-sequence-models-for-software.html?m=1, 2023.

Marcelli, A., Graziano, M., Ugarte-Pedrero, X., Fratantonio, Y., Mansouri, M., and Balzarotti, D. How machine learning is solving the binary function similarity problem. In *31st USENIX Security Symposium (USENIX Security 22)*, pp. 2099–2116, 2022.

Nye, M., Andreassen, A. J., Gur-Ari, G., Michalewski, H., Austin, J., Bieber, D., Dohan, D., Lewkowycz, A., Bosma, M., Luan, D., et al. Show your work: Scratchpads for intermediate computation with language models. *arXiv preprint arXiv:2112.00114*, 2021.

Ott, M., Edunov, S., Baevski, A., Fan, A., Gross, S., Ng, N., Grangier, D., and Auli, M. Fairseq: A fast, extensible toolkit for sequence modeling. In *2019 Annual Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies: Demonstrations*, 2019.

Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., et al. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems*, 32, 2019.

Pei, K., Guan, J., Broughton, M., Chen, Z., Yao, S., Williams-King, D., Ummadisetty, V., Yang, J., Ray, B., and Jana, S. Stateformer: fine-grained type recovery from binaries using generative state modeling. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 690–702, 2021.

Pei, K., Xuan, Z., Yang, J., Jana, S., and Ray, B. Trex: Learning execution semantics from micro-traces for binary similarity. *IEEE Transactions on Software Engineering*, 2022.

Peng, H., Li, G., Wang, W., Zhao, Y., and Jin, Z. Integrating tree path in transformer for code representation. *Advances in Neural Information Processing Systems*, 34:9343–9354, 2021.

Perraudin, N., Defferrard, M., Kacprzak, T., and Sgier, R. Deepsphere: Efficient spherical convolutional neural network with healpix sampling for cosmological applications. *Astronomy and Computing*, 27:130–146, 2019.

Rabin, M. R. I., Bui, N. D., Wang, K., Yu, Y., Jiang, L., and Alipour, M. A. On the generalizability of neural program models with respect to semantic-preserving program transformations. *Information and Software Technology*, 135: 106552, 2021.

Raffel, C., Shazeer, N., Roberts, A., Lee, K., Narang, S., Matena, M., Zhou, Y., Li, W., and Liu, P. J. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research*, 21(140):1–67, 2020. URL http://jmlr.org/papers/v21/20-074.html.

Ramakrishnan, G., Henkel, J., Wang, Z., Albarghouthi, A., Jha, S., and Reps, T. Semantic robustness of models of source code. *arXiv preprint arXiv:2002.03043*, 2020.

Reiser, P., Neubert, M., Eberhard, A., Torresi, L., Zhou, C., Shao, C., Metni, H., van Hoesel, C., Schopmans, H., Sommer, T., et al. Graph neural networks for materials science and chemistry. *Communications Materials*, 3(1):93, 2022.

Romero, D. W. and Cordonnier, J.-B. Group equivariant stand-alone self-attention for vision. *arXiv preprint arXiv:2010.00977*, 2020.

Roziere, B., Gehring, J., Gloeckle, F., Sootla, S., Gat, I., Tan, X. E., Adi, Y., Liu, J., Remez, T., Rapin, J., et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.

Souza, B. and Pradel, M. Lexecutor: Learning-guided execution. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pp. 1522–1534, 2023.

Sun, Z., Zhu, Q., Xiong, Y., Sun, Y., Mou, L., and Zhang, L. Treegen: A tree-based transformer architecture for code generation. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 34, pp. 8984–8991, 2020.

Ullah, S., Han, M., Pujar, S., Pearce, H., Coskun, A., and Stringhini, G. Can large language models identify and reason about security vulnerabilities? not yet. *arXiv preprint arXiv:2312.12575*, 2023.

Wang, K. and Su, Z. Blended, precise semantic program embeddings. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation*, pp. 121–134, 2020.

Wang, R., Walters, R., and Yu, R. Incorporating symmetry into deep dynamics models for improved generalization. *arXiv preprint arXiv:2002.03061*, 2020.

Wang, S., Li, Z., Qian, H., Yang, C., Wang, Z., Shang, M., Kumar, V., Tan, S., Ray, B., Bhatia, P., et al. Recode: Robustness evaluation of code generation models. *arXiv preprint arXiv:2212.10264*, 2022.

Wang, Y., Wang, W., Joty, S., and Hoi, S. C. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. *arXiv preprint arXiv:2109.00859*, 2021.

Wen, Y., Yin, P., Shi, K., Michalewski, H., Chaudhuri, S., and Polozov, A. Grounding data science code generation with input-output specifications. *arXiv preprint arXiv:2402.08073*, 2024.

West, D. B. et al. *Introduction to graph theory*, volume 2. Prentice hall Upper Saddle River, 2001.

Xu, X., Liu, C., Feng, Q., Yin, H., Song, L., and Song, D. 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*, pp. 363–376, 2017.

Ye, H., Martinez, M., and Monperrus, M. Neural program repair with execution-based backpropagation. In *Proceedings of the 44th international conference on software engineering*, pp. 1506–1518, 2022.

Yefet, N., Alon, U., and Yahav, E. Adversarial examples for models of code. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–30, 2020.

Yi, K., Wu, J., Gan, C., Torralba, A., Kohli, P., and Tenenbaum, J. Neural-symbolic vqa: Disentangling reasoning from vision and language understanding. *Advances in neural information processing systems*, 31, 2018.

Ying, C., Cai, T., Luo, S., Zheng, S., Ke, G., He, D., Shen, Y., and Liu, T.-Y. Do transformers really perform badly for graph representation? *Advances in Neural Information Processing Systems*, 34:28877–28888, 2021.

Zhang, N. Hikari – an improvement over Obfuscator-LLVM. https://github.com/HikariObfuscator/Hikari, 2017.

Zhang, Z., Tao, G., Shen, G., An, S., Xu, Q., Liu, Y., Ye, Y., Wu, Y., and Zhang, X. Pelican: Exploiting backdoors of naturally trained deep learning models in binary code analysis. In *32nd USENIX Security Symposium*, 2023.

A. Detailed Preliminaries

This section formally defines the symmetry group and the invariance and equivariance properties against the symmetry group.

Symmetry group. Intuitively, a *symmetry* is a transformation or operation on an object that preserves certain properties of the object. For example, in the context of image classification, a rotation operation acting on an image of a ball, which does not change the label of the ball, can be considered a symmetry. A symmetry group is a set of such symmetries with some additional properties. An arbitrary set of symmetries does not always form a symmetry group. To form a symmetry group, a set of operations must possess certain additional properties as described below.

Definition A.1. A *symmetry group* $(G, \circ)$ consists of a non-empty set G of transformations and a binary operator $\circ : G \times G \to G$, where $\circ$ operates on two elements (i.e., transformations) in G, e.g., $x, y \in G$, and produces a new transformation $z = x \circ y$. $(G,\circ)$ should satisfy four axioms:

- **Associativity**: $\forall x,y,z \in G, x \circ (y \circ z) = (x \circ y) \circ z$
- **Identity**: $\exists \mathbf{1} \in G, \forall x \in G, x \circ \mathbf{1} = \mathbf{1} \circ x$
- **Inverse**: $\forall x \in G, \exists x^{-1} \in G, x \circ x^{-1} = \mathbf{1}$
- **Closure**: $\forall x,y \in G, x \circ y \in G$

Action of a symmetry group. As defined above, the elements of a G are abstract transformations that become concrete when they *act* on some set X, i.e., they transform some object $x \in X$ into another object $x' \in X$ while keeping some properties of

the object *invariant*. Formally, an action of a symmetry group G is defined as follows:

Definition A.2. An **action** $\bullet$ of a symmetry group $(G, \circ)$ is a binary operation defined on a set of objects X, i.e., $\bullet : G \times X \to X$[1], where

- **Identity:** $\forall x \in X, \mathbf{1} \bullet x = x$
- **Compatibility:** $\forall g, h \in G, x \in X, (g \circ h) \bullet x = g \bullet (h \bullet x)$

As a concrete example, X can be a set of programs and G can be all possible instruction permutations that preserve the input-output behavior of the programs in X. It might seem unclear at this point how these permutations form a group (satisfying group axioms). We will formalize the notion of permutations and their actions on programs in §3.3.

Notation. It is common in the group theory literature to use $\circ$ to denote both *action* and *composition*, when it is clear from the context which operation is being used (Higgins et al., 2018). For example, $(g \circ h) \circ x$ denotes *composing* the two transformations g and h and then letting the composite transformation *act* on an object x. It is also customary to interchange $g(x)$ and $g \circ x$ where both denote applying a function/action on x. Therefore, we treat $g \bullet (h \bullet x)$, $g \circ (h \circ x)$, and $g(h(x))$ as the same and follow this convention in the rest of this paper.

Invariance and equivariance. A symmetry group comes with two properties, namely *invariance* and *equivariance*, that formalize the concept of preservation of some properties when a set X is acted upon by the symmetry group G. Invariance refers to the property that remains unchanged under the action of the symmetry group. Equivariance, on the other hand, expresses the compatibility between the action of the symmetry group and the property.

To define this more precisely, we need to introduce a function $f : X \to Y$, where X is the set under consideration and Y is the co-domain representing the range of possible values associated with the property of interest. The function f maps each element $x \in X$ to a corresponding element y in the set Y, indicating the property's value for that particular element. We now define the equivariance and invariance of f operating on X against the group operations in G.

Definition A.3. Let $f : X \to Y$ be a function where X and Y are two sets and G be the symmetry group that acts on both sets X and Y.[2]

- **Invariant**: f is called **G-invariant** if $\forall g \in G, \forall x \in X, f(g \circ x) = f(x)$.
- **Equivariant**: f is called **G-equivariant** if $\forall g \in G, \forall x \in X, f(g \circ x) = g \circ f(x)$.

Given the definition of G-equivariant function, we have the following lemmas:

Lemma 4. Let f_1, f_2 be two functions that are both G-equivariant and $h = f_1 \circ f_2$ be the new function composed by f_1 and f_2. h is also G-equivariant.

Lemma 5. Let f_1, f_2 be two functions where f_1 is G-equivariant and f_2 is G-invariant, and $h = f_2 \circ f_1$ be the function composed by f_1 and f_2. h is G-invariant.

Self-attention layers. Given the embeddings of all vertices f_i from $\mathcal{IG}$, we consider a sequence of embeddings by flattening $\mathcal{IG}$ following the order of instructions in c. Let this sequence of embeddings be denoted as $e = (e_1, ..., e_n)$. The self-attention computation, denoted as A, takes e as input and produces another sequence of embeddings, denoted as $(e'_1, ..., e'_n)$.

The core operations in self-attention A involve updating each embedding e_i through the following steps:

1. First, it maps each embedding e_i to three embeddings (query, key, and value): $q_i = f_q(e_i)$, $k_i = f_k(e_i)$, $v_i = f_v(e_i)$, where f_q, f_k, and f_v are affine transformations (i.e., fully-connected linear layers) parameterized by w_q, w_k, and w_v, respectively.

2. Next, it computes the attention score a_{ij} between every pair of embeddings e_i and e_j by taking the dot product between the query q_i of e_i and the key k_j of e_j: $a_{ij} = q_i \cdot k_j$. The attention scores form a square matrix, where each cell a_{ij} indicates the attention that e_i should pay to e_j. The attention scores are then divided by $\sqrt{d}$ (the dimension of the embedding vectors), scaled using the softmax function to ensure they sum up to 1: $\hat{a}_{ij} = \frac{\exp(aij)}{\sum_{j=1}^{n} \exp(a_{ij})}$. These two operations are denoted by s.

3. Finally, the scaled attention score $\hat{a}_{ij}$ is multiplied by v_j, and a vector sum is computed: $e'_i = \sum_{j=1}^{n} \hat{a}_{ij} v_{ij}$.

B. Complete Proofs

Lemma 4. Let f_1 and f_2 be two functions that are both G-equivariant and $h = f_1 \circ f_2$ be the new function composed by f_1 and f_2. h is also G-equivariant.

[1] In group theory literature, this is often called the left action, but we will omit "left" as it is the only type of action we will use in this paper.

[2] We assume that X and Y have the same number of elements for the action of G to be defined on both X and Y.

Proof. For all $g \in G$ and any input x, we have

$$\begin{aligned} h(g \circ x) &= (f_1 \circ f_2)(g \circ x) \\ &= f_1(f_2(g \circ x)) && \triangleright \text{ Associativity} \\ &= f_1(g \circ f_2(x)) && \triangleright\ f_2 \text{ is equivariant to } g \\ &= g \circ f_1(f_2(x)) && \triangleright\ f_1 \text{ is equivariant to } g \\ &= g \circ (f_1 \circ f_2)(x) && \triangleright \text{ Associativity} \\ &= g \circ h(x) \end{aligned}$$

Therefore, $h(g \circ x) = g \circ h(x)$, so h is G-equivariant.

Lemma 5. Let f_1 and f_2 be two functions where f_1 is G-equivariant f_2 is G-invariant, and $h = f_2 \circ f_1$ be the new function composed by applying f_1 and then f_2. h is G-invariant.

Proof. For all $g \in G$ and any input x, we have

$$\begin{aligned} h(g \circ x) &= (f_2 \circ f_1)(g \circ x) \\ &= f_2(f_1(g \circ x)) && \triangleright \text{ Associativity} \\ &= f_2(g \circ f_1(x)) && \triangleright\ f_1 \text{ is equivariant to } g \\ &= f_2(f_1(x)) && \triangleright\ f_2 \text{ is invariant to } g \\ &= (f_2 \circ f_1)(x) && \triangleright \text{ Associativity} \\ &= h(x) \end{aligned}$$

Theorem 1. The set of automorphisms $\sigma \in Aut(\mathcal{IG})$ forms a program symmetry group.

Proof. Consider an arbitrary $\sigma \in Aut(\mathcal{IG})$. Definition 3.6 states that for all $f_i \in \{f_1,...,f_n\}$, $\sigma(f_i)$ have the same edges as $\mathcal{IG}$ before σ was applied. As σ is a permutation and there is also a bijective mapping between f_i and c_i, i.e., f_i always interprets c_i, we have $\sigma(f_i) = f_i(\sigma \circ c_i, in_i)$. Definition 3.6 also states that $\sigma(f_i)$ is connected with the same edges. Therefore, the output of $\sigma(f_i) = out_i$. We thus have $f_i(\sigma \circ c_i, in_i) = out_i = f_i(c_i, in_i), \forall \sigma \in Aut(\mathcal{IG})$ and $\forall f_i \in \{f_1,...,f_n\}$. Therefore, all $\sigma \in Aut(\mathcal{IG})$ are semantics-preserving program symmetries, according to Definition 3.4. Moreover, it is well known in the literature that the automorphisms of any graph form a group by satisfying group axioms (Definition A.1) (Biggs et al., 1993; West et al., 2001). Therefore, $Aut(\mathcal{IG})$ forms a group of program symmetries, according to Definition 3.5: $Aut(\mathcal{IG}) \in G$.

Permutation matrix. Let π be a symmetry in the permutation group that permutes input embeddings $e \in \mathbb{R}^{d \times n}$ to the self-attention layer. Applying π is done by e with a permutation matrix $p_\pi \in \{0,1\}^{n \times n}$ (Knuth, 1970). p_π is an orthogonal binary matrix with a single 1 in each column and row, and 0s elsewhere. Right-multiplying e with p_π permutes columns, and left-multiplying e^T with p_π^T permutes rows.

Lemma 1. The biased self-attention layer computing the embedding $e_i' = GA(e_i)$ is $Aut(\mathcal{IG})$-invariant.

Proof.

$$\begin{aligned} e_i' &= GA(\sigma \cdot e_i) \\ &= w_v \sigma(e) \cdot s(w_k \sigma(e)^T \cdot w_q e_i + \sigma(d_i)) \end{aligned}$$

d_i is a column vector, so permuting the row of d_i is achieved by $p_\sigma^T d_i$ (see §3.4):

$$\begin{aligned} &= w_v e p_\sigma \cdot s((w_k e p_\sigma)^T \cdot w_q e_i + p_\sigma^T d_i) \\ &= w_v e p_\sigma \cdot s(p_\sigma^T (w_k e)^T \cdot w_q e_i + p_\sigma^T d_i) \\ &= w_v e (p_\sigma p_\sigma^T) \cdot s((w_k e)^T \cdot w_q e_i + d_i) \end{aligned}$$

p_σ is an orthogonal matrix (see §3.4):

$$\begin{aligned} &= w_v e \cdot s((w_k e)^T \cdot w_q e_i + d_i) \\ &= GA(e_i) \end{aligned}$$

Lemma 2. The distance matrix d of PDG remains invariant under the action of $\sigma \in Aut(PDG)$.

Proof. We need to show that the longest path $p_{\sigma(i)\sigma(j)}$ from $\sigma(T_{ij})$ to $\sigma(V_i)$ remains the same as p_{ij} (the same applies to $n_{\sigma(i)\sigma(j)}$). Without loss of generality, we focus on proving $p_{\sigma(i)\sigma(j)} = p_{ij}$.

Assume there exists a longest path $P = (T_{ij}, ..., V_i)$. Let $P' = (\sigma(T_{ij}),...,\sigma(V_i))$ be the corresponding longest path in $\sigma(PDG)$ under the automorphism σ. We need to demonstrate two properties.

First, P' is a valid path from $\sigma(T_{ij})$ to $\sigma(V_i)$. Since P is a valid path, T_{ij} is adjacent to its next node in P (denoted as V_m), and this holds for every pair of neighboring nodes until V_i. As σ is an automorphism, the same adjacency relationship holds for P', where $\sigma(T_{ij})$ is adjacent to $\sigma(V_m)$ and so on, until $\sigma(V_i)$. Hence, P' is a valid path from $\sigma(T_{ij})$ to $\sigma(V_i)$ in PDG.

Second, we aim to show that $|P| = |P'|$, meaning $p_{\sigma(i)\sigma(j)} = p_{ij}$. Suppose, for contradiction, that $p_{\sigma(i)\sigma(j)} \neq p_{ij}$. Let's consider the case where $p_{\sigma(i)\sigma(j)} > p_{ij}$. This implies that the length of the path $P' = (\sigma(T_{ij}),\sigma(V_m),...,\sigma(V_n),\sigma(V_i))$ is longer than p_{ij}.

Now, let's apply σ^{-1} to each node in P', resulting in $\sigma^{-1}(P')$. Since σ^{-1} is also in $Aut(PDG)$ and $\sigma^{-1}(\sigma(V)) = V$ (Definition A.1), each pair of adjacent nodes in P', after applying σ^{-1}, remains adjacent. Furthermore, the path formed by these adjacent nodes has a length of $p_{\sigma(i)\sigma(j)}$, connecting T_{ij} and V_i in the original PDG.

Therefore, we obtain a path in PDG connecting T_{ij} and V_i that is longer than p_{ij}, contradicting the fact that p_{ij} is the longest path in PDG between T_{ij} and V_i. Thus, we reject the assumption that $p_{\sigma(i)\sigma(j)} < p_{ij}$.

Similarly, we can prove that $p_{\sigma(i)\sigma(j)} > p_{ij}$ is also false by demonstrating its contradiction with the fact that $p_{\sigma(i)\sigma(j)}$ is the longest path in $\sigma(PDG)$.

Hence, we conclude that $p_{\sigma(i)\sigma(j)} = p_{ij}$, and as a result, the positive distance matrix dp remains invariant under the action of $\sigma \in Aut(PDG)$.

By following the same steps, we can prove that $n_{\sigma(i)\sigma(j)} = n_{ij}$, demonstrating the invariance of the negative distance matrix dn under the action of $\sigma \in Aut(PDG)$.

Therefore, the distance matrix d remains invariant.

Lemma 3. The distance matrix d of PDG commutes with permutation matrix p_σ of the automorphism $\sigma \in Aut(PDG)$: $d \cdot p_\sigma = p_\sigma \cdot d$.

Proof. According to Lemma 2, we have:

$$
\begin{aligned}
p_\sigma^T \cdot d \cdot p_\sigma &= d \\
p_\sigma \cdot p_\sigma^T \cdot d \cdot p_\sigma &= p_\sigma \cdot d && \triangleright \text{ Apply } p_\sigma \text{ on both side} \\
d \cdot p_\sigma &= p_\sigma \cdot d && \triangleright \ p_\sigma \text{ is orthogonal}
\end{aligned}
$$

Lemma 6. Standard self-attention layer A is equivariant to the group of all permutations of input sequences.

Proof. Based on the operations performed by the self-attention layer and the permutation matrix, we can show the equivariance property as follows (Ji et al., 2019):

$$
\begin{aligned}
&A(\pi \cdot e) \\
&= w_v \pi(e) \cdot s(w_k \pi(e)^T \cdot w_q \pi(e)) \\
&= w_v e p_\pi \cdot s((w_k e p_\pi)^T \cdot w_q e p_\pi) && \triangleright \text{ Apply } p_\pi \\
&= w_v e p_\pi \cdot s(p_\pi^T (w_k e)^T \cdot w_q e p_\pi) \\
&= w_v e (p_\pi p_\pi^T) \cdot s((w_k e)^T \cdot w_q e) p_\pi \\
&= w_v e \cdot s((w_k e)^T \cdot w_q e) p_\pi && \triangleright \ p_\pi \text{ is orthogonal} \\
&= \pi(A(e))
\end{aligned}
$$

Based on Lemma 2 and Lemma 3, we now prove Theorem 2 – the biased self-attention layer is $Aut(\mathcal{IG})$-equivariant.

Proof.

$$
\begin{aligned}
&GA(\sigma \cdot e) \\
&= w_v \sigma(e) \cdot s(w_k \sigma(e)^T \cdot w_q \sigma(e) + \sigma(d_{\mathcal{IG}}))
\end{aligned}
$$

$\sigma(\cdot)$ denotes applying the permutation matrix p_σ. As we have $\sigma(d_{\mathcal{IG}}) = d_{\mathcal{IG}}$ (the first property of $d_{\mathcal{IG}}$):

$$
= w_v e p_\sigma \cdot s((w_k e p_\sigma)^T \cdot w_q e p_\sigma + d_{\mathcal{IG}})
$$

Softmax s is permutation equivariant, and $d_{\mathcal{IG}} \cdot p_\sigma = p_\sigma \cdot d_{\mathcal{IG}}$ (the second property of $d_{\mathcal{IG}}$):

$$
\begin{aligned}
&= w_v e (p_\sigma p_\sigma^T) \cdot s((w_k e)^T \cdot w_q e \cdot p_\sigma + d_{\mathcal{IG}} \cdot p_\sigma) \\
&= w_v e \cdot s((w_k e)^T \cdot w_q e + d_{\mathcal{IG}}) \cdot p_\sigma \\
&= \sigma(GA(e))
\end{aligned}
$$

C. SYMC Implementation Details

Input sequences to self-attention. The Transformer self-attention layer takes an input sequence of embeddings e generated by the embedding layer Emb. It consists of four input sequences: the instruction sequence c, per-instruction positional embeddings, and node centrality, denoted as x_c, x_{pos}, x_{ind}, and x_{outd}, respectively. For example, given the instruction sequence `a=a+1;b=a`, x_c represents the tokenized sequence as (`a`,`=`,`a`,`+`,`1`,`b`,`=`,`a`). x_{pos} assigns positions such that each new instruction/statement begins with position 1 of its first token and increases by 1 for each subsequent token within the instruction.

The centrality of each instruction is encoded by the in-degree and out-degree of the corresponding node in PDG. For each token in c_i, we annotate it with its in-degree (number of incoming edges) and out-degree (number of outgoing edges). For instance, in the case of `a=a+1;b=a`, the in-degree sequence x_{ind} is $(0, 0, 0, 0, 0, 1, 1, 1)$, and the out-degree sequence x_{outd} is (1,1,1,1,1,0,0,0).

We embed the four sequences independently using the embedding layers Emb_c, Emb_{pos}, Emb_{ind}, and Emb_{outd}. The final input embedding sequences $Emb(x)$ are obtained by summing the embedded sequences for each token: $Emb(x) = Emb_c(x_c) + Emb_{pos}(x_{pos}) + Emb_{ind}(x_{ind}) + Emb_{outd}(x_{outd})$. We have the following lemma:

Lemma 7. The sum of the input embedding tokens sequences is $Aut(PDG)$-equivariant: $Emb(\sigma \circ x) = \sigma \circ Emb(x)$.

Group axiom of inclusion specifies that composing the $Aut(PDG)$-equivariant embedding layers with $Aut(PDG)$-equivariant MHA layers results in an $Aut(PDG)$-equivariant representation learning component r in our implementation.

D. Detailed Experiment Setup

D.1. Implementation Details

We implement SYMC using Fairseq (Ott et al., 2019) PyTorch (Paszke et al., 2019). We conduct all the experiments on three Linux servers with Ubuntu 20.04 LTS, each featuring

an AMD EPYC 7502 processor, 128 virtual cores, and 256GB RAM, with 12 Nvidia RTX 3090 GPUs in total.

PDG construction. To compute PDG for x86 assembly code, we utilize Ghidra to lift the assembly code into P-Code, an intermediate representation used by Ghidra, to track implicit data and control flow via the FLAGS register. The key advantage of using Ghidra P-Code is that it keeps the side effects of the instructions, e.g., manipulating flag registers, explicit. For example, `cmp` instruction will set the zero flag implicitly at the assembly level, but Ghidra P-Code will translate it into a series of IR instructions with one of them operating on the ZF explicitly. We then analyze the data and control dependencies between each pair of P-Code instructions and flag the corresponding assembly code pair as dependent if at least one dependent P-Code instruction pair exists between those of the assembly code pair.

To compute PDG for Java functions, we employ JavaParser on Java ASTs for each statement to analyze control and data dependencies. We iterate through every pair of the statements and connect the dependent statement pairs with the directed edge. We extract control-flow dependencies between statements by connecting edges between different basic blocks to prevent permutations among basic blocks.

Datasets. We use the Java dataset collected by Allamanis et al. (2016) to evaluate the function name prediction. The dataset includes 11 Java projects, such as Hadoop, Gradle, etc., totaling 707K methods and 5.6M statements. We fix Hadoop as our test set and use the other projects for training, to ensure the two sets do not overlap.

For binary analysis, we collect and compile 27 open-source projects, such as OpenSSL, ImageMagic, CoreUtils, SQLite, etc., which contain approximately 1.13M functions and 137.6M instructions. We categorize the binaries based on the compilers (`GCC` or `Clang`), optimizations (`O0`-`O3`), and obfuscations (using a LLVM-based obfuscation passes based on Hikari (Zhang, 2017)) and show their statistics in Table 4.

Table 4: The statistics of our binary dataset, categorized by compilers, optimizations, obfuscations, and lengths.

	# Files	# Functions	# Instructions
Different Compilers			
`GCC`	1,140	274,840	33,464,420
`Clang`	1,136	279,832	31,949,673
Different Optimization Levels			
`O0`	285	98,451	10,202,328
`O1`	285	61,298	7,096,903
`O2`	285	61,298	7,096,903
`O3`	285	57,023	9,101,578
Different Compiler Obfuscations			
`bcf`	158	61,701	9,173,168
`cff`	158	59,724	11,146,990
`ind`	158	56,291	2,501,422
`spl`	158	61,379	9,652,268
`sub`	158	59,694	6,198,900

D.2. Experiment Configurations

Program analysis tasks for evaluation. For source code analysis tasks, we focus on the *method name prediction* and *defect prediction*. Method name prediction aims to predict the function name (in natural language) given the body of the method. This task has been extensively evaluated by prior works to test the generalizability of code models (Rabin et al., 2021). Following the strategies adopted in SymLM (Jin et al., 2022), we tokenize the function names and formulate the function name prediction as a multi-label classification problem, i.e., multiple binary classifications that predict the presence of a specific token in the vocabulary. We then match the predicted tokens with the tokenized ground truth tokens to compute the F1 score. We thus employ a 2-layer fully-connected network $F:\mathbb{R}^d \rightarrow \{0,1\}^L$ on top of a mean-pooled embedding from self-attention layers to ensure $Aut(PDG)$-invariance (§3.5), where L is the vocabulary of all function name tokens in our dataset.

The defect prediction task is much more simplified than method name prediction. It is a binary classification task to predict whether a given Java method is buggy or not. We obtain the dataset from Defects4J (Just et al., 2014).

We consider three binary analysis tasks commonly used to evaluate ML-based approaches to security applications (Li et al., 2021). The first task is *function similarity detection*. It aims to detect semantically similar functions, e.g., those compiled by different compiler transformations (see below). This task is often used to detect vulnerabilities, i.e., by searching similar vulnerable functions in firmware, or malware analysis, i.e., by searching similar malicious functions to identify the malware family (Marcelli et al., 2022; Xu et al., 2017). We leverage the pooling-based predictor (§3.5) by taking the mean of the embeddings e produced by the last self-attention layer and feed that to a 2-layer fully-connected neural network $F:\mathbb{R}^d \rightarrow \mathbb{R}^d$. We then leverage the cosine distance between the output of F for a pair of function embeddings, i.e., e^1,e^2, to compute their similarity: $cos(F(\mu(e^1)),F(\mu(e^2)))$.

The second task is *function signature prediction* (Chua et al., 2017). It aims to predict the number of arguments and their source-level types given the function in stripped binaries. Similar to function similarity detection, we stack a 2-layer fully-connected network $F:\mathbb{R}^d \rightarrow L$ on top of mean-pooled embeddings from self-attention layers, which outputs the function signature label, e.g., $L=\{$`int`,`float`,...$\}$.

The third task is *memory region prediction* (Guo et al., 2019), which aims to predict the type of memory region, i.e., stack, heap, global, etc., that each memory-accessing instruction can

access in a stripped binary. As the prediction happens for each instruction, we employ the token-level predictor (§3.5) F: $\mathbb{R}^d \to L$, where $L = \{\texttt{stack}, \texttt{heap}, \texttt{global}, \texttt{other}\}$.

Baselines. We consider nine baselines for function name prediction, including the dedicated models trained to predict function names (Alon et al., 2019; 2018; Fernandes et al., 2018) and code LLMs (Luo et al., 2023; Guo et al., 2020; Lachaux et al., 2021; Wang et al., 2021; Roziere et al., 2023). For defect prediction, we excluded the models specialized for function name prediction, while including two additional code models that have been evaluated in the defect prediction task (Guo et al., 2022; Feng et al., 2020). Note that the dataset used to train the baseline LLMs might overlap with our test set. For example, Hadoop (our test set for function name prediction, see Appendix D) is included in BigCode (HuggingFace & ServiceNow, 2022), one of the widely used datasets to train code LLMs. Our goal is to demonstrate SYMC still generalizes better than existing code LLMs under such a disadvantaged setting.

For tasks (3)-(5), we compare to PalmTree (Li et al., 2021), the only binary code model that has evaluated on all our considered tasks. To ensure a fair comparison, we include three PalmTree versions: PalmTree, PalmTree-O, and PalmTree-N. PalmTree is pre-trained on *2.25 billion* instructions. PalmTree-O is pre-trained on *137.6 million* instructions using our own dataset (Appendix D), with full access to fine-tuning and evaluation data (excluding labels), while not accessible by SYMC as it is not pre-trained. We aim to show SYMC's strong generalizability even in this disadvantaged setting. PalmTree-N serves as the baseline Transformer encoder without being pre-trained.

Transformations. We consider a set of semantics-preserving transformations beyond PDG automorphisms to evaluate how preserving $Aut(PDG)$-equivariant improves SYMC's generalizability. Notably, some of these program transformations (described below) have enabled instruction reordering, which inherently performs instruction permutations.

We consider two categories of binary code transformations: (1) *compiler optimizations*, where we examine 4 optimization levels (`O0`–`O3`) from GCC-7.5 and Clang-8, some of which involve instruction permutations, like reordering for scheduling purposes (`-fdelayed-branch`, `-fschedule-insns`); and (2) *compiler-based obfuscations*, where we follow SymLM (Jin et al., 2022) by using 5 obfuscations written in LLVM, i.e., control flow flattening (`cff`), instruction substitution (`sub`), indirect branching (`ind`), basic block split (`spl`), and bogus control flow (`bcf`), which inherently include reordering instructions, e.g., adding a trampoline.

Hyperparameters. We use SYMC with 8 attention layers, 12 attention heads, and a maximum input length of 512. For training, we use 10 epochs, a batch size of 64, and 14K/6K training/testing samples (strictly non-overlapping) unless stated otherwise. We employ 16-bit weight parameters for SYMC to optimize for memory efficiency.

Evaluation metrics. For most analysis tasks (§5), we use *F1 score*, the harmonic mean of precision and recall. We follow the existing works (Jin et al., 2022) and adopt their definition of F1 beyond the binary classifier. Take function name prediction as an example, we first tokenize both the ground truth and the predicted function names into a set of tokens, i.e., W and W', respectively. In this case, precision measures out of W', how many tokens in W' appear in W: $precision = \frac{W \cap W'}{|W'|}$, and recall measures out of all the tokens in W, how many of them are correctly predicted in W': $recall = \frac{W \cap W'}{|W|}$. We measure the precision and recall and compute the F1 score for each sample accordingly. We then average them across all samples.

For function similarity detection, as the cosine distance between two function embeddings can be an arbitrary real value between -1 and 1, a threshold is needed to determine whether pairs are similar or not. Therefore, we employ the ROC curve by varying the thresholds and measuring the corresponding True Positive Rate (TPR): $\text{TPR} = \frac{\text{True Positives}}{\text{True Positives+False Negatives}}$ and False Positive Rate (FPR): $\text{FPR} = \frac{\text{False Positives}}{\text{False Positives+True Negatives}}$. The ROC curve is then plotted with FPR at the x-axis and TPR at the y-axis. Following Li et al. (2021), we leverage the Area Under Curve (AUC) score of the ROC curve to quantify the performance for ease of comparison.

We note that AUC-ROC might not be the most reliable metric (Arp et al., 2022), but we choose it primarily for comparing to the baselines whose results are measured in AUC-ROC (Li et al., 2021).

E. Additional Experiments

E.1. Generalization and Robustness

Table 5 shows the complete results of SYMC and baselines against semantics-preserving code transformations across different analysis tasks. w measures the number of steps of applying (non-repeated) 2-statement permutations by permuting different statements. The larger the w, the more permutations are applied. While we do not observe a clear trend that a higher violation rate or decreased performance correlates with the number of times applying permutation, SYMC consistently shows consistent performance and 0 violation rate. On the contrary, all the baselines are not robust against the permutations and have their labels changed, i.e., by up to 86% violation rate.

Table 6 shows the complete results when evaluating SYMC and other baselines on new samples transformed by the semantics-preserving transformations that have never been presented in the training. We integrate CodeWordNet (Jin et al., 2022) to relax predicted names to a cluster of synonyms, addressing the issue of ambiguity of function names. However, the performance of SYMC decreases to 0.309 (was 0.374) when we measure the exact match. As discussed in §6.1, we observe that SYMC

Table 5: Complete evaluation statistics on samples under different percentages of semantics-preserving permutations. F1 measures the prediction performance of function name, function signature, and memory region. AUC (area under the ROC curve) measures the function similarity detection performance. The violation rate is highlighted in red. The larger the violation rate, the darker the color.

		Model Size	Train Size	F1 & AUC					Invariance Violation (%)			
				Before	$w=1$	$w=2$	$w=3$	$w=4$	$w=1$	$w=2$	$w=3$	$w=4$
Function Name	SYMC	68.4M	202M	0.363	0.364*	0.363	0.363	0.363	0	0.1*	0	0
	code2seq	6.3M	5.1G	0.255	0.238	0.236	0.237	0.247	54	53	57	61
	code2vec	348M	32G	0.177	0.199	0.195	0.197	0.196	53	53	52	52
	CodeLlama	7B	N/A	0.317	0.317	0.314	0.303	0.314	19	18	19	18
	CodeT5	770M	N/A	0.254	0.254	0.254	0.254	0.254	9	13	15	16
	DOBF	428M	N/A	0.163	0.182	0.182	0.175	0.201	22	28	36	41
	GGNN	53M	2.4G	0.016	0.016	0.016	0.016	0.016	4	4	5	7
	GPT-4	N/A	N/A	0.303	0.313	0.317	0.329	0.307	42	43	45	43
	GraphCodeBERT	481M	N/A	0.208	0.205	0.212	0.202	0.206	13	22	28	31
	WizardCoder	3B	N/A	0.339	0.347	0.348	0.359	0.346	6	7	12	14
Defect Prediction	SYMC	67.7M	720K	0.688	-	-	-	0.688	-	-	-	0
	CodeBERT	476M	N/A	0.622	-	-	-	0.617	-	-	-	4.1
	CodeT5	770M	N/A	0.633	-	-	-	0.6	-	-	-	6
	DOBF	428M	N/A	0.624	-	-	-	0.615	-	-	-	2.7
	GraphCodeBERT	481M	N/A	0.617	-	-	-	0.617	-	-	-	1.3
	UnixCoder	504M	N/A	0.671	-	-	-	0.671	-	-	-	2.9
Function Signature	SYMC	58.3M	12M	0.88	0.88	0.88	0.88	0.88	0	0	0	0
	PalmTree	3.2M	17.4G	0.59	0.55	0.49	0.42	0.41	12	23	18	24
	PalmTree-O	3.2M	5.3G	0.49	0.48	0.45	0.41	0.41	19	6	12	6
	PalmTree-N	3.2M	614M	0.19	0.41	0.41	0.41	0.41	83	82	83	86
Memory Region	SYMC	58.9M	340M	0.86	0.86	0.86	0.86	0.86	0	0	0	0
	PalmTree	3.07M	17.9G	0.57	0.45	0.45	0.48	0.43	17	17	28	18
	PalmTree-O	3.07M	5.8G	0.57	0.42	0.45	0.47	0.44	10	13	14	11
	PalmTree-N	3.07M	1.1G	0.32	0.22	0.29	0.17	0.2	30	36	31	32
Function Similarity	SYMC	58.9M	133M	0.96	0.96	0.96	0.96	0.96	0	0	0	0
	PalmTree	3.06M	17.4G	0.72	0.61	0.53	0.71	0.69	18	19	30	31
	PalmTree-O	3.06M	5.3G	0.8	0.79	0.76	0.72	0.72	30	28	30	35
	PalmTree-N	3.06M	614M	0.71	0.64	0.56	0.66	0.72	11	18	24	38

*We observe a slight value change due to the floating point precision error by adopting memory-efficient 16-bit.

outperforms the strong baselines, e.g., code2seq, by 30.8%.

Unseen optimizations. We vary the compiler optimizations in training and evaluation and include reference experiments where the training and evaluation share the same optimization options (marked in gray). For function similarity detection, training on `O0-O1` means the function pair has one function compiled with `O0` and the other with `O1`. In the case of evaluating on unseen optimizations, the corresponding testing set has to come from those compiled with `O2-O3` to ensure the optimizations are unseen.

Figure 8 shows that SYMC outperforms PalmTree by 31% when evaluated on unseen optimizations. SYMC experiences a performance drop (e.g., by 28.6%) when not trained on `O0` but tested on those compiled with `O0`. We believe this drop is caused by the extensive optimizations already enabled at the `O1` (e.g., GCC employs 47 optimizations to aggressively reduce execution time and code size). The shift in distribution between `O1` and `O0` is much more pronounced than between `O2` and `O1`, indicated by a KL divergence of 1.56 from `O1` to `O0` compared to 0.06 (96.2% lower) from `O3` to `O2`. Nevertheless, when evaluated on seen optimizations,

Table 6: The performance (F1) of SYMC and baselines against different unseen code transformations.

Transform	Applied	SYMC	code2seq	code2vec	CodeLlama	GGNN	GPT-4	WizardCoder
Variable Rename	Before	**0.389**	0.334	0.264	0.461	0.029	0.356	0.362
	After	**0.375**	0.335	0.247	0.426	0.026	0.351	0.361
Statement Permute	Before	**0.363**	0.241	0.177	0.317	0.019	0.303	0.339
	After	**0.363**	0.234	0.196	0.314	0.019	0.307	0.346
Loop Exchange	Before	**0.373**	0.283	0.243	0.414	0.007	0.310	0.379
	After	**0.357**	0.299	0.241	0.399	0.007	0.308	0.366
Boolean Exchange	Before	**0.421**	0.332	0.268	0.360	0.031	0.329	0.414
	After	**0.412**	0.272	0.242	0.447	0.026	0.323	0.406
Unused Statement	Before	**0.347**	0.296	0.267	0.429	0.016	0.316	0.358
	After	**0.342**	0.285	0.26	0.428	0.012	0.309	0.350
Switch to If	Before	**0.372**	0.31	0.376	0.430	0.027	0.326	0.385
	After	**0.372**	0.293	0.33	0.429	0.009	0.332	0.379

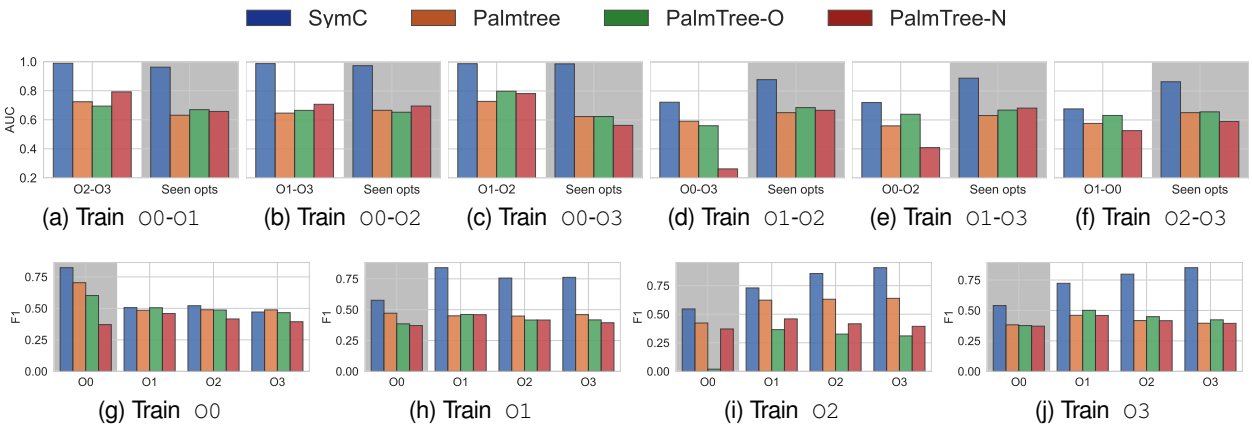


Figure 8: Unseen optimization evaluation. The upper row, i.e., (a)-(f), shows the results on function similarity detection. The lower row, i.e., (g)-(j), are results on function signature prediction. We also include the evaluation on seen optimizations (marked in gray).

SYMC outperforms PalmTree by 28.1% on average.

Unseen obfuscations. We compare SYMC to baselines on generalization to unseen obfuscations. Figure 9 shows that SYMC outperforms PalmTree (on average) on unseen and seen obfuscations by 33.3% and 36.6%, respectively. Similar to the observations in evaluating unseen optimizations, while the obfuscations are not directly related to instruction permutations (i.e., automorphisms in $Aut(PDG)$), SYMC maintains its superior performance.

Unseen lengths. Besides the code transformations, we look into SYMC's generalization to *longer* sequences than those seen in training, a popular task for evaluating model generalizability (Gordon et al., 2019). We divide samples into four length bins (`bin1` to `bin4`) based on their distribution in the dataset (§5). The bins are non-overlapping and increase in length. For example, we used bins [0-10], [1-20], [21-50], and [51-500] for function similarity detection. Figure 10 demonstrates that SYMC maintains strong generalization to longer sequences, outperforming PalmTree by 41.8%.

Table 7: The performance (F1) of SYMC and baselines against the adversarial transformations transformations.

	Orig.	Adv.	Invariance Violation (%)
SYMC	**52.9**	**47.5**	**26**
GraphCodeBERT	52.56	42.89	51
DOBF	51.59	39.68	51
CodeT5	44.21	36.66	47

Adversarial robustness. In addition to randomly transforming samples, we consider adversarial attacks where the generation

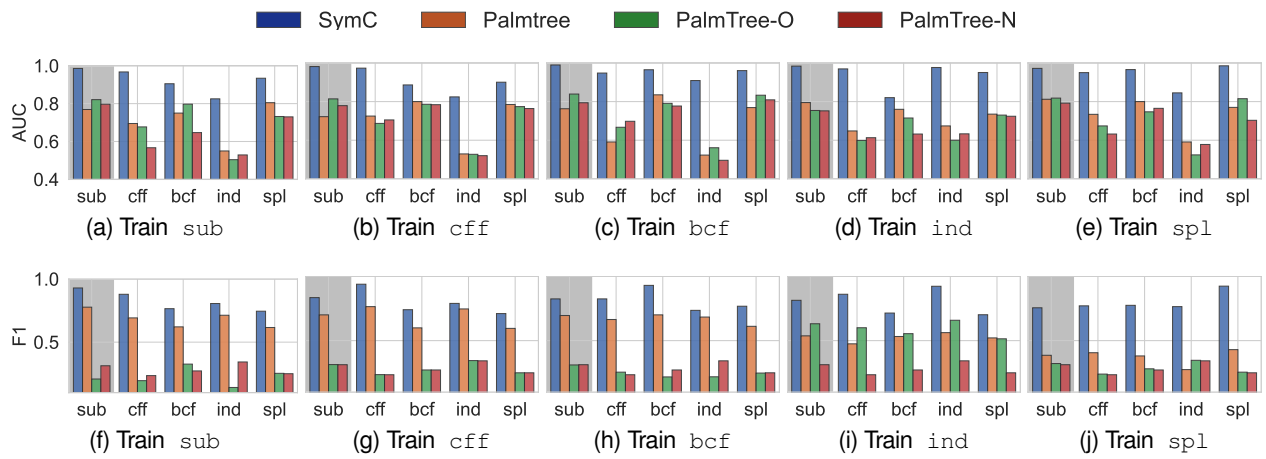


Figure 9: Unseen obfuscations evaluation. Similar to Figure 8, the upper row, i.e., (a)-(e), shows the results on function similarity detection. The lower row, i.e., (f)-(j), are results on function signature prediction. We also include the evaluation on seen optimizations (marked in gray).

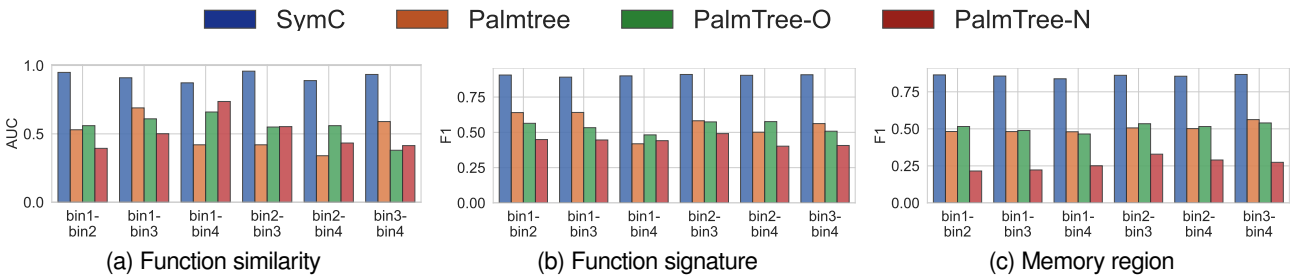


Figure 10: Evaluation on unseen samples with longer lengths. `bin1-bin4` denotes training on samples with lengths in `bin1` and testing on those in `bin4`.

of semantics-preserving transformations is further guided by an objective that maximizes the changed predictions of the model. In particular, we compare SYMC and the baselines against the adversarial attack, i.e., Averloc (Ramakrishnan et al., 2020), for function name prediction. The adversarial transformations implemented in Averloc include a subset of the code transformations we considered, e.g., variable renaming, dead code insertion, etc., with additional transformations such as loop unrolling. We follow the setting in Averloc by computing the adversarial attacks against a seq2seq model trained by the Averloc authors, and evaluate SYMC and the baselines on the generated adversarial examples. This ensures a fair comparison by evaluating all the models on the same set of adversarial examples.

Table 7 shows that SYMC outperforms the second-best baseline, GraphCodeBERT, by 10.7% and 49%, in F1 and violation rate on the adversarial examples, respectively. This indicates the strong robustness of SYMC against adversarial code transformations, even though the attacks are not statement permutations.

E.2. Efficiency

Overhead. To incorporate the inductive bias of code, many code models involve extracting and encoding code structures, including our baselines, e.g., GraphCodeBERT and GGNN. This is because computing PDGs statically is not overly expensive. Figure 11 shows the runtime performance (in milliseconds) per code sample of SYMC and PalmTree on extracting code structures, training, and inference, using the same exact hardware.

SYMC's cheap computation of PDG incurs 88.8× less runtime overhead than PalmTree. However, our approach does incur additional computational cost for graph construction. Therefore, it remains an interesting research problem to incorporate system optimization, e.g., caching, to improve the efficiency of the PDG computation during inference.

Training efficiency. We study the training effort (including both pre-training and fine-tuning) of SYMC and PalmTree. Table 8 shows their GPU hours, power, and emitted carbon dioxide estimation when they reach 0.5 F1 score in memory region prediction. We assume the GPU always reaches its power cap

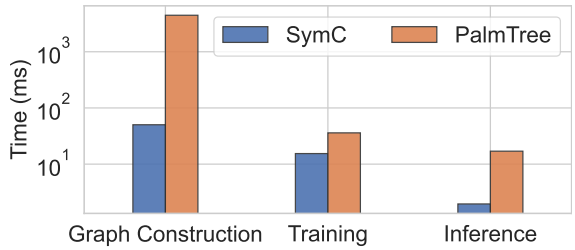


Figure 11: Comparing SYMC to standard Transformer encoder in terms of the additional overhead introduced by constructing PDG, and training and inference with PDG-biased self-attention layers.

Table 8: The resource consumed by training SYMC and other baselines to reach 0.5 F1 score in memory region prediction.

	Time (Hours)	**Power** (kWh)	**Carbon** (CO_2eq)
SYMC	**0.07**	**0.025**	**0.009**
PalmTree-O*	89.67	31.38	11.64

*PalmTree did not disclose its hours for pre-training, so we include the pre-training time (in 10 epochs) based on our own pre-trained PalmTree.

(350W) to estimate an upper bound of the power usage. CO_2eq stands for the carbon dioxide equivalent, a unit for measuring carbon footprints. By being more training efficient, SYMC incurs 1,281$\times$ less total GPU time, power, and emitted carbon dioxide than PalmTree in obtaining the same performance.