Evaluating the Effectiveness of Deep Learning Models for Foundational Program Analysis Tasks

QIAN CHEN, Nanjing University, China
CHENYANG YU, Nanjing University, China
RUYAN LIU, Nanjing University, China
CHI ZHANG, Nanjing University, China
YU WANG*, Nanjing University, China
KE WANG*, Visa Research, USA
TING SU, East China Normal University, China
LINZHANG WANG, Nanjing University, China

While deep neural networks provide state-of-the-art solutions to a wide range of programming language tasks, their effectiveness in dealing with foundational program analysis tasks remains under explored. In this paper, we present an empirical study that evaluates four prominent models of code (*i.e.*, CuBERT, CodeBERT, GGNN, and Graph Sandwiches) in two such foundational tasks: (1) alias prediction, in which models predict whether two pointers must alias, may alias or must not alias; and (2) equivalence prediction, in which models predict whether or not two programs are semantically equivalent. At the core of this study is CodeSem, a dataset built upon the source code of real-world flagship software (*e.g.*, Linux Kernel, GCC, MySQL) and manually validated for the two prediction tasks. Results show that all models are accurate in both prediction tasks, especially CuBERT with an accuracy of 89% and 84% in alias prediction and equivalence prediction, respectively. We also conduct a comprehensive, in-depth analysis of the results of all models in both tasks, concluding that deep learning models are generally capable of performing foundational tasks in program analysis even though in specific cases their weaknesses are also evident.

Our code and evaluation data are publicly available at https://github.com/CodeSemDataset/CodeSem.

CCS Concepts: • **Software and its engineering → Automated static analysis**; • **Computing methodologies → Neural networks**.

Additional Key Words and Phrases: Deep Learning, Alias Analysis, Equivalence Checking

*Corresponding authors.

Authors' addresses: Qian Chen, State Key Laboratory for Novel Software Technology, Department of Computer Science and Technology, Nanjing University, Nanjing, China, qc@smail.nju.edu.cn; Chenyang Yu, State Key Laboratory for Novel Software Technology, Department of Computer Science and Technology, Nanjing University, Nanjing, China, mf21330109@smail.nju.edu.cn; Ruyan Liu, State Key Laboratory for Novel Software Technology, Department of Computer Science and Technology, Nanjing University, Nanjing, China, mf21330053@smail.nju.edu.cn; Chi Zhang, State Key Laboratory for Novel Software Technology, Department of Computer Science and Technology, Nanjing University, Nanjing, China, zhangchi_seg@smail.nju.edu.cn; Yu Wang, State Key Laboratory for Novel Software Technology, Department of Computer Science and Technology, Nanjing University, Nanjing, China, yuwang_cs@nju.edu.cn; Ke Wang, Visa Research, Palo Alto, USA, kewang@visa.com; Ting Su, Shanghai Key Lab of Trustworthy Computing, Software Engineering Institute, East China Normal University, Shanghai, China, tsu@sei.ecnu.edu.cn; Linzhang Wang, State Key Laboratory for Novel Software Technology, Department of Computer Science and Technology, Nanjing University, Nanjing, China, lzwang@nju.edu.cn.

ACM 2475-1421/2024/4-ART112
https://doi.org/10.1145/3649829

ACM Reference Format:
Qian Chen, Chenyang Yu, Ruyan Liu, Chi Zhang, Yu Wang, Ke Wang, Ting Su, and Linzhang Wang. 2024. Evaluating the Effectiveness of Deep Learning Models for Foundational Program Analysis Tasks. *Proc. ACM Program. Lang.* 8, OOPSLA1, Article 112 (April 2024), 29 pages. https://doi.org/10.1145/3649829

1 INTRODUCTION

Riding on the stunning advancements of deep learning in recent years, deep neural networks have demonstrated remarkable capabilities in inferring semantic programs properties (*e.g.*, bug detection [Allamanis et al. 2018; Wang et al. 2020], program repair [Chen et al. 2019; Dinella et al. 2019], code documentation generation [Feng et al. 2020; Wang and Su 2020]). In light of the growing integration of neural technologies into the programming language (PL) research, a key question remains under addressed: *how far can we push deep learning models in the field of program analysis; are they even capable of solving tasks that are traditionally in the realm of foundational static analysis?* By "foundational", we mean static analysis that serves the foundation for various client analyses. The response to this question has a profound impact on the programming language community. If the answer is yes, neural networks can become a viable alternative for developing static analysis applications perhaps with their own advantages (*e.g.*, easier to create, higher efficiency). Conversely, if the answer is no, it can shed light on the deficiencies of existing code models, motivating future research to build finer models for program analysis. In this paper, we aim to answer this question. Specifically, we undertake a systematic and rigorous evaluation on the efficacy of deep neural networks in performing foundational program analysis tasks.

We face two primary challenges: first, (1) how to define prediction tasks that correspond to the foundational tasks in program analysis; second, (2) how to curate a dataset at scale that serves the tasks being defined. For the first challenge, we draw on two foundational problems in PL research — alias analysis and equivalence checking — to derive alias prediction and equivalence prediction respectively. Since the two analyses are among the most well-established, important static analysis, with an extensive literature and broad range of applications, we design our evaluation task in the mold of alias analysis and equivalence checking. Alias prediction requires models to classify if two pointers must alias, may alias[1] or must not alias. Equivalence prediction is about predicting whether or not two programs are semantically equivalent. It is worth mentioning that equivalence prediction differs in fundamental ways from clone detection [Horwitz 1990], a well-known problem in software engineering. Specifically, much of the clone detection work focuses on syntactic similarity of source code [Golubev et al. 2021; Jiang et al. 2007; Kamiya et al. 2002; Sajnani et al. 2016; Wang et al. 2018; Yuan and Guo 2012] whereas equivalence checking requires models to predict the semantic equivalence of programs, thus is clearly a better fit to the central theme of this work.

To overcome the second challenge, our key idea is to leverage the results of the corresponding program analysis methods so that models can be trained on potentially unlimited amount of labeled data without any human effort in data labeling. However, the quality of such labeled data poses a risk given that results of any program analysis method are almost certain to contain noise (since static analysis must involve approximation due to Rice's theorem, hence the issue of false positives; and dynamic analysis can not reason about all possible program behavior, hence the issue of false negatives). In response, we propose a key adjustment to a prominent, two-stage learning encompassing pre-training and fine-tuning. This adjustment allows models to exploit the labeled data despite its inherent noise.

Our training approach consists of three phases: generalized pre-training, specialized pre-training, and fine-tuning. While generalized pre-training and fine-tuning directly correspond to the two

[1]To better align with the goals of this work, we have modified the conventional definition of the may-alias relation in PL literature. Specifically, it now represents a disjoint aliasing relation to must-alias (Section 2.1).

stages in pre-training and fine-tuning, specialized pre-training, a new, in-between stage, trains models to approximate downstream prediction tasks using the results of the corresponding program analysis methods. Even though data used at this stage carries noise, models can take advantage of it to learn a coarse decision boundary. This boundary is subsequently refined during the fine-tuning stage, utilizing clean, ground-truth data. Take the alias prediction as an example. In specialized pre-training, we deploy models to learn from results of two alias analysis [Kastrinis et al. 2018; Zheng and Rugina 2008], which produce three classes of alias pairs: ground-truth must-alias and must-not-alias, and noisy may-alias (details are provided in Section 5.2.1). By learning from real must-alias, must-not-alias, and noisy may-alias, models first solve an easier sub-task of separating must-alias and must-not-alias in the specialized pre-training stage. Then, models conquer the entire task by refining the decision boundary between may-alias and must-alias/must-not-alias using ground-truth data in the fine-tuning stage. We note that our training workflow, especially the last two phases, resembles an influential learning strategy called curriculum training [Bengio et al. 2009], under which models are trained on data shuffled in the ascending order of difficulty levels. To a certain degree, the correlation to curriculum training validates of our training approach.

Our Goals. In this work, we set out to explore what all is achievable within the machinery of deep neural networks in program analysis. Given that deep learning models are fundamentally based on pattern recognition, an approach that is not considered particularly suitable for simulating classical static analysis algorithms, we believe such an exploration is a worthwhile pursuit. Technically, our primary goal is to evaluate the performance of four influential code models (*i.e.*, CuBERT, CodeBERT, GGNN, and Graph Sandwiches) in two foundational tasks of program analysis: alias prediction and equivalence prediction. It is worth noting that our aim is *not* to propose novel solutions to alias or equivalence prediction, or even replace existing algorithms for alias analysis or equivalence checking with deep neural networks. Both of these can be steps too far given the current landscape of deep models of code, and there is no sufficient evidence that deep neural networks are even applicable to the kind of foundational static analysis considered in this paper. As a secondary goal, we aim to benchmark the performance of graph models (*i.e.*, GGNN and Graph Sandwiches) *w.r.t.* different graph representations of code including Abstract Syntax Tree-based, Control Flow Graph-based, and Program Dependence Graph-based.

Advantages over Existing Benchmarks. Compared to existing benchmarks for code models [Husain et al. 2019; Lu et al. 2021; Puri et al. 2021], our work offers two crucial advantages. First, our dataset consists of code extracted from programs of real-world flagship software rather than coding platforms at which code are written to solve specific algorithmic problems. It is clear their specialized, non-standard programs (*e.g.*, implement division with subtraction only; test if a given number is a palindrome) are not particularly relevant to real programming settings to which code models are designed to apply. Therefore, results of code models on those benchmarks are unlikely to generalize to useful end goals in the real world. Second, those works evaluate deep learning models in tasks that are well-explored by prior works (*e.g.*, variable misuse prediction [Allamanis et al. 2018], method name prediction [Alon et al. 2019]). Therefore, their findings provide limited new insights into the strengths and weaknesses of deep learning models. In contrast, we define alias prediction and equivalence prediction, derived from two foundational static analysis, as the prediction tasks in our evaluation.

Summary of Main Findings. Our results show that all models are accurate in both prediction tasks, in particular, CuBERT, the most accurate model, achieves 89% accuracy in alias prediction and 84% accuracy in equivalence prediction. We observe that program representation is a key factor to the performance of graph models. The accuracy of the exact same model can vary significantly depending on the type of graphs in which programs are represented in both prediction tasks.

However, we find that the program dependency graph-based representation enables GGNN and Graph Sandwiches to achieve their highest accuracy in both prediction tasks. This underscores the power and versatility of this program representation for graph models. Regarding the comparison between our training approach and the existing pre-training and fine-tuning approach, models trained with our approach are almost always more accurate than those undergoing pre-training and fine-tuning, demonstrating the effectiveness and generality of our approach to training models towards solving the foundational program analysis tasks.

Next, we perform an in-depth analysis on models' results in order to gain a deeper understanding of what models have learned in the two prediction tasks. In alias prediction, we examine the nature of the alias pairs that models detected from the locality aspect. That is, whether the detected alias pairs are generally very local or they span a number of statements. Our analysis reveals that all models achieve sustained accuracies even when the instructions that establish aliasing relations become increasingly distant. In equivalence prediction, we first investigate if models have resorted to a simple, template-matching approach to determining the semantic equivalence of two programs. That is, did models simply memorize equivalent program pairs in the training set into templates, which they then use to match test pairs by accommodating shallow, syntactic variations. However, we find that only 11% of equivalent program pairs in the test set can be matched to a pair of training programs with minor syntactic modifications, indicating the learning capacity of code models that goes far beyond such a template-matching approach. To further demonstrate the challenges that code models have overcome to achieve high accuracies, we consider three well-established equivalence checking tools (*i.e.*, *trace alignment* [Churchill et al. 2019], *ARDIFF* [Badihi et al. 2020], and *Rêve* [Felsing et al. 2014]) as reference points, in particular, we assess their performance on CodeSem. Quite surprisingly, all tools perform exceedingly poorly, with the top-performer achieving under 25% accuracy, and more importantly, we find that all code models have coped comfortably with the very issues that severely hinder those equivalence checking tools (*e.g.*, limitations in aligning execution traces, difficulties in solving complicated path constraints).

Overall, we conclude that in general, the evaluated code models are capable of handling even the foundational tasks in program analysis. On the other hand, we also identify specific areas in which models still have significant headroom to improve. For example, models can be imprecise in dealing with pointers with complicated def-use patterns in alias prediction; in addition, they also display limited capability to perform sophisticated reasoning about the program behavior in equivalence prediction. With regard to the weaknesses of code models, we point out potential directions of future research for continuously improving deep learning models in programming language tasks.

In summary, we make the following contributions in this paper:

(1) We define two learning tasks — alias prediction and equivalence prediction — for evaluating code models in performing foundational program analysis tasks.
(2) We assemble CodeSem, a dataset for alias and equivalence prediction, using code exclusively extracted from the program of real-world flagship software.
(3) We propose a general, novel, three-stage training approach that leverages the results of program analysis tools to train models towards alias prediction and equivalence prediction.
(4) We present the results of an extensive quantitative and qualitative evaluation of four prominent models of code (*i.e.*, CuBERT, CodeBERT, GGNN, and Graph Sandwiches) in alias and equivalence prediction.

2 PREDICTION TASKS

This section defines alias prediction and equivalence prediction, two downstream tasks that we use to evaluate deep learning models.

2.1 Alias Prediction

Alias prediction is derived from alias analysis, which checks whether two pointer variables refer to the same memory location at a particular program point. Effective alias analysis plays an essential role in nearly all program analyses for object-oriented programs [Sridharan et al. 2013]. For example, computing a precise inter-procedural control-flow graph, a prerequisite for many program analyses, often requires significant reasoning on pointer aliasing to resolve virtual dispatch. Furthermore, any program analysis attempting to discover non-trivial properties of an object must reason about mutations to that object through pointer aliases. As a foundation for many client analyses, alias analysis facilitates typestate analysis [Phulia et al. 2020], taint analysis [Tripp et al. 2013], and value flow analysis [Shi et al. 2018].

Unlike the classical may-alias analysis (for determining may-alias or must-not-alias pairs) or must-alias analysis (for identifying specifically must-alias pairs), alias prediction classifies if two pointer variables must alias, may alias, or must not alias all at once, a more elegant approach to identifying aliasing relations. While the conventional definition of must- and must-not-alias [Altucher and Landi 1995; Balatsouras et al. 2017; Fink et al. 2006] still applies to our prediction task (Definition 2.1 and 2.2), we make a minor modification to the definition of may-alias [Altucher and Landi 1995; Horwitz 1997], in particular, we add an additional constraint (underlined in Definition 2.3) to exclude must-alias from may-alias.

Definition 2.1. (*Must Alias*) Two variables v_1 and v_2 is must-alias at a program point n if for all executions to n, v_1 and v_2 refer to the same location at n.

Definition 2.2. (*Must-not Alias*) Two variables v_1 and v_2 is must-not alias at a program point n if v_1 and v_2 do not refer to the same location at n on any execution to n.

Definition 2.3. (*May Alias*) Two variables v_1 and v_2 is may-alias at a program point n if for some $\underline{\textit{but not all}}$ executions to n, v_1 and v_2 refer to the same location at n.

2.2 Equivalence Prediction

We derive equivalence prediction from equivalence checking, a long-standing problem in computer science. It is an important building block for many client applications. For example, in the setting of compiler verification, equivalence checking is employed to verify the correctness of transformations performed by compilers [Necula 2000; Sewell et al. 2013; Tate et al. 2009]; superoptimizers also rely on equivalence checker to ensure that the optimized programs maintain the same semantics of the original programs [Churchill et al. 2019, 2017; Dahiya and Bansal 2017]. In addition, equivalence checking has been applied to problem domains like program synthesis [Schkufza et al. 2013] and code refactoring [Ramos and Engler 2011]. A central issue to equivalence checking is the definition of semantic equivalence. In this work, we follow Churchill et al. [2019]'s formalization.

Definition 2.4. (*Semantic Equivalence*) Two programs p_1 and p_2 are semantically equivalent if, when p_1 and p_2 start in identical machine states (*e.g.*, registers, stack, heap), (1) they have identical output registers and heap states; or (2) they encounter the same run-time error (or loop forever).

The output registers and heap states together reflect the state in which machine ends after executing a program. We ignore stack because it is used for allocating local, temporary variables.

3 TRAINING APPROACH

In this section, we first introduce the code models selected for this study, followed by a detailed presentation of our training workflow.

3.1 Models of Code

CuBERT. Built upon BERT [Devlin et al. 2019] (Bidirectional Encoder Representations from Transformers), the core of CuBERT [Kanade et al. 2020] is a multi-layer bidirectional Transformer [Vaswani et al. 2017]. CuBERT is a natural application of BERT in programming language tasks. Similar to BERT, CuBERT is first pre-trained on a larger dataset to learn general code embeddings and then fine-tuned on a smaller dataset for specific downstream tasks such as variable-misuse classification, wrong binary operator prediction, and exception type prediction.

CodeBERT. CodeBERT [Feng et al. 2020] is among the latest models of code based on BERT. Unlike CuBERT, CodeBERT is a bimodal code model for programming language and natural language. A main point of novelty is its capability to pre-train with both bimodal data (*i.e.*, code-text pairs) and unimodal data (*i.e.*, code or text alone). CodeBERT achieves state-of-the-art results on several cross-lingual prediction tasks like code search and code document generation.

GGNN. GGNN (Gated Graph Neural Network) [Allamanis et al. 2018], a variant of Graph neural network, has been widely used in machine learning models of source code. Central to GGNN is the message-passing mechanism, and the way it works is at each message-passing step, every node sends messages to its neighbors and summarizes messages received from its neighbors which it uses together with its prior state to compute the new state with a GRU cell [Cho et al. 2014].

Graph Sandwiches. Graph Sandwiches [Hellendoorn et al. 2019] aims to replicate the strength of sequence models in learning global data properties for a base graph model. Specifically, it incorporates sequential layers into message-passing steps of graph neural networks such that the node embeddings are not solely learned from the message-passing layers, the way that node representations are learned in a typical GNN, but also from sequence models like RNN [Rumelhart and McClelland 1987] or Transformer.

3.2 Training Workflow

Our training workflow consists of three stages: generalized pre-training, specialized pre-training, and fine-tuning. Below, we give the details of each training stage.

3.2.1 Generalized Pre-Training. The generalized pre-training stage corresponds to the pre-training phase in the widely-used, two-stage learning approach, namely generative pre-training followed by discriminative fine-tuning. The goal of this stage is to learn general, foundational properties of source code independent of downstream prediction tasks. To design the task at the generalized pre-training stage, we can simply adopt the pre-training tasks used in sequence models. Specifically, CuBERT is pre-trained with *masked sequence token prediction* and *next sentence prediction* concurrently [Devlin et al. 2019; Kanade et al. 2020]. The former is to mask some percentage of tokens in an input program at random for the model to predict. The latter is for the model to predict if two given sentences follow each other. A sentence here refers to a logical code line which is the shortest sequence of contiguous lines that makes up a legal statement. For CodeBERT, we use *replaced token detection* [Clark et al. 2020], in which models learn to detect whether a token is original or counterfeit generated by language models. We note that replaced token detection is the only pre-training task of CodeBERT that can work with unimodal data (*i.e.*, only code in our setting). The other pre-training task, which deals with bimodal data (*i.e.*, text and code), is not applicable to our setting.

As GGNN and Graph Sandwiches lack well-established pre-training tasks, we leverage the pre-training tasks commonly used in sequence models. To this end, we introduce *masked graph node prediction*, based on CuBERT's masked sequence token prediction task. This task involves randomly masking some nodes in a program graph, which the models must then predict. For simplicity, we only mask nodes that directly correspond to tokens of a program, such as terminal nodes in an abstract syntax tree. Furthermore, we only consider nodes that have previously appeared in the program prior to the masking locations. To replicate the setup of masked sequence token prediction, we mask 15% of the nodes in each program graph for masked graph node prediction. Additionally, we draw inspiration from CuBERT's next sentence prediction to develop *adjacent edge prediction* for GGNN and Graph Sandwiches. Specifically, we train these models to predict the type of edge connecting two randomly selected nodes from a graph. We also note that in this task, the absence of an edge is considered a prediction class.

3.2.2 Specialized Pre-Training. Specialized pre-training initiates the process of adapting model parameters learned in the generalized pre-training stage towards the specific prediction tasks. However, unlike fine-tuning which relies on clean, ground-truth data to facilitate knowledge transfer, specialized pre-training merely aims to learn a coarse decision boundary for the specific prediction task with data that contains an acceptable amount of noise. The advantage of this approach is that it allows models to exploit potentially unlimited amounts of labeled data produced by corresponding program analysis methods during the specialized pre-training stage. This, in turn, can help to more effectively adapt the learned model to downstream tasks during fine-tuning stage.

For Alias Prediction, we propose a task called *noisy alias prediction*, where models learn from the results of a sound may-alias [Zheng and Rugina 2008] and sound must-alias analysis [Kastrinis et al. 2018]. As explained in Section 1, this task involves a two-step process in which models initially (1) learn a precise decision boundary between must- and must-not-alias given that data in these two categories is ground-truth. Subsequently, (2) models learn a rough decision boundary between must- and may-alias, and must-not- and may-alias, respectively given that may-alias data is noisy. These decision boundaries, those learned in Step (2) in particular, will be refined in the later fine-tuning stage. Because separating must- and must-not-alias (which is the main objective of the specialized pre-training stage) is an easier sub-task in alias prediction since they are the two extremes of aliasing relations, and differentiating between must-/must-not-alias and may-alias (which is the main objective of fine-tuning stage) poses a greater challenge. Specific to GGNN and Graph Sandwiches, our pre-training workflow (generalized pre-training followed by specialized pre-training) aligns with Weihua et al. [2020]'s approach for pre-training graphs models in general: pre-training tasks should enable graph models to capture both local (*e.g.*, nodes and edges) and global (*e.g.*, graph-level) properties. Clearly, the former is the objective of generalized pre-training while the latter is the objective of specialized pre-training in our training approach.

For Equivalence Prediction, we design *noisy functional equivalence prediction* task where models learn to predict whether or not two programs have the same input-output pair. In this task, models learn from results of a testing method for functional equivalence [Jiang and Su 2009]. Since this testing method can only refute the equivalence of two programs, it yields ground-truth data for inequivalent programs and noisy data for equivalent programs. Using these labeled data, models aim to learn a coarse separation of programs *w.r.t.* (a noisy version of) functional equivalence, which later will be refined *w.r.t.* (the exact version of) semantic equivalence.

3.2.3 Fine-Tuning Tasks. After the generalized and specialized pre-training stages, models undergo fine-tuning tasks that directly correspond to the prediction tasks in this evaluation — alias prediction or equivalence prediction. During the fine-tuning stage, only ground-truth data is used to complete the adaption of model parameters learned from the two pre-training stages towards the downstream prediction tasks. Figure 1 depicts the training workflow for all evaluated models. Specifically, CuBERT undergoes concurrent training in the generalized pre-training stage for masked sequence token prediction and next sentence prediction. CodeBERT is trained with replaced token detection during the generalized pre-training stage. Both models are then trained with noisy alias prediction or

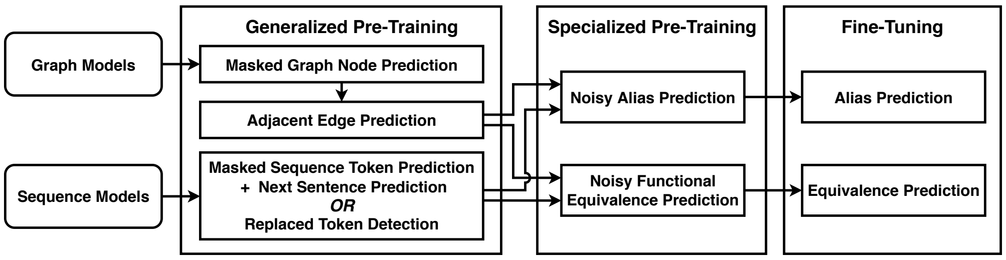


Fig. 1. The complete training workflow.

noisy functional equivalence prediction during the specialized pre-training stage before being fine-tuned on each prediction task. GGNN and Graph Sandwiches follow the same training workflow, starting with masked graph node prediction followed by adjacent edge prediction. In practice, this order yields minor advantages over models trained concurrently or in reverse order. Afterward, both models are trained with noisy alias prediction or noisy functional equivalence prediction before being fine-tuned on alias or equivalence prediction tasks.

4 MODEL ARCHITECTURES FOR ALIAS AND EQUIVALENCE PREDICTION

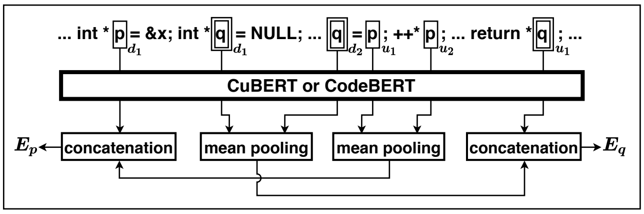
In this section, we explain how we utilize the existing architecture of each code model to solve the two prediction tasks.

Model Architecture for Alias Prediction. Let $\mathcal{F}$ be a single function or a set of functions in a call chain; let v_1 and v_2 denote two variables for which an aliasing relation is predicted at program point n within $\mathcal{F}$, we design models to take a 3-tuple input in the form of $<\mathcal{F}, Def_{v_1}, Def_{v_2}>$ where Def_{v_1} and Def_{v_2} are the sets of definitions of v_1 and v_2 that are live at n. The reason we only consider live definitions of v_1 and v_2 is because they are the instructions that exclusively determine the aliasing relations of v_1 and v_2. Since there may be distinct definitions of v_1/v_2 that can be live at n in the context of $\mathcal{F}$'s control flow (*e.g.*, both line 4 and 6 can be a live definition of q at line 9 in Figure 2a), we take into account all of them. Since the way variables are used may also provide useful information, we introduce another design that incorporates uses of variable definitions from the def-use chain [Stoltz et al. 1994]. In this design, models will take a 5-tuple input in the form of $<\mathcal{F}, Def_{v_1}, Use_{v_1}, Def_{v_2}, Use_{v_2}>$ as input, where Use_{v_1}/Use_{v_2} is a set of uses from the def-use chain of Def_{v_1}/Def_{v_2}. Figure 2b illustrates the architecture of sequence models for alias prediction using the code in Figure 2a as an example. In both designs, the first element of the input tuple, $\mathcal{F}$, is represented by token sequence, which is fed into sequence models to generate the final hidden vector of each token in $\mathcal{F}$. Regarding the embeddings of Def_{v_1} and Def_{v_2}, we extract the final hidden vector of the token on the Left Hand Side (LHS) of Def_{v_1} and Def_{v_2} (*e.g.*, $\boxed{\mathrm{p}}_{d_1}$ in Figure 2b for the definition of p at line 3 in Figure 2a). Our rationale is that LHS can represent the whole definition because of the way tokens communicate through the self-attention mechanism. If Def_{v_1} or Def_{v_2} contains multiple definitions, we perform mean pooling over the embeddings of every definition's LHS (*e.g.*, mean-pooling the embedding of $\boxed{\boxed{\mathrm{q}}}_{d_1}$ and $\boxed{\boxed{\mathrm{q}}}_{d_2}$ as depicted in Figure 2b). If variable uses are not considered, then we simply feed a softmax regression layer with the concatenation of embeddings of Def_{v_1} and Def_{v_2} to predict the aliasing relation between v_1 and v_2. Otherwise, the following steps will be taken.

First, we compute the embedding of Use_{v_1} or Use_{v_2} by mean-pooling the hidden state of every use of Def_{v_1} or Def_{v_2}, where each use is represented by a token that indicates the usage point of v_1 or v_2. For instance, $\boxed{\mathrm{p}}_{u_1}$ and $\boxed{\mathrm{p}}_{u_2}$ are the two use points of the definition of p at line 3 in Figure 2a, and their embeddings are mean-pooled to produce the embedding of the uses of the definition

```
int main(int argc, char** argv) {
  int x = argc;
  int *p = &x;
  int *q = NULL;
  while (x < 5) {
    q = p;
    ++*p;
  }
  return *q;
}
```

(a) An example program having two pointers p and q.



(b) The architecture of sequence models for predicting the aliasing relation between p (annotated as $\boxed{\text{p}}$) and q (annotated as $\boxed{\boxed{\text{q}}}$).

Fig. 2. The architecture of sequence models for alias prediction.

of p. Next, we concatenate the embedding of Use_{v_1} or Use_{v_2} to that of Def_{v_1} or Def_{v_2} to produce the final embedding of v_1 (denoted by E_{v_1}) or v_2 (denoted by E_{v_2}). For example, concatenating the embedding resulted from the mean-pooling of $\boxed{\text{p}}_{u_1}$ and $\boxed{\text{p}}_{u_2}$ with $\boxed{\text{p}}_{d_1}$ produces E_p as depicted in Figure 2b. Finally, we use a softmax regression layer with the concatenation of E_{v_1} and E_{v_2} to predict the aliasing relation between of v_1 or v_2. It is worth mentioning that we have experimented with alternatives to mean-pooling in both cases (for computing embeddings of definitions and uses) such as max pooling, weighted sum (*i.e.*, soft attention), and none of which display a notable improvement in either case. For graph models, after completion of the message-passing process, we extract the embedding of nodes that correspond to the LHS of every definition in Def_{v_1} and Def_{v_2} and every use point of Def_{v_1} and Def_{v_2}, then we follow the same procedure described above for sequence models to predict the aliasing relation between v_1 and v_2.

Model Architecture for Equivalence Prediction. Given a tuple <$\mathcal{F}_1, \mathcal{F}_2$> representing the two input programs (here $\mathcal{F}$ stands for the same meaning as in alias prediction), all models take the same approach: computing the hidden vector of each token (*resp.*, node) in the sequence (*resp.*, graph), and then performing mean pooling over the hidden vectors of all tokens (*resp.*, nodes) to obtain a single embedding for $\mathcal{F}_1$ or $\mathcal{F}_2$. Again, Other methods such as max pooling or weighted sum were also experimented with, but did not show notable improvement. Similar to alias prediction, we add a one-layer softmax regression to predict the equivalence of $\mathcal{F}_1$ and $\mathcal{F}_2$ using the concatenation of their embeddings.

5 THE DATASET

We compile three distinct datasets, including generalized pre-training dataset, specialized pre-training dataset, and fine-tuning dataset to suit our training approach. Regarding the programming languages in which we build our datasets, we take into account the following factors: (1) the analysis of source code, both dynamic and static, in the language to be chosen should be well-supported by existing tool-chains, infrastructure, *etc.*, such that the workload that we undertake in conducting such a large-scale, extensive evaluation can be reduced to a minimum; (2) the source code in the language to be chosen should present sufficient challenges in both prediction tasks. For example, in the case of alias prediction, we require languages to be chosen to feature complicated pointer operations (*e.g.*, address-of assignment, load and store operations, pointer arithmetic). Because C and C++ are likely the only languages that satisfy our criteria, and they are also foundational, widely-used programming languages, we build our datasets with C/C++ code.

We select fourteen open-source software for our study: Linux Kernel, GCC, MySQL, Git, tmux, Redis, curl, LevelDB, H2O, libgit2, The Silver Searcher, Protocol Buffers, aria2, and fish. They range from mid-scale (with tens of thousands lines of code) to large-scale programs (with hundreds of thousands or even millions of lines of code), and all of them are well-established (with decades-long

Table 1. Statistics of CodeSem.

Generalized	Alias Prediction Dataset						Equivalence Prediction Dataset			
Pre-training	Specialized Pre-training			Fine-tuning			Specialized Pre-training		Fine-tuning	
Functions	Must.	May.	Must-not.	Must.	May.	Must-not.	Equiv.	Inequiv.	Equiv.	Inequiv.
200,013	10,204	10,030	10,613	11,647	11,739	11,806	66,403	66,538	10,655	10,861

Table 2. The complexity of CodeSem evidenced by the length (*resp.*, size) of token sequences (*resp.*, graphs) rendered by programs in the dataset. Overall, CodeSem is sufficiently complex to ensure the difficulty of the alias and equivalence prediction task.

Datasets		#tokens in sequences				#nodes in graphs			
		min	max	mean	median	min	max	mean	median
Generalized Pre-training		10	23192	127	68	10	75910	203	71
Alias	Specialized Pre-training	20	10858	730	335	36	25377	928	369
	Fine-tuning	15	11078	738	349	38	23604	786	292
Equivalence	Specialized Pre-training	67	20129	1544	1036	40	45208	2451	1332
	Fine-tuning	56	18729	1382	1131	40	40278	2158	1602

history). Table 1 in the supplemental material provides the details of each project. The source code of those software implement a broad range of functionalities such as data transmission, memory management, and cross-compilation, which makes our dataset diverse. All software contribute to the generalized pre-training, the specialized pre-training, and the fine-tuning dataset, in addition, the three datasets are extracted from different portions of the program of each software to avoid duplicates. From each project, we collect roughly the same number of samples for each prediction class for all three datasets. Table 1 gives an overview of CodeSem. In the subsequent sections, we explain in detail how the three datasets are created.

5.1 Dataset of Generalized Pre-Training

The training data for all generalized pre-training tasks can be easily generated from any valid code. Specifically, we extract all files that can be parsed by Clang [Lattner 2008] from the codebase of each software mentioned earlier, which resulted in 120,814 files in total. From them, we randomly pick 200,013 functions. Every function is used to generate data for all generalized pre-training tasks for all evaluated models. The first row in Table 2 gives the length (*resp.*, size) of token sequences (*resp.*, graphs) of the data used in the generalized pre-training tasks.

5.2 Dataset of Specialized Pre-Training and Fine-Tuning

5.2.1 Alias Prediction. We adopt a sound may- and must-alias analysis to obtain data points for the specialized pre-training dataset. A sound may-alias analysis over-approximates aliasing relations, ensuring that all potential alias pairs are found [Sridharan and Bodík 2006; Yong et al. 1999; Zheng and Rugina 2008]. As a result, the results of a sound may-alias consist of ground-truth must-not-alias and noisy may-alias which are almost certain to contain must-not-alias. In contrast, a sound must-alias analysis computes an under-approximation of the aliasing relations that are guaranteed to hold [Kastrinis et al. 2018]. In other words, the must-alias produced by the sound must-alias analysis is a subset of all ground-truth must-alias pairs. Therefore, a combination of the may- and must-alias analysis results in three types of aliasing pairs: ground-truth must-alias pairs (which are directly found by the must-alias analysis), ground-truth must-not-alias pairs (which are directly found by the may-alias analysis), and noisy may-alias pairs, in particular, noisy may-alias pairs are obtained by the subtraction of must-alias pairs (found by the must-alias analysis) from may-alias pairs (found by the may-alias analysis).

As we explained above, may-alias pairs (found by the may-alias analysis) also include must-not-alias pairs; and must-alias pairs (found by the must-alias analysis) only account for a subset of all real must-alias pairs within may-alias pairs, therefore, the subtraction of must-alias pairs from may-alias pairs results in a mixture of all three types of alias pairs. For this reason, we term may-alias pairs "noisy". The ground-truth must-alias, must-not-alias pairs, and noisy may-alias directly constitute the specialized pre-training dataset. To obtain ground-truth data for the fine-tuning dataset, we begin by running the same set of analyses (on a different portion of the program of each selected software) used to generate the specialized pre-training dataset. From the results, we select the must- and must-not-alias that are guaranteed to be correct, and then involve human experts to label the may-alias that the analysis can not precisely determine.

Specialized Pre-training Dataset. For the may-alias analysis, we adopt the method proposed by Zheng and Rugina [2008], which is implemented in LLVM [Lattner and Adve 2004] as an inter-procedural, context-, flow-, and field-insensitive analysis. This approach strikes a balance between precision and scalability and is considered state-of-the-art in alias analysis. For the must-alias analysis, we refer to the work of Kastrinis et al. [2018], who propose a novel data structure based on equivalence classes as the backbone of their must-alias analysis. We realize their approach into an inter-procedural, context-, flow-, and field-sensitive must-alias analysis. Below, we give a detailed presentation on how to create the specialized pre-training dataset based on those two analyses.

First, we run the must-alias analysis to identify must-alias relations. This step generates alias classes, each of which contains variables that must alias with each other. Therefore, to create must-alias pairs, we simply combine two variables from the same alias class. Second, we run the may-alias analysis integrated in LLVM to obtain may-alias and must-not-alias information. Similarly, LLVM generates alias sets where variables within each set may point to the same memory location. Thus, combining two variables from different aliasing sets results in must-not-alias pairs. To construct may-alias pairs, we ensure that we do not reuse variables that are already identified with must-alias relation. Specifically, we filter LLVM's alias sets so that each set only contains one variable from the same alias class generated by our must-alias analysis. Among the remaining variables in each of LLVM's alias sets, we combine two variables from the same set to create may-alias pairs. Regarding the must-alias analysis, we analyzed 54,421 functions out of 120,814 files. These functions do not overlap with the 200,013 functions used to generate the generalized pre-training dataset. As a result, we identified 32,196 equivalence classes. On the same set of functions, LLVM produced 69,809 alias sets for the may-alias analysis. To maintain the diversity of our alias pairs, we make each variable appear exactly once among the three classes of alias pairs. In addition, to avoid duplication between the specialized pre-training and fine-tuning dataset, we randomly selected 26,899 functions (out of 54,421) to construct the specialized pre-training dataset and held out the rest for the fine-tuning dataset. In the end, we collected 10,204 must-alias pairs, 10,030 may-alias pairs, and 10,613 must-not-alias pairs in the specialized pre-training dataset for alias prediction.

Fine-tuning Dataset. To obtain ground-truth data for the fine-tuning dataset, we utilize the results of the sound may- and must-alias analysis on the holdout set of functions (27,522 in total), and then manually label the data points that the two analysis can not precisely determine. Because the must-alias and must-not-alias pairs are guaranteed to be correct, we focus on verifying the label of may-alias pairs. We engaged the assistance of 32 PhD students for the labeling task, each of whom is familiar with alias analysis. Below, we give the details of the manual labeling process.

We assigned every may-alias pair to two PhD students. When labeling a may-alias pair, human raters see the entire codebase of the corresponding project for the sake of precision of the labeling process. For difficult alias pairs (*e.g.*, those involving global information) where human raters disagree, we design a separate procedure to resolve their labels. For circumstances where raters disagree whether an alias pair is may-alias or must-alias, we first instrument all program paths that one rater (*i.e.*, the one who gives the may-alias label) thinks the two pointers are not alias, in particular, we log the memory locations that the two pointers point to on those paths. Then, we use AFL [Zalewski 2016] to fuzz test the program. After the fuzzer finishes, we check via the log if the two pointers ever point to different memories on any of these instrumented paths. If so, they are may-alias, otherwise, we check if the fuzzer indeed covered all instrumented paths, if so, we label them must-alias; if not, we simply discard this alias pair because its label can not be determined. Similarly, when human raters disagree on whether an alias pair is may-alias or must-not-alias, we check if the two pointers ever point to the same memories on the paths which one rater (the one who gives the may-alias label) thinks they are alias. If we find that the two pointers point to the same memory on at least one instrumented path, they are may-alias, otherwise, we label them must-not-alias or discard them depending on whether the fuzzer has covered all instrumented paths.

Based on the above-mentioned approach for determining the labels of alias data, we now validate if the construction of ground truth matches the prediction samples that models would work with. We address each of the three classes of alias pairs separately for discussion. (1) For must-not-alias pairs identified by sound may-alias analysis, we configure LLVM to produce all intermediate analysis steps. In particular, whatever code are used by the analysis will be included in the construction of the data points, in other words, models consume the same information that the analysis does for determining the must-not alias relations. (2) Similarly, for must-alias pairs identified by the sound must-alias analysis, we also include all code consumed by the analysis to construct the data points. This means again models observe the same information in a data point that the analysis uses to determine the must-alias relations. (3) For each noisy may-alias pair, human raters, assisted by fuzzers, identify the scope of code from which the ground-truth can be determined. All code falling within this scope will be included in the construction of the data point. Consequently, in all cases, we ensure there is no mismatch between model predictions and the construction of ground truth.

The entire labeling process took approximately five months. Overall, the average inter-rater reliability is good, with a kappa score of 0.86. In the end, we collected 11,647 must-alias pairs, 11,739 may-alias pairs, and 11,806 must-not-alias pairs for the fine-tuning dataset. The second and third rows of Table 2 present the length (*resp.*, size) of token sequences (*resp.*, graphs) for alias prediction. Clearly, these data points are sufficiently complex to ensure the difficulty of the alias prediction task. As shown in Table 1, the generalized pre-training dataset contains substantially more data points than the specialized pre-training and fine-tuning datasets because small programs that do not contain any aliases are included in the generalized pre-training dataset but are ineligible for the specialized pre-training and fine-tuning datasets.

5.2.2 Equivalence Prediction. We build an automated data pipeline to create the specialized pre-training dataset. The pipeline consists of two steps: (1) extracting code fragments from the program of selected software; (2) testing their functional equivalence dynamically. Regarding the fine-tuning dataset, we first run the same data pipeline on a different portion of the program of each selected software, and then manually confirm if the code tested to be functionally equivalent is semantically equivalent, considering that functionally inequivalent code is knowingly semantically inequivalent.

Specialized Pre-training Dataset. In our efforts to create a specialized pre-training dataset for equivalence prediction, we seek out code that are functionally equivalent, meaning, they have the same input-output pairs. To achieve this, we rely on EqMiner [Jiang and Su 2009], which adopts a random-testing-based approach to extract code fragments that are functionally equivalent. For all selected software programs, EqMiner generates 3,680 equivalent sets from 15,067 functions in 120,814 C/C++ files. Again, these functions do not overlap with any of the 200,013 functions reserved for the generalized pre-training dataset. Code fragments within an equivalent set are likely equivalent to each other, while code fragments from different equivalent sets are known to be inequivalent. Based on the output of EqMiner, collecting equivalent code pairs is straightforward: combining any two elements in each equivalent set. To form inequivalent code pairs, we combine elements from different equivalent sets. Following our approach to creating the dataset for alias prediction, we make each code fragment appear only once in either equivalent code pairs or inequivalent code pairs to ensure the diversity of our dataset. To avoid duplicates between the specialized pre-training and fine-tuning datasets for equivalence prediction, we adopt a similar approach as we did for alias prediction. Specifically, we randomly selected half of the equivalent sets generated by EqMiner to construct the specialized pre-training dataset, while holding out the other half for the fine-tuning dataset. In total, we collected 66,403 equivalent and 66,538 inequivalent code pairs in the specialized pre-training dataset.

Fine-tuning Dataset. For the remaining equivalent sets (1,840 in total), we check if code fragments in each set are indeed semantically equivalent, considering that code fragments from different sets are certain to be semantically inequivalent. We enlisted the help of 130 undergraduate students from the computer science department at our university for this task. Since code segments produced by EqMinder are always self-contained, meaning, they include all the information needed to determine the label, raters only need to inspect the code fragments as presented. Much like in alias prediction, it's important to emphasize that there is no mismatch between the construction of ground truth and the presentation of prediction samples. This is because models deal with the very same code fragments that EqMiner/human raters do when confirming their labels. Each code pair was inspected by 2 students, and the labeling process took just over two months to complete. On average, the inter-rater reliability, as measured by the kappa score, is 0.72. In the end, we obtained 10,655 equivalent code pairs and 10,861 inequivalent code pairs. Inequivalent code pairs are collected in the same way as they are in the specialized pre-training stage. The last two rows in Table 2 give the length (*resp.*, size) of token sequences (*resp.*, graphs) for equivalence prediction. As with alias prediction, specialized pre-training and fine-tuning datasets contain less data than generalized pre-training dataset due to the exclusion of small functions.

6 EXPERIMENTS

In this section, we provide an overview of our experimental setup and describe the program representations used by each code model in our evaluation. We then present the results of all models in both alias and equivalence prediction tasks. Finally, we conduct an in-depth analysis of the models' results in these two prediction tasks and discuss our findings.

6.1 Experimental Setup

Vocabulary. Since CodeSem does not have a particularly large vocabulary size or frequent occurrence of rare words, we do not adopt BPE [Sennrich et al. 2016] to address the out-of-vocabulary issue. Instead, we take a simpler approach by constructing the vocabulary at the word-level while excluding rare words. Specifically, we consider tokens that appeared at least ten times among all data points in CodeSem. For tokens that appear less than ten times, we replace them with $[OOV]$ in both the training and test datasets. The sequence model has a vocabulary size of 190,029, while the graph model has a vocabulary size of 277,302.

Hyperparameters. To ensure the optimal performance of each code model, we tune their hyperparameters using Bayesian Optimization [Martinez-Cantin 2014], a common method for hyperparameter tuning. We normalize all evaluated models *w.r.t.* the number of model parameters, which is

Table 3. Important hyperparameters for each code model.

Models	Patience	Batch size	#Attention heads	#Layers	Hidden size	Learning rate	Optimizer
CuBERT	20	32	4	4	256	5e-5	Adam
CodeBERT	20	32	4	4	256	5e-5	Adam
GGNN	20	72	/	5	192	0.001	Adam
Graph Sandwiches	20	72	/	5	192	0.001	Adam

Table 4. The train-test split of CodeSem on which CuBERT achieves the median accuracy.

Generalized Pre-training	Training		Validation		
	115,528		52,907		
Alias Prediction	Specialized Pre-training		Fine-tuning		
	Training	Validation	Training	Validation	Test
	17,420	5,638	21,163	6,991	7,038
Equivalence Prediction	Specialized Pre-training		Fine-tuning		
	Training	Validation	Training	Validation	Test
	83,054	31,826	14,012	3,201	4,303

around 3M per model. Table 3 shows the value for some of the most important hyperparameters for each code model.

Validity of Models. We validate all code models on their original tasks. The details are provided in Section 2 of the supplemental material.

Prediction Setting. Given the high degree of similarity that can exist between code from the same project, we evaluate the models in a cross-project prediction setting. Specifically, we perform 14 rounds of cross-validation, with each project left to the test set in turn, and the models trained on the remaining projects. When a project is reserved for the test set, we first remove all its code from the generalized and specialized pre-training dataset, then use only its data points on the fine-tuning dataset to construct the test set. This approach ensures that models do not observe any code from the project reserved for the test set during the training process. To report our evaluation results, we choose the train-test split where a model achieves the median accuracy among all splits (hereinafter referred to as the median model). Table 4 presents the details about the train-test split for the median model of CuBERT. Those for the other code models are left to the supplemental material (Section 3).

Metric. First, we adopt accuracy, a standard metric, to evaluate the performance of models. In order to provide a holistic view of models' performance, we also report their accuracy for each prediction class. In alias prediction, we report the accuracy of (median) models for must-, may-, and must-not-alias, respectively (represented by the three numbers in parentheses in each cell in Table 5). In equivalence prediction, we report the accuracy of (median) models for equivalent and inequivalent code pairs (represented by the two numbers in parentheses in each cell in Table 6). Additionally, we take confidence intervals into account and use a common confidence level of 95% to compute them [Cao et al. 2022; Pantiuchina et al. 2021; Sun et al. 2022]. This provides a more accurate representation of the performance of each (median) model.

Hardware. All experiments are carried out on a Ubuntu 18.04.6 LTS server, with 10 Intel Xeon Gold 6248 CPUs @ 2.50GHz, 2 NVIDIA Tesla A100 GPUs (80GB GPU memory).

6.2 Program Representations

We investigate the impact of different code representations on graph models only since CuBERT and CodeBERT simply take code as token sequences.

6.2.1 AST with Additional Edges. It is the original code representation designed for GGNN. To represent both the syntactic and semantic structure of code, Allamanis et al. [2018] propose a program graph that incorporates extra edges into ASTs (*e.g.*, connecting each terminal node to its successor, connecting a variable to others that the variable is data or control dependent on).

6.2.2 Leaves-Only Graph. Leaves-only graph, originally proposed by Hellendoorn et al. [2019] for Graph Sandwiches, consists of exclusively the terminal nodes from ASTs. Specifically, after removing standard AST edges (*i.e.*, parent-child edge), it moves down other edges (inherited from Allamanis et al. [2018] work) from non-terminal nodes – which typically represent a span of multiple tokens – to terminal nodes that represent the starting token of that span. For example, an edge that is used to connect AST nodes of two *ForStatement* will be moved down to connect the corresponding *for* tokens. This representation has two advantages: (1) it is substantially more compressed than AST-based graphs, and often uses fewer nodes while retaining most edges; additionally (2) it accommodates sequence models better because all edges are directly connected among tokens.

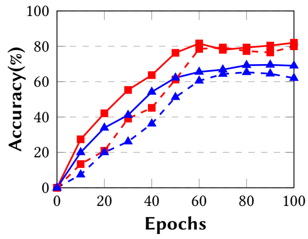
6.2.3 Control Flow Graph. We adopt the Control Flow Graph (CFG)-based representation presented by Wang et al. [2020]. Specifically, given a standard CFG, we first split a graph node representing a basic block into multiple nodes, each of which is a single statement. Subsequently, we add additional edges to connect every statement with its immediate successor within the same basic block. For edges in the original control flow graphs, we change their start (*resp.*, end) nodes from a basic block to its last (*resp.*, first) statement after the split. Then, we design two statement representation methods: (1) we replace each statement node with a sequence of nodes, among which each node represents a token in the statement. Specifically, every token node is connected to its immediate successor, and the first token node will become the new start or end node of edges that were connecting the statement nodes before (hereinafter, this whole program representation scheme is abbreviated to CFG+token seq); (2) we replace each statement node with its abstract syntax tree, the method proposed by Si et al. [2018]. The root nodes of each AST are used to connect statements in the CFG. In the remainder of this paper, we refer to this program representation as CFG+AST.

6.2.4 Program Dependence Graph. The last program representation is based on program dependence graph (PDG) which makes explicit both the data and control dependence for each operation in a program [Ferrante et al. 1987]. Like CFG-based graph representation, we represent every node in a PDG, which is a single statement, by its token sequence (hereinafter denoted by PDG+toke seq) or abstract syntax tree (hereinafter denoted by PDG+AST). Section 4 in the supplemental material gives an example of this graph representation.

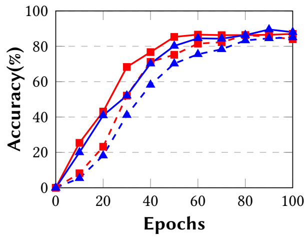
Inter-Procedure Code Representations. For sequence models, we concatenate the token sequence of each function in the order it is called. For example, when *Fun*$_1$ calls *Fun*$_2$ which then calls *Fun*$_3$, we concatenate the token sequence of *Fun*$_1$, *Fun*$_2$, and *Fun*$_3$ in turn. For graph models, we first construct the graph for each function as described above, and then connect the node representing the callsite in the caller to the entry (or root) node in the CFG/PDG (or AST) of the callee.

6.3 Results of Alias Prediction

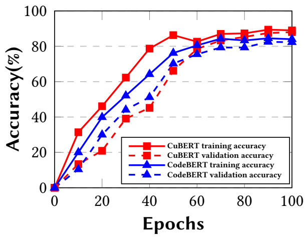
During training, we adopt a common approach — early stopping [Prechelt 1998] — to prevent models from overfitting. As displayed in Table 3, the patience is set to be 20 (*i.e.*, we wait 20 epochs before early stop if no progress on the validation set). Figure 3 plots the training and validation



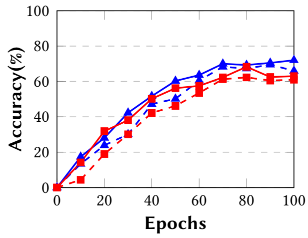
(a) the generalized pre-training stage of sequence models



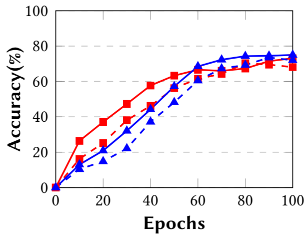
(b) the specialized pre-training stage of sequence models



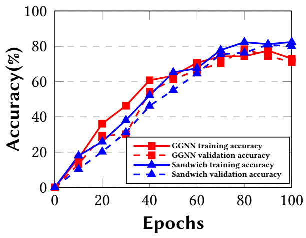
(c) the fine-tuning stage of sequence models



(d) the generalized pre-training stage of graph models



(e) the specialized pre-training stage of graph models



(f) the fine-tuning stage of graph models

Fig. 3. The trend of models' accuracy on training and validation sets *w.r.t.* the number of epochs at each training stage in alias prediction. For each code model, the curves correspond to the median model that achieves the highest test accuracy among those adopting different def-use configuration, and consuming different program representations. These median models (with the highest accuracies) are precisely the ones that learned from the train-test split presented in Table 4 and Section 3 of the supplemental material.

accuracy of median models over training epochs for all code models, in particular, at each training stage, models are saved at the point when their accuracies on the validation set reach the apex. We compare the performance of models trained with our approach against those trained with pre-training (using generalized pre-training dataset) and fine-tuning (using fine-tuning dataset). Table 5 presents the results from which we summarize important findings below:

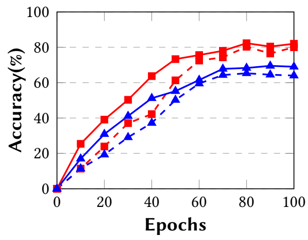
- With a confidence level of 95%, we compute the confidence interval for all models to fall in the range of (-1.0%,+1.0%), strongly indicating our results provide an accurate representation of models' performance in alias prediction. In terms of accuracy, sequence models performed notably better than graph models, with CuBERT and CodeBERT achieving the highest accuracy of 89% and 84% while GGNN and Graph Sandwiches achieving the highest accuracy of 78% and 81%. The choice of program representation is crucial to the performance of graph models. The accuracy of the same model can vary drastically with different program representations. For example, when GGNN switches from PDG+token seq to CFG+AST while keeping the other factors unchanged (trained with our approach and adopting the configuration of DEF+USE for variable embedding), it experienced a 29% accuracy drop.
- In most cases, embedding variables with uses leads to a higher overall model accuracy than without. Occasionally, incorporating uses can also hurt model accuracy. This may be caused by complicated usage patterns of variables that negatively affect the precision of their embeddings. After all, the way that variables are used does not fundamentally determine whether they are alias or not. Regarding the comparison between our training approach and pre-training and fine-tuning, models trained with our approach are almost always more accurate than those undergoing pre-train and fine-tuning. In fact, with the same def-use configuration for variable

Table 5. Results of alias prediction. All numbers in the table are percentages of accuracy and those in parentheses represent the accuracy of median models for must-alias, may-alias, and must-not-alias, respectively. Numbers in bold are the highest accuracy of each model.

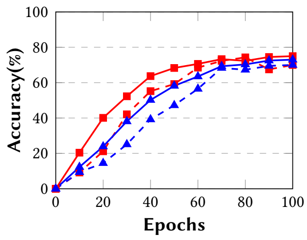
Models	Program Representation	Generalized, Specialized Pre-training + Fine-tuning		Pre-training + Fine-tuning	
		DEF+USE	DEF	DEF+USE	DEF
GGNN	AST+extra edges	63 (69, 95, 25)	62 (70, 88, 29)	66 (80, 92, 27)	65 (72, 79, 44)
	CFG+token seq	74 (71, 68, 85)	60 (10, 87, 82)	72 (63, 69, 84)	57 (27, 53, 90)
	CFG+AST	49 (62, 79, 41)	44 (10, 62, 48)	47 (48, 83, 40)	33 (44, 36, 31)
	PDG+token seq	**78 (74, 70, 90)**	65 (62, 49, 89)	74 (66, 70, 87)	63 (57, 60, 74)
	PDG+AST	77 (60, 84, 93)	66 (59, 55, 89)	75 (70, 78, 80)	59 (59, 46, 71)
Graph Sandwiches	Leaves-only graph	55 (56, 69, 39)	55 (57, 73, 35)	54 (60, 66, 35)	52 (53, 80, 22)
	CFG+token seq	69 (47, 78, 82)	66 (47, 70, 80)	69 (46, 73, 87)	65 (45, 71, 78)
	CFG+AST	48 (51, 67, 44)	46 (57, 55, 42)	49 (47, 78, 44)	46 (58, 51, 42)
	PDG+token seq	71 (59, 68, 88)	74 (64, 74, 88)	67 (39, 84, 84)	71 (62, 67, 84)
	PDG+AST	**81 (75, 78, 89)**	72 (73, 86, 59)	79 (74, 91, 74)	77 (65, 85, 83)
CuBERT	Token Seq	**89 (88, 88, 89)**	87 (86, 88, 89)	87 (89, 84, 89)	86 (85, 85, 89)
CodeBERT	Token Seq	83 (73, 85, 89)	**84 (75, 87, 89)**	81 (71, 84, 89)	77 (66, 87, 89)

embedding and program representation, 18 out of 24 models achieve higher accuracies thanks to our training workflow. While the improvement may not be substantial, its consistency in producing models with higher accuracies is a testament to its effectiveness.

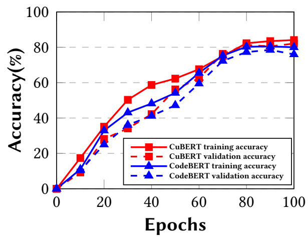
- CuBERT, the most accurate model, also exhibits the most balanced accuracy across all three labels. CodeBERT also achieves balanced accuracies to a certain extent. Graph models display a wide range of results: on the one hand, GGNN and Graph Sandwiches achieve rather balanced accuracies while achieving the highest accuracy; on the other hand, their accuracy heavily skew towards a certain category, causing significant disparity for all three labels. We suspect that program representation is an important factor to how balanced models' accuracies will be.
- Due to hardware constraints, code samples having more than 2,000 tokens, which amounts to less than 5% of the alias data in CodeSem, do not fit into the attention windows of CuBERT or CodeBERT [Kitaev et al. 2020]. For these code samples, we adopt the approach of prior work [Puri et al. 2021] by truncating the code samples to the first 2,000 tokens. If the first 2,000 tokens do not contain any live definition for either of the two variables, the code sample is discarded. As a result of this truncation, almost all of the retained code samples only have partial information about the predicted pointers (*i.e.*, lacking either definition or use for at least one variable). Despite this, both CuBERT and CodeBERT achieve approximately the same accuracy for the truncated code as they do for the complete code. This finding suggests that even partial information about the pointers is sufficient to express their aliasing relations. Additionally, sequence models are highly capable of capitalizing such partial information to solve the alias prediction task.
- The may-alias analysis [Zheng and Rugina 2008] that we rely on to construct CodeSem achieves 12.5% accuracy for must-not-alias. We note that classical may-alias analysis do not differentiate must-alias from may-alias, therefore they can not deal with may- and must-alias pairs in CodeSem. If may-alias and must-alias are combined into a single prediction class, the analysis would achieve 100% in this category thanks to its soundness guarantees. The must-alias analysis [Kastrinis et al. 2018] achieves 30.7% accuracy on must-alias pairs, and it can not handle the other two categories in the alias prediction task. Similarly, when must-not-alias are combined with may-alias as a single prediction class, the analysis would achieve 100% accuracy. To sum up, in terms of accuracy,



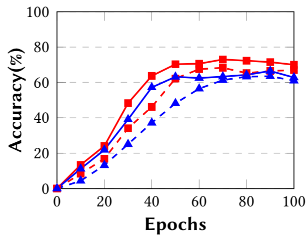
(a) the generalized pre-training stage of sequence models



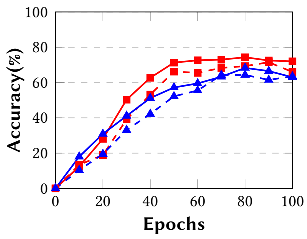
(b) the specialized pre-training stage of sequence models



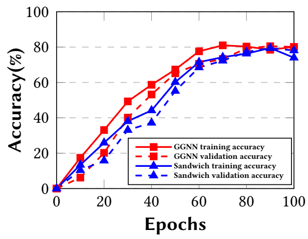
(c) the fine-tuning stage of sequence models



(d) the generalized pre-training stage of graph models



(e) the specialized pre-training stage of graph models



(f) the fine-tuning stage of graph models

Fig. 4. The trend of models' accuracy on training and validation sets *w.r.t.* the number of epochs at each training stage in equivalence prediction. For each code model, the curves correspond to the median model that achieves the highest test accuracy among those consuming different program representations.

all deep learning models are far superior to the two alias analyses, however, alias analyses provide theoretical guarantees on the correctness of certain alias pairs that deep learning models can not.

6.4 Results of Equivalence Prediction

Figure 4 plots the training and validation accuracy of median models over training epochs for all code models. Similar to the approach taken in alias prediction, we save models at the point when their accuracies on the validation set reach the apex at each training stage to prevent overfitting. We present the results for all median models in Table 6. The key findings are summarized below.

- Similar to the results in the alias prediction task, all models achieved a confidence interval between -1.4% to +1.4% at a 95% confidence level. Again, this strongly indicates our results are an accurate reflection of models' performance in equivalence prediction. All models achieved comparable high accuracy in this task. Specifically, CuBERT, which achieves the highest accuracy at 84%. CodeBERT, GGNN, and Graph Sandwiches are close behind achieving the accuracy of 81%, 81%, and 79%, respectively. In general, all models achieve balanced accuracy.
- Compared to alias prediction, the role of program representation is less significant in the performance of graph models in equivalence prediction. Nevertheless, both GGNN and Graph Sandwiches show a swing in accuracy of 5% to 10% depending on the program representation used. Interestingly, both models achieve their highest accuracy with PDG+AST. Together with the results of alias prediction, this demonstrates the power of PDG as a principled program representation for learning code models for foundational program analysis tasks.
- In almost all cases, models undergoing our training workflow are more accurate than those trained with pre-training and fine-tuning. These results and those of the alias prediction task confirm the role of the specialized pre-training stage in training models towards solving foundational program analysis tasks.

Table 6. Results of equivalence prediction. All numbers in the table are percentages of accuracy and those in parentheses represent the accuracy of models for equivalent and inequivalent programs, respectively. Numbers in bold denote the highest accuracy achieved by each model.

Models	Program Representation	Generalized, Specialized Pre-training + Fine-tuning	Pre-training + Fine-tuning
GGNN	AST+extra edges	79 (83, 75)	76 (79, 74)
	CFG+token seq	76 (81, 72)	71 (66, 76)
	CFG+AST	79 (82, 77)	72 (67, 77)
	PDG+token seq	80 (83, 77)	78 (79, 77)
	PDG+AST	**81 (84, 78)**	80 (87, 73)
Graph Sandwiches	Leaves-only graph	79 (83, 74)	77 (78, 77)
	CFG+token seq	74 (75, 74)	73 (67, 78)
	CFG+AST	69 (64, 74)	71 (69, 74)
	PDG+token seq	78 (80, 76)	77 (78, 76)
	PDG+AST	**79 (86, 72)**	77 (76, 78)
CuBERT	Token Seq	**84 (86, 82)**	83 (86, 80)
CodeBERT	Token Seq	**81 (79, 82)**	80 (80, 80)

- To ensure that all code samples can fit into the attention windows of CuBERT or CodeBERT, we truncate larger programs, which make up around 4% of the equivalence data in our dataset, to their first 2,000 tokens. When tested on these truncated programs, CuBERT and CodeBERT achieve 72% and 67% accuracy, respectively. Both are significantly lower than their accuracies on the complete code. This is an expected result since semantic equivalence, a global code property, requires models to reason about code in their entirety. This highlights a potential limitation of BERT-based models in predicting global properties of large programs due to hardware constraints.

6.5 A Deep, Comprehensive Analysis of the Results of All Models

We conduct an in-depth analysis of the results of evaluated models to gain an insight into their ability to solve the alias and equivalence prediction task. In particular, we focus on alias pairs (*i.e.*, must-alias and may-alias pairs) and equivalent code pairs, which are also the point of study in alias analysis and equivalence checking, in this qualitative evaluation. For GGNN and Graph Sandwiches, we pick the model that achieved the best results (among different program representations and def-use configuration) in our experiments previously for this qualitative analysis.

6.5.1 *Analyzing Model Results in Alias Prediction.*

Investigating the impact of NL-level information in code. To investigate the impact of natural language-level (NL-level) information in code, such as comments and variable names, on model performance, we devise the following experiment. First, our models do not consider comments in the code, thus they do not have any impact on model performance. Second, to anonymize variable identifiers, we assign variables with randomly generated names (consisting of letters and digits) in each code example. After these two steps, we retrain models on these code examples without any NL-level information. Results show that for CuBERT or CodeBERT, the drop in model accuracy is around 2% on average across ten runs. For GGNN and Graph Sandwiches, the decrease in model accuracy is around 4% on average across ten runs. Based on this observation, we conclude that the NL-level information in code has a minimal impact on model performance in alias prediction. The reason that NL-level information barely affects model accuracies is that NL-level information is unrelated to the essence of the prediction task, specifically, the aliasing relation between pointer variables is independent of their names. Also, our training data does not exhibit spurious correlations [Ribeiro

Table 7. Model accuracy *w.r.t.* the distance of aliasing relations. The first/second number in every cell is the accuracy of a code model for must-alias/may-alias pairs.

Models \ Distance	0	$1 \leq D < 4$	$4 \leq D < 7$	$7 \leq D < 10$	$10 \leq D$
CuBERT	91%, 90%	91%, 93%	84%, 88%	84%, 85%	80%, 81%
CodeBERT	77%, 88%	74%, 92%	80%, 91%	72%, 86%	70%, 82%
GGNN	80%, 73%	75%, 69%	72%, 65%	72%, 66%	58%, 59%
Graph Sandwiches	80%, 80%	72%, 83%	73%, 82%	72%, 77%	63%, 65%

et al. 2016] pertaining to NL-level information (as verified by human raters in the labeling process), thus, models can not pick up NL artifacts in code examples to link with their labels.

Exploring model performance for alias pairs w.r.t. locality. In a systematic evaluation, we examine how accuracies of code models vary with the locality of alias pairs. Specifically, we investigate whether code models are only capable of handling local aliases or whether they can also handle aliases that span several statements. To answer this question, we first define a metric called distance (D) that measures the number of statements between the definitions that make two variables alias. Formally, let p be a program path along which $[d_1, d_2, \dots d_n]$ are definitions (executed in order) that make two variables alias. The distance metric counts the number of statements between the execution of the first definition d_1 and the last definition d_n. If two variables become alias due to one definition such as *int* $^*p,^*q; p = q;$, then their distance is 0. Next, we classify must- and may-alias pairs in the test set of each code model into five categories based on the distance metric: $D = 0$, $1 \leq D < 4$, $4 \leq D < 7$, $7 \leq D < 10$, and $10 \leq D$. We record the accuracy of each code model in each category and report the results in Table 7. Although in most cases all models become less accurate as the distance of alias pairs increases, they still achieve acceptable accuracy in general. For example, no model has a decrease of more than 8% accuracy when $D < 10$ for either must- or may-alias pairs, demonstrating a level of sustainability across the locality spectrum.

Last but not least, we also observe that all code models coped with different types of assignments that make variables aliases, such as *address-of* ($p = \&q$), *load* ($*p = q$), *store* ($p = *q$), indicating that all models have learned a comprehensive set of semantic features for predicting aliasing relations. Overall, our analysis shows that all models have adequately solved the alias prediction task.

6.5.2 Analyzing Model Results in Equivalence Prediction. We skip the analysis of the impact of NL-level information on model performance. Because (1) like in alias prediction, models do not take into account comments in the code either during training or test; and (2) code examples in CodeSem, which are generated by EqMiner, are already anonymized.

Confirming the Validity of CodeSem. First and foremost, we validate CodeSem for equivalence prediction. As we explained at the very beginning of this paper, equivalence prediction task requires models to predict whether or not two programs are semantically equivalent. This means that data points in CodeSem should not be simple, syntax-level code clones that can be easily detected by syntactic similarity. To confirm this, we run a well-established, highly impactful clone detection tool, Deckard [Jiang et al. 2007], on the test set of each code model. Our results show that in the best case (on CuBERT's test set) Deckard has 61% (*resp.*, 63%) accuracy on pairs of equivalent (*resp.*, inequivalent) programs, which is marginally higher than the chance-level accuracy. We have experimented with a wide range of Deckard's hyperparameters and the results above are the optimal in balancing the precision (aiming to avoid reporting false clones) and recall (aiming to capture all real clones). The experiments confirm the validity of CodeSem for equivalence prediction.

Since semantic clone detection [Roy et al. 2009] shares some similarities with the equivalence prediction task, we choose Tailor [Liu et al. 2023], the most recent tool for semantic clone detection, to evaluate on CodeSem. Tailor exhibits outstanding performance on BigCloneBench [Svajlenko et al. 2014] and OJClone [Mou et al. 2016], achieving close to 100% accuracy on both benchmarks. Like our models, Tailor is deployed to predict in a cross-project setting: we evaluate Tailor on each train-test split of CodeSem and report the median accuracy that Tailor achieves. We note that we have tuned Tailor's hyperparameters using Bayesian Optimization to ensure that it achieves its optimal performance on CodeSem. We find that Tailor attains a median accuracy of 68% (47% and 88% on pairs of equivalent and inequivalent programs respectively), which is significantly lower than its accuracy on BigCloneBench and OJClone. The degradation of Tailor's performance highlights a substantial disparity between equivalence prediction and semantic clone detection. In particular, Tailor's struggle against equivalent programs (which is the main subject of equivalence prediction) strongly indicates the advantages of CodeSem over existing benchmarks. Since CodeSem'data is extracted from well-established, real-world programs, it poses a bigger challenge to code models compared to the simpler programs often found on coding platforms.

Refuting a Template-Matching Approach. Next, we investigate whether code models have relied solely on a simple, template-matching approach for recognizing equivalent programs. That is, given a pair of code examples (α, β) in the test set, do models merely attempt to find another pair (x, y) that they memorized from the training set such that α and β are syntactically similar to x and y (or y and x) respectively. To answer this question, we aim to quantify the number of code pairs in the test set that models would have predicted correctly if they had successfully memorized all code pairs from the training set. Again, we use Deckard, set up with the optimal hyperparameters, to conduct this experiment. Results show that in the best case (*i.e.*, on CodeBERT's test set,) less than 11% of equivalent code pairs in the test set have a syntactically similar counterpart in the training set. This suggests that this simple, template-matching approach is insufficient for recognizing equivalent programs in CodeSem.

Comparing Models with Equivalence Checkers. Moving on, we now seek to understand what models have learned in equivalence prediction task. For this purpose, we evaluate state-of-the-art equivalence checkers on CodeSem as baselines for comparison with code models. This experiment can reveal challenges posed by CodeSem that are beyond state-of-the-art equivalence checkers. Thus, by analyzing how well models cope with those challenges, we can gain a deep understanding of their capability in the equivalence prediction task. We pick *trace alignment* [Churchill et al. 2019], *ARDIFF* [Badihi et al. 2020], and *Rêve* [Felsing et al. 2014], which are prominent equivalence checking tools in the literature, to analyze all code pairs in the fine-tuning dataset. As shown in Table 8, none of the tools perform adequately on CodeSem. In fact, the most accurate tool, *trace alignment*, achieves an accuracy of below 25%. Therefore, we conclude that CodeSem presents significant challenges that current equivalence checkers are not equipped to handle. Next, we discuss the specific challenges that the three equivalence checkers face, and how code models have successfully addressed these challenges, respectively.

Trace alignment checks the equivalence of two programs based on the alignment of their concrete execution traces. It starts by generating test cases to execute the two programs and then aligns their execution traces using a linear function: $c_1 v_1 - c_2 v_2 = k$, where $c_1, c_2 \in \{1, 2, 4, 8, 16\}, k \in \mathbf{Z}$ are parameters, and v_1, v_2 are registers or stack-allocated locations in the two programs. Next, it

Table 8. Accuracy of state-of-the-art equivalence checkers on CodeSem. Values in parentheses represent the accuracy of each tool for equivalent, and inequivalent program pairs in the fine-tuning set of CodeSem respectively. We count the result of a tool to be incorrect if it timed out. Increasing the timeout parameter to around ten times its value does not help improve the performance of any tool.

	trace alignment	*ARDIFF*	*Rêve*
Accuracy	24.6% (16.9%, 27.4%)	12.5% (6.8%, 18.1%)	3.7% (3.1%, 4.2%)

```c
static enum cmd_retval cmd_list_keys_exec(...) {
  ...
  tmp = xmalloc(tmpsize);
  while (table != NULL) { ...
    while (bd != NULL) { ...
      while (tmpused + cplen + 1 >= tmpsize) {
        tmpsize *= 2;
        tmp=xrealloc(tmp, tmpsize);
        ...
        tmpsize += pow(diff, 2); ...
      } ...
}}}
```

```c
static enum cmd_retval list_keys(...) {
  ...
  buf = xmalloc(bufsize); ...
  for (; table != NULL;) { ...
    for (; bd != NULL;) { ...
      for (; bufsize - 1 <= tmpused + len;) {
        bufsize <<= 1;
        buf=xrealloc(buf, bufsize);
        ...
        bufsize += pow(diff, 2); ...
      } ...
}}}
```

Fig. 5. Two equivalent programs that *trace alignment* considers inequivalent. In contrast, all four models correctly predict them to be equivalent. Code omitted by $\cdots$ is not related to the weakness of the tool.

constructs an automaton using the aligned traces to simulate the behavior of the combination of the two programs. Finally, the method determines the equivalence between the two programs by verifying the satisfiability of the synthesized constraints from the automaton. Among all shortcomings of *trace alignment* (*e.g.*, simplistic alignment predicate, insufficient coverage of the generated test cases), the major weakness that accounts for most of its wrong results is the simplistic alignment predicate. Specifically, when reasoning about memory allocations in the heap or using non-linear functions is required to align the execution traces of the two programs, *trace alignment* would fail. Figure 5 shows an example of equivalent programs in CodeSem that *trace alignment* considers inequivalent. To establish the alignment of the execution traces of the two programs, *trace alignment* must consider variable *tmp* in the left hand side of Figure 5 and *buf* in the right hand side, which are both allocated on the heap. In addition, *trace alignment* also needs to deal with non-linear function *pow*(). Since *trace alignment* can only reason with local variables (allocated on the stack) and linear functions, the tool fails to recognize that the two programs are semantically equivalent. In fact, we find 4,591 pairs of equivalent programs in the fine-tuning set of CodeSem that *trace alignment* fails to recognize precisely due to its overly simplistic alignment predicates. In contrast, for those (among the 4,591 pairs of equivalent programs) that are included in the test sets of models, CuBERT, CodeBERT, GGNN, and Graph Sandwiches achieve 85.6%, 83.1%, 74.5%, and 77.9% accuracy respectively, indicating that all models are capable of recognizing the equivalence between programs even if their execution traces denote rather different semantics which *trace alignment* can not align.

Badihi et al. [2020] propose *ARDIFF* for enhancing the scalability of equivalence checking techniques based on symbolic execution. At the core of *ARDIFF* is a series of heuristics for identifying the parts of a program that can be pruned out to simplify the analysis. Despite a notable step forward, *ARDIFF* does not solve a fundamental issue with symbolic execution in handling large programs: the path constraints can be too complex (*e.g.*, size, nonlinearity) for the underlying SMT solver to solve. In total, we find 5,113 pairs of equivalent programs in the fine-tuning set of CodeSem on which *ARDIFF* timed out due to this limitation. An example is provided in Figure 4 in the supplemental material. In contrast, CuBERT, CodeBERT, GGNN, and Graph Sandwiches achieve 83.7%, 86.2%, 80.1%, and 75.9% accuracy on those (among the 5,113 pairs) that are in their respective test sets. These findings suggest that models are significantly more effective in handling larger programs with more complicated path constraints.

Rêve converts two programs into logical verification conditions (VC) and employs an SMT solver to determine their equivalence. Like *ARDIFF*, *Rêve* suffers from significant scalability issues, in addition, *Rêve* is limited to integer programs and does not support arrays. All of these are significant contributors to *Rêve*'s poor performance. We find 5,385 wrong results that *Rêve* produced on equivalent programs in the fine-tuning set of CodeSem are due to the aforementioned weaknesses. In contrast, CuBERT, CodeBERT, GGNN, and Graph Sandwiches achieved 85.1%, 81.3%, 77.4%, and

79.8% accuracy, respectively, on the subset of these 5,385 program pairs that are in their respective test sets. This demonstrates that the models are not restricted to certain types of programs, such as integer or floating-point, with or without arrays.

6.5.3 The Weaknesses of Models. We also thoroughly analyze the mispredictions made by all models to identify their weaknesses. In alias prediction, we find that models often struggle with pointers that have many definition and use points, possibly due to a lack of precision in identifying the exact point at which alias occurs. Figure 6 shows an example where *url* is the pointer that has many definition and use points, and all models fail to identify it as an alias of *colon_ptr* when the execution exits from the second *while* loop (from line 14 to 16).

```
1  static char *url_normalize_1(
2      const char *url, ...) {
3    ...
4    spanned = strspn(url, URL_SCHEME_CHARS);
5    if (!spanned || !isalpha(url[0]) ||
6        spanned + 3 > url_len ||
7        url[spanned] != ':' ||
8        url[spanned + 1] != '/' ||
9        url[spanned + 2] != '/')
10     ...
11   while (spanned--)
12     strbuf_addch(&norm, tolower(*url++));
13   ...
14   while (url < colon_ptr){
15     strbuf_addch(&norm, tolower(*url++));
16     url_len--;} ... }
```

Fig. 6. Aliases that all models fail to recognize. *url* (highlighted in shadow box) and *colon_ptr* (underlined) are aliases when exiting from the *while* loop (from line 14-16) in which case *url* equals *colon_ptr*.

For equivalence prediction, we discover a specialized class of equivalent programs in CodeSem that pose significant challenges to all models, as their equivalence relies on important assumptions about the structure of input data. Consider the programs in Figure 7, which are equivalent only if the input string (represented by the parameter *char* line*) conforms with a specific format that involves two-level delimiters: semicolons as the first and commas as the second (*e.g.*, "$a, b; c, d; e, f;$"). If the input string fails the format check (*i.e.*, *regex_match*() function), both programs return *NULL*, otherwise, they extract the j^{th} sub-element (with commas as the separator) within the i^{th} element of the string (with semicolons as the separator). The program on the left takes a natural approach of first splitting the string with semicolons to obtain the i^{th} element, and then splitting the i^{th} element with commas to obtain the j^{th} sub-element. The program on the right uses the two delimiters in a reversed order: commas first and then semicolons. As a necessary processing step, the original index (i, j) is adjusted in the following manner: when j is 1, the index (i, j) becomes $(i, j + 1)$ (*i.e.*, $(i, 2)$); when j is 2, the index becomes $(i + 1, 1)$ (line 8 and 9). After the index adjustment, the (new) j^{th} sub-element within the (new) i^{th} element, obtained by splitting the string with commas and then semicolons refers to the same character as the output of the program on the left. However, there is an exception to this rule when the character to be found is located at the very beginning of the input string, in which case the input string is split exactly once (with commas), and the first element is directly returned (line 5 to 7). Overall, the two

```
1  #include <regex>
2  const char* getfield(char* line, int i, int j) {
3    char* tok; regex reg("(.,.;)+\\n");
4    if (!regex_match(line, reg))
5        return NULL;
6    for (tok = strtok(line, ";");
7        tok && *tok;
8        tok = strtok(NULL, ";\n")) {
9      if (--i) continue;
10     for (tok = strtok(tok, ",");
11         tok && *tok;
12         tok = strtok(NULL, ",\n")) {
13       if (!--j)
14         return tok;
15   }}
16   return NULL;
17 }
```

```
1  #include <regex>
2  const char* getfield(char* line, int i, int j) {
3    char* tok; regex reg("(.,.;)+\\n");
4    if (!regex_match(line, reg)) return NULL;
5    if (i == j && j == 1) {
6      tok = strtok(line, ","); return tok;
7    }
8    j++;
9    if (j == 3) {i++; j = 1;}
10   for (tok = strtok(line, ","); tok && *tok;
11       tok = strtok(NULL, ",\n")) {
12     if (--i)  continue;
13     for (tok = strtok(tok, ";"); tok && *tok;
14         tok = strtok(NULL, ";\n")) {
15       if (!--j)  return tok;}}
16   return NULL;
17 }
```

Fig. 7. Two semantically equivalent programs that all four models incorrectly predict as inequivalent.

programs are indeed semantically equivalent, but recognizing their equivalence requires a complex reasoning procedure that models may not be capable of.

6.5.4 Outlook for Future Research. In light of the weaknesses of models, we suggest some directions for future research. Firstly, to mitigate the decrease in model accuracy caused by variables with a high number of definition and use points in alias prediction, more fine-grained embedding methods could be explored. For instance, the precision of variable embeddings could benefit from the interaction between the two variables aliased with each other. The enhanced precision of variable embeddings could ultimately help to improve the model accuracy in alias prediction task. For equivalence prediction, training models to formally reason about program behavior can be a pathway forward. For example, pre-training models towards objectives pertaining to pre- or post-conditions (*e.g.*, predict post-conditions given the pre-condition and the statement to be executed) could help models to capture the semantics of program statements at a deeper level. This, in turn, could improve their overall model accuracy in equivalence prediction task.

7 THREATS TO VALIDITY

Threats to External Validity. Our work is subject to certain external threats that may impact its validity. Firstly, due to limitations in the available tool-chain and infrastructure, our study focuses exclusively on programs written in C/C++. It is therefore reasonable to question the generalizability of our findings to other programming languages. However, given that C/C++ remain widely used languages and are often the subject of programming language research, we believe that our findings are still significant and relevant. As for the choice of models, first, models used in our evaluation are not the latest which are built upon large language models, however, our primary goal is to compare different neural architectures and traditional static analysis methods in alias and equivalence prediction. Also, we believe that our findings are likely to hold for the latest code models given their higher capability. Second, Graph Sandwiches has a particularly larger design space based on the type of sequence models' layers and how they interleave with GGNN layers. In our evaluation, we use RNN Sandwich, which wraps every message-passing layer in GGNN with an RNN, since it is one of the most accurate models according to the evaluation reported in [Hellendoorn et al. 2019].

Threats to Internal Validity. Human errors represent a threat to the internal validity of our study since the labeling process involves humans in the loop. Specifically, validating the results of LLVM and EqMiner is a rather tedious and error-prone task that could potentially affect the correctness of our findings. Despite these challenges, we have taken great care to minimize the impact of human errors by paying close attention to details. Given the practical limitations of our study, we believe that the potential risk of human errors is acceptable and should be tolerated.

8 RELATED WORK

In this section, we discuss three strands of related work: alias analysis, equivalence checking, and benchmarks for code models.

8.1 Alias Analysis

Thiessen and Lhoták [2017] introduce a context-sensitive analysis by combining the CFL-reachability and k-limited context strings, so that it obtains advantages of both methods. Phulia et al. [2020] design a sound must-not alias analysis to explore the optimization opportunity enabled by non-deterministic expression evaluation semantics. Wilson and Lam [1995] describe a flow-, context-sensitive pointer analysis algorithm for C programs that summarizes the behavior of procedures to increase its efficiency. Hardekopf and Lin [2009] present an inter-procedural, flow-sensitive pointer analysis that combines the idea of partial static single assignment and a heavy-analysis. Zhang et al. [2013] present two fast algorithms for Dyck-CFL-reachability on bidirected trees and graphs, and apply the algorithms to a context-insensitive alias analysis for Java [Yan et al. 2011]. Guo et al. [2019] propose a neural architecture to enhance the capability of Value Set Analysis to perform alias analysis at the binary level.

8.2 Equivalence Checking

Churchill et al. [2019] introduce a method of building a trace alignment for two given functions in the case of a set of user-provided test cases and constructing a product program for equivalence checking. Sharma et al. [2013] present a data-driven algorithm for checking the equivalence of loops written in x86 assembly, in particular, it solves an over-approximated relationship of input states to output states of the two loops. Dahiya and Bansal [2017] present a black-box equivalence checker to verify transformations produced by modern compilers. Gupta et al. [2018] propose an equivalence checking algorithm that allows the inference of the required invariants through the generation of counter-examples using SMT solvers. On the machine learning side, Kommrusch et al. [2023] aim to find semantically-preserving rewrite rules to convert one program to another. Regarding compiler optimization which is also related to equivalence checking, Trofin et al. [2021] propose a framework called MLGO1 to integrate machine learning techniques including Policy Gradient and Evolution Strategies into industrial compilers.

8.3 Benchmarks for Code Models

The work closest to ours is CodeXGLUE [Lu et al. 2021], which presents a benchmark of 10 tasks for model evaluation. CodeSem differs from CodeXGLUE in three ways. First, CodeSem is extracted from real-world programs while CodeXGLUE is a collection of programming solutions to algorithmic problems. Second, the prediction tasks CodeSem uses to evaluate models correspond to foundational program analysis tasks compared to those in CodeXGLUE (*e.g.*, clone detection, code translation). Third, CodeXGLUE features only sequential models whereas CodeSem also considers graph models. Another recent work, CodeNet [Puri et al. 2021], presents a large-scale dataset CodeNet. Like Lu et al. [2021], Puri et al. [2021] collect their data from online programming platforms while we assemble CodeSem from large-scale real-world programs. In addition, we propose two new tasks: alias prediction and equivalence prediction. Another related dataset is CodeSearchNet [Husain et al. 2019], which is used for semantic code search task. Wang and Christodorescu [2019] propose COSET, a benchmark for evaluating machine learning models in learning the semantics rather than syntax of code.

9 CONCLUSION

In this paper, we present CodeSem, a first-of-its-kind large-scale, real-world, and high-quality dataset designed to evaluate deep learning models in two foundational tasks in program analysis: alias prediction and equivalence prediction. We also propose a general, novel learning approach that makes it possible for models to leverage results of static analysis methods. With this learning approach, we train four influential code models — CuBERT, CodeBERT, GGNN, and Graph Sandwiches — towards the two prediction tasks. Our evaluation shows that, in general, all models display satisfactory performance in both tasks. However, we also identify the specific weaknesses of each model that should be addressed in future work. We release all the code and evaluation data for public access, and hope that the scale, diversity, and authenticity of CodeSem will offer unprecedented opportunities in this interdisciplinary area of research.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous reviewers for their comments and suggestions. This work was partially supported by the National Natural Science Foundation of China under Grant No. 62232001, No. 62032010, No. 62202220, and No. 62172200, the Collaborative Innovation Center of Novel Software Technology and Industrialization, Jiangsu Funding Program for Excellent Postdoctoral Talent, the Fundamental Research Funds for the Central Universities (No. 2023300180), and the Leading-edge Technology Program of Jiangsu Natural Science Foundation (No. BK20202001).

REFERENCES

Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. 2018. Learning to Represent Programs with Graphs. In *International Conference on Learning Representations (ICLR '18)*. https://doi.org/10.48550/arXiv.1711.00740

Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. 2019. code2vec: Learning Distributed Representations of Code. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–29. https://doi.org/10.1145/3290353

Rita Z. Altucher and William Landi. 1995. An Extended Form of Must Alias Analysis for Dynamic Allocation. In *Proceedings of the 22nd ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (San Francisco, California, USA) *(POPL '95)*. Association for Computing Machinery, New York, NY, USA, 74–84. https://doi.org/10.1145/199448.199466

Sahar Badihi, Faridah Akinotcho, Yi Li, and Julia Rubin. 2020. ARDiff: scaling program equivalence checking via iterative abstraction and refinement of common code. In *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 13–24. https://doi.org/10.1145/3368089.3409757

George Balatsouras, Kostas Ferles, George Kastrinis, and Yannis Smaragdakis. 2017. A Datalog Model of Must-Alias Analysis. In *Proceedings of the 6th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis* (Barcelona, Spain) *(SOAP 2017)*. Association for Computing Machinery, New York, NY, USA, 7–12. https://doi.org/10.1145/3088515.3088517

Yoshua Bengio, Jérôme Louradour, Ronan Collobert, and Jason Weston. 2009. Curriculum Learning. In *Proceedings of the 26th Annual International Conference on Machine Learning* (Montreal, Quebec, Canada) *(ICML '09)*. Association for Computing Machinery, New York, NY, USA, 41–48. https://doi.org/10.1145/1553374.1553380

Junming Cao, Bihuan Chen, Chao Sun, Longjie Hu, Shuaihong Wu, and Xin Peng. 2022. Understanding performance problems in deep learning systems. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 357–369. https://doi.org/10.1145/3540250.3549123

Zimin Chen, Steve James Kommrusch, Michele Tufano, Louis-Noël Pouchet, Denys Poshyvanyk, and Martin Monperrus. 2019. Sequencer: Sequence-to-sequence learning for end-to-end program repair. *IEEE Transactions on Software Engineering* (2019). https://doi.org/10.1109/TSE.2019.2940179

Kyunghyun Cho, Bart van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. 2014. Learning Phrase Representations using RNN Encoder–Decoder for Statistical Machine Translation. In *Proceedings of the 2014 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. https://doi.org/10.48550/arXiv.1406.1078

Berkeley Churchill, Oded Padon, Rahul Sharma, and Alex Aiken. 2019. Semantic program alignment for equivalence checking. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 1027–1040. https://doi.org/10.1145/3314221.3314596

Berkeley Churchill, Rahul Sharma, JF Bastien, and Alex Aiken. 2017. Sound Loop Superoptimization for Google Native Client. In *Proceedings of the Twenty-Second International Conference on Architectural Support for Programming Languages and Operating Systems* (Xi'an, China) *(ASPLOS '17)*. Association for Computing Machinery, New York, NY, USA, 313–326. https://doi.org/10.1145/3037697.3037754

Kevin Clark, Minh-Thang Luong, Quoc V. Le, and Christopher D. Manning. 2020. ELECTRA: Pre-training Text Encoders as Discriminators Rather Than Generators. In *International Conference on Learning Representations*. https://doi.org/10.48550/arXiv.2003.10555

Manjeet Dahiya and Sorav Bansal. 2017. Black-box equivalence checking across compiler optimizations. In *Asian Symposium on Programming Languages and Systems*. Springer, 127–147. https://doi.org/10.1007/978-3-319-71237-6_7

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *NAACL-HLT (1)*. 4171–4186. https://aclweb.org/anthology/papers/N/N19/N19-1423/

Elizabeth Dinella, Hanjun Dai, Ziyang Li, Mayur Naik, Le Song, and Ke Wang. 2019. Hoppity: Learning Graph Transformations to Detect and Fix Bugs in Programs. In *International Conference on Learning Representations (ICLR '19)*.

Dennis Felsing, Sarah Grebing, Vladimir Klebanov, Philipp Rümmer, and Mattias Ulbrich. 2014. Automating regression verification. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*. 349–360. https://doi.org/10.1145/2642937.2642987

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*. Association for Computational Linguistics, Online, 1536–1547. https://doi.org/10.48550/arXiv.2002.08155

Jeanne Ferrante, Karl J Ottenstein, and Joe D Warren. 1987. The program dependence graph and its use in optimization. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 9, 3 (1987), 319–349. https://doi.org/10.1145/24039.24041

Stephen Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanuel Geay. 2006. Effective Typestate Verification in the Presence of Aliasing. In *Proceedings of the 2006 International Symposium on Software Testing and Analysis* (Portland, Maine, USA) *(ISSTA '06)*. Association for Computing Machinery, New York, NY, USA, 133–144. https://doi.org/10.1145/1146238.1146254

Yaroslav Golubev, Viktor Poletansky, Nikita Povarov, and Timofey Bryksin. 2021. Multi-threshold token-based code clone detection. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 496–500. https://doi.org/10.1109/SANER50967.2021.00053

Wenbo Guo, Dongliang Mu, Xinyu Xing, Min Du, and Dawn Song. 2019. {DEEPVSA}: Facilitating Value-set Analysis with Deep Learning for Postmortem Program Analysis. In *28th USENIX Security Symposium (USENIX Security 19)*. 1787–1804.

Shubhani Gupta, Aseem Saxena, Anmol Mahajan, and Sorav Bansal. 2018. Effective use of SMT solvers for program equivalence checking through invariant-sketching and query-decomposition. In *International Conference on Theory and Applications of Satisfiability Testing*. Springer, 365–382. https://doi.org/10.1007/978-3-319-94144-8_22

Ben Hardekopf and Calvin Lin. 2009. Semi-Sparse Flow-Sensitive Pointer Analysis *(POPL '09)*. Association for Computing Machinery, New York, NY, USA, 226–238. https://doi.org/10.1145/1480881.1480911

Vincent J Hellendoorn, Charles Sutton, Rishabh Singh, Petros Maniatis, and David Bieber. 2019. Global relational models of source code. In *International conference on learning representations*.

Susan Horwitz. 1990. Identifying the Semantic and Textual Differences between Two Versions of a Program. In *Proceedings of the ACM SIGPLAN 1990 Conference on Programming Language Design and Implementation* (White Plains, New York, USA) *(PLDI '90)*. Association for Computing Machinery, New York, NY, USA, 234–245. https://doi.org/10.1145/93542.93574

Susan Horwitz. 1997. Precise flow-insensitive may-alias analysis is NP-hard. *ACM Transactions on Programming Languages and Systems (TOPLAS)* 19, 1 (1997), 1–6. https://doi.org/10.1145/239912.239913

Hamel Husain, Ho-Hsiang Wu, Tiferet Gazit, Miltiadis Allamanis, and Marc Brockschmidt. 2019. CodeSearchNet challenge: Evaluating the state of semantic code search. *arXiv preprint arXiv:1909.09436* (2019). https://doi.org/10.48550/arXiv.1909.09436

Lingxiao Jiang, Ghassan Misherghi, Zhendong Su, and Stephane Glondu. 2007. DECKARD: Scalable and Accurate Tree-Based Detection of Code Clones. In *29th International Conference on Software Engineering (ICSE'07)*. 96–105. https://doi.org/10.1109/ICSE.2007.30

Lingxiao Jiang and Zhendong Su. 2009. Automatic mining of functionally equivalent code fragments via random testing. In *Proceedings of the eighteenth international symposium on Software testing and analysis*. 81–92. https://doi.org/10.1145/1572272.1572283

T. Kamiya, S. Kusumoto, and K. Inoue. 2002. CCFinder: a multilinguistic token-based code clone detection system for large scale source code. *IEEE Transactions on Software Engineering* 28, 7 (2002), 654–670. https://doi.org/10.1109/TSE.2002.1019480

Aditya Kanade, Petros Maniatis, Gogul Balakrishnan, and Kensen Shi. 2020. Learning and Evaluating Contextual Embedding of Source Code. In *Proceedings of the 37th International Conference on Machine Learning (Proceedings of Machine Learning Research, Vol. 119)*. PMLR, 5110–5121.

George Kastrinis, George Balatsouras, Kostas Ferles, Nefeli Prokopaki-Kostopoulou, and Yannis Smaragdakis. 2018. An Efficient Data Structure for Must-Alias Analysis. In *Proceedings of the 27th International Conference on Compiler Construction* (Vienna, Austria) *(CC 2018)*. Association for Computing Machinery, New York, NY, USA, 48–58. https://doi.org/10.1145/3178372.3179519

Nikita Kitaev, Lukasz Kaiser, and Anselm Levskaya. 2020. Reformer: The Efficient Transformer. In *International Conference on Learning Representations*. https://openreview.net/forum?id=rkgNKkHtvB

Steve Kommrusch, Martin Monperrus, and Louis-Noël Pouchet. 2023. Self-Supervised Learning to Prove Equivalence Between Straight-Line Programs via Rewrite Rules. *IEEE Transactions on Software Engineering* 49, 7 (2023), 3771–3792. https://doi.org/10.1109/TSE.2023.3271065

Chris Lattner. 2008. LLVM and Clang: Next generation compiler technology. In *The BSD conference*, Vol. 5.

Chris Lattner and Vikram Adve. 2004. LLVM: A compilation framework for lifelong program analysis & transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004*. IEEE, 75–86. https://doi.org/10.1109/CGO.2004.1281665

Jiahao Liu, Jun Zeng, Xiang Wang, and Zhenkai Liang. 2023. Learning Graph-based Code Representations for Source-level Functional Similarity Detection. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 345–357. https://doi.org/10.1109/ICSE48619.2023.00040

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin Clement, Dawn Drain, Daxin Jiang, Duyu Tang, Ge Li, Lidong Zhou, Linjun Shou, Long Zhou, Michele Tufano, MING GONG, Ming Zhou, Nan Duan, Neel Sundaresan, Shao Kun Deng, Shengyu Fu, and Shujie LIU. 2021. CodeXGLUE: A Machine Learning Benchmark Dataset for Code Understanding and Generation. In *Thirty-fifth Conference on Neural Information Processing Systems Datasets and Benchmarks Track (Round 1).* https://doi.org/10.48550/arXiv.2102.04664

Ruben Martinez-Cantin. 2014. BayesOpt: A Bayesian Optimization Library for Nonlinear Optimization, Experimental Design and Bandits. *arXiv preprint arXiv:1405.7430* (2014). https://doi.org/10.48550/arXiv.1405.7430

Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. 2016. Convolutional neural networks over tree structures for programming language processing. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence.* 1287–1293.

George C Necula. 2000. Translation validation for an optimizing compiler. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation.* 83–94. https://doi.org/10.1145/349299.349314

Jevgenija Pantiuchina, Bin Lin, Fiorella Zampetti, Massimiliano Di Penta, Michele Lanza, and Gabriele Bavota. 2021. Why Do Developers Reject Refactorings in Open-Source Projects? *ACM Transactions on Software Engineering and Methodology (TOSEM)* 31, 2 (2021), 1–23. https://doi.org/10.1145/3487062

Ankush Phulia, Vaibhav Bhagee, and Sorav Bansal. 2020. OOElala: order-of-evaluation based alias analysis for compiler optimization. In *Proceedings of the 41st ACM SIGPLAN Conference on Programming Language Design and Implementation.* 839–853. https://doi.org/10.1145/3385412.3385962

Lutz Prechelt. 1998. Early stopping-but when? In *Neural Networks: Tricks of the trade.* Springer, 55–69. https://doi.org/10.1007/3-540-49430-8_3

Ruchir Puri, David S Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladimir Zolotov, Julian Dolby, Jie Chen, Mihir Choudhury, Lindsey Decker, et al. 2021. CodeNet: A large-scale AI for code dataset for learning a diversity of coding tasks. *arXiv preprint arXiv:2105.12655* (2021). https://doi.org/10.48550/arXiv.2105.12655

David A Ramos and Dawson R Engler. 2011. Practical, low-effort equivalence verification of real code. In *International Conference on Computer Aided Verification.* Springer, 669–685. https://doi.org/10.1007/978-3-642-22110-1_55

Marco Tulio Ribeiro, Sameer Singh, and Carlos Guestrin. 2016. "Why should i trust you?" Explaining the predictions of any classifier. In *Proceedings of the 22nd ACM SIGKDD international conference on knowledge discovery and data mining.* 1135–1144. https://doi.org/10.1145/2939672.2939778

Chanchal K Roy, James R Cordy, and Rainer Koschke. 2009. Comparison and evaluation of code clone detection techniques and tools: A qualitative approach. *Science of Computer Programming* 74, 7 (2009), 470–495. https://doi.org/10.1016/j.scico.2009.02.007

David E. Rumelhart and James L. McClelland. 1987. *Learning Internal Representations by Error Propagation.* 318–362.

Hitesh Sajnani, Vaibhav Saini, Jeffrey Svajlenko, Chanchal K. Roy, and Cristina V. Lopes. 2016. SourcererCC: Scaling Code Clone Detection to Big-Code. In *Proceedings of the 38th International Conference on Software Engineering* (Austin, Texas) *(ICSE '16).* Association for Computing Machinery, New York, NY, USA, 1157–1168. https://doi.org/10.1145/2884781.2884877

Eric Schkufza, Rahul Sharma, and Alex Aiken. 2013. Stochastic Superoptimization *(ASPLOS '13).* Association for Computing Machinery, New York, NY, USA, 305–316. https://doi.org/10.1145/2451116.2451150

Rico Sennrich, Barry Haddow, and Alexandra Birch. 2016. Neural Machine Translation of Rare Words with Subword Units. In *Proceedings of the 54th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers).* Association for Computational Linguistics, Berlin, Germany, 1715–1725. https://doi.org/10.18653/v1/P16-1162

Thomas Arthur Leck Sewell, Magnus O. Myreen, and Gerwin Klein. 2013. Translation Validation for a Verified OS Kernel. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Seattle, Washington, USA) *(PLDI '13).* Association for Computing Machinery, New York, NY, USA, 471–482. https://doi.org/10.1145/2491956.2462183

Rahul Sharma, Eric Schkufza, Berkeley Churchill, and Alex Aiken. 2013. Data-driven equivalence checking. In *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications.* 391–406. https://doi.org/10.1145/2509136.2509509

Qingkai Shi, Xiao Xiao, Rongxin Wu, Jinguo Zhou, Gang Fan, and Charles Zhang. 2018. Pinpoint: Fast and precise sparse value flow analysis for million lines of code. In *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation.* 693–706. https://doi.org/10.1145/3192366.3192418

Xujie Si, Hanjun Dai, Mukund Raghothaman, Mayur Naik, and Le Song. 2018. Learning Loop Invariants for Program Verification. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems* (Montréal, Canada) *(NIPS '18).* 7762–7773.

Manu Sridharan and Rastislav Bodík. 2006. Refinement-based context-sensitive points-to analysis for Java. *ACM SIGPLAN Notices* 41, 6 (2006), 387–400. https://doi.org/10.1145/1133255.1134027

Manu Sridharan, Satish Chandra, Julian Dolby, Stephen J. Fink, and Eran Yahav. 2013. *Alias Analysis for Object-Oriented Programs.* Springer Berlin Heidelberg, Berlin, Heidelberg, 196–232. https://doi.org/10.1007/978-3-642-36946-9_8

Eric Stoltz, Michael P Gerlek, and Michael Wolfe. 1994. Extended SSA with factored use-def chains to support optimization and parallelism. In *HICSS (2)*. 43–53. https://doi.org/10.1109/HICSS.1994.323280

Zhensu Sun, Li Li, Yan Liu, Xiaoning Du, and Li Li. 2022. On the importance of building high-quality training datasets for neural code search. In *Proceedings of the 44th International Conference on Software Engineering*. 1609–1620. https://doi.org/10.1145/3510003.3510160

Jeffrey Svajlenko, Judith F Islam, Iman Keivanloo, Chanchal K Roy, and Mohammad Mamun Mia. 2014. Towards a big data curated benchmark of inter-project code clones. In *2014 IEEE International Conference on Software Maintenance and Evolution*. IEEE, 476–480. https://doi.org/10.1109/ICSME.2014.77

Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. 2009. Equality Saturation: A New Approach to Optimization. In *Proceedings of the 36th Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages* (Savannah, GA, USA) *(POPL '09)*. Association for Computing Machinery, New York, NY, USA, 264–276. https://doi.org/10.1145/1480881.1480915

Rei Thiessen and Ondřej Lhoták. 2017. Context Transformations for Pointer Analysis. In *Proceedings of the 38th ACM SIGPLAN Conference on Programming Language Design and Implementation* (Barcelona, Spain) *(PLDI 2017)*. Association for Computing Machinery, New York, NY, USA, 263–277. https://doi.org/10.1145/3062341.3062359

Omer Tripp, Marco Pistoia, Patrick Cousot, Radhia Cousot, and Salvatore Guarnieri. 2013. Andromeda: Accurate and scalable security analysis of web applications. In *International Conference on Fundamental Approaches to Software Engineering*. Springer, 210–225. https://doi.org/10.1007/978-3-642-37057-1_15

Mircea Trofin, Yundi Qian, Eugene Brevdo, Zinan Lin, Krzysztof Choromanski, and David Li. 2021. Mlgo: a machine learning guided compiler optimizations framework. *arXiv preprint arXiv:2101.04808* (2021). https://doi.org/10.48550/arXiv.2101.04808

Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Advances in neural information processing systems*. 5998–6008.

Ke Wang and Mihai Christodorescu. 2019. COSET: A Benchmark for Evaluating Neural Program Embeddings. *arXiv preprint arXiv:1905.11445* (2019). https://doi.org/10.48550/arXiv.1905.11445

Ke Wang and Zhendong Su. 2020. Blended, Precise Semantic Program Embeddings. In *Proceedings of the 41st ACM SIGPLAN International Conference on Programming Language Design and Implementation (PLDI '20)*. https://doi.org/10.1145/3385412.3385999

Pengcheng Wang, Jeffrey Svajlenko, Yanzhao Wu, Yun Xu, and Chanchal K. Roy. 2018. CCAligner: A Token Based Large-Gap Clone Detector. In *Proceedings of the 40th International Conference on Software Engineering* (Gothenburg, Sweden) *(ICSE '18)*. Association for Computing Machinery, New York, NY, USA, 1066–1077. https://doi.org/10.1145/3180155.3180179

Yu Wang, Ke Wang, Fengjuan Gao, and Linzhang Wang. 2020. Learning Semantic Program Embeddings with Graph Interval Neural Network. *Proceedings of the ACM on Programming Languages* 4, OOPSLA, Article 137 (Nov. 2020), 27 pages. https://doi.org/10.1145/3428205

Hu Weihua, Liu Bowen, Gomes Joseph, Zitnik Marinka, Liang Percy, Pande Vijay, and Leskovec Jure. 2020. Strategies for Pre-training Graph Neural Networks. In *International Conference on Learning Representations*. https://doi.org/10.48550/arXiv.1905.12265

Robert P. Wilson and Monica S. Lam. 1995. Efficient Context-Sensitive Pointer Analysis for C Programs. In *Proceedings of the ACM SIGPLAN 1995 Conference on Programming Language Design and Implementation* (La Jolla, California, USA) *(PLDI '95)*. Association for Computing Machinery, New York, NY, USA, 1–12. https://doi.org/10.1145/207110.207111

Dacong Yan, Guoqing Xu, and Atanas Rountev. 2011. Demand-driven context-sensitive alias analysis for Java. In *Proceedings of the 2011 International Symposium on Software Testing and Analysis*. 155–165. https://doi.org/10.1145/2001420.2001440

Suan Hsi Yong, Susan Horwitz, and Thomas Reps. 1999. Pointer analysis for programs with structures and casting. *ACM SIGPLAN Notices* 34, 5 (1999), 91–103. https://doi.org/10.1145/301631.301647

Yang Yuan and Yao Guo. 2012. Boreas: An Accurate and Scalable Token-Based Approach to Code Clone Detection. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering* (Essen, Germany) *(ASE 2012)*. Association for Computing Machinery, New York, NY, USA, 286–289. https://doi.org/10.1145/2351676.2351725

Michał Zalewski. 2016. American Fuzzy Lop-Whitepaper. *https://lcamtuf. coredump. cx/afl/technical_details. txt.* (2016).

Qirun Zhang, Michael R Lyu, Hao Yuan, and Zhendong Su. 2013. Fast algorithms for Dyck-CFL-reachability with applications to alias analysis. In *Proceedings of the 34th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 435–446. https://doi.org/10.1145/2491956.2462159

Xin Zheng and Radu Rugina. 2008. Demand-driven alias analysis for C. In *Proceedings of the 35th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 197–208. https://doi.org/10.1145/1328438.1328464

Received 20-OCT-2023; accepted 2024-02-24