OBSCURACODER: POWERING EFFICIENT CODE LM PRE-TRAINING VIA OBFUSCATION GROUNDING

Anonymous authors
Paper under double-blind review

ABSTRACT

Language models (LMs) have become a staple of the code-writing toolbox. Their pre-training recipe has, however, remained stagnant over recent years, barring the occasional changes in data sourcing and filtering strategies. In particular, research exploring modifications to Code-LMs' pre-training objectives, geared towards improving data efficiency and better disentangling between syntax and semantics, has been noticeably sparse, especially compared with corresponding efforts in natural language LMs. In this work, we examine grounding on obfuscated code as a means of helping Code-LMs look beyond the surface-form syntax and enhance their pre-training sample efficiency. To this end, we compile **ObscuraX**, a dataset of approximately 55M source and obfuscated code pairs in seven languages. Subsequently, we pre-train **ObscuraCoder** models, ranging in size from 255M to 2.8B parameters, on a 272B-token corpus that includes ObscuraX and demonstrate that our obfuscation-based pre-training recipe leads to consistent improvements in Code-LMs' abilities compared to both vanilla autoregressive pre-training as well as existing de-obfuscation (DOBF) objectives. ObscuraCoder demonstrates sizeable gains across multiple tests of syntactic and semantic code understanding, along with improved capabilities in multilingual code completion, multilingual code commit summarization, and multi-purpose library-oriented code generation.

1 INTRODUCTION

In recent years, language models for code generation (Code-LMs) have become a near-indispensable accessory in a developer's toolbox. Their enhancement of productivity has proliferated into most of the software development lifecycle, including automating unit test generation, code infilling, predicting build errors, and code refactoring, *inter alia*, propelling their broad adoption in production (Dunay et al., 2024; Frömmgen et al., 2024; Murali et al., 2024). Code-LMs owe their competence gains in these areas to an ever-increasing parameter count (Liu et al., 2024) coupled with ever-larger scale of their pre-training: code corpora is not only increasing in size but also in quality due to progressively better data sourcing and filtering (Yang et al., 2024). While improvements to the hardware (Prabhakar et al., 2024) and software (Shah et al., 2024) stack are likely to continue and in turn facilitate training of even larger models, LM training scaling-laws (Hoffmann et al., 2022) predict diminishing gains from the relatively slow growth of publicly available code corpora (Lozhkov et al., 2024). This warrants a departure from the dominant trend of autoregressively training larger Code-LMs on more code and towards alternative pre-training objectives that more efficiently exploit the inherent structure and regularity (Hindle et al., 2016) of source code.

Disentangling code syntax and semantics. All modern high-level programming source code inherently contains two information channels: syntactic and semantic (Knuth, 1984; Casalnuovo et al., 2020). While the former refers to conventional syntactic and naming choices developers make to facilitate interpretation and maintainability of the code (Casalnuovo et al., 2019), the latter pertains to the algorithmic intent behind the code. These two channels latently constrain each other and cannot be (fully) separated in form, only in understanding. To become a mainstay of software development, Code-LMs must excel in both syntactic and semantic code understanding.

The mastery of this dual-channel nature of programming code has so far, however, eluded Code-LMs. Existing studies of Code-LMs show that their grasp of syntax is much better and obtained much quicker during pre-training than their understanding of code semantics (Naik et al., 2022), with their representations being sensitive to semantic-preserving surface-form perturbations (Rabin et al., 2021).

The standard autoregressive language modelling latches on to common syntactic sub-spans, failing to capture the higher-level constructs in code (Ma et al., 2022), shown to be efficiently captured by specialized architectures (Shi et al., 2022). This is especially true for verbose languages such as Java, where separators account for a significant portion of all tokens (Rahman et al., 2019).

Consequently, disentangling Code-LMs' understanding of code syntax and semantics can unlock improvements on downstream tasks. Evidence for this has been demonstrated at inference-time: Code-LMs completion performance drastically improves when provided with fine-grained syntactic intent, described in natural language (Mueller et al., 2024; Sun et al., 2024; Li et al., 2023a; Jain et al., 2024; Wang et al., 2024a). A similar effect occurs when Code-LMs are prompted to generate code in multiple phases with the aid of code sketches that break down complex compositional syntactic structures (Guo et al., 2022b; Zan et al., 2022). Attempts at similarly untwining the two channels at pre-training time using contrastive learning (Wang et al., 2022) or translation (Chakraborty et al., 2022) between semantically equivalent code fragments have been hamstrung by the limited attainability and diversity of such data (Bui et al., 2021). Other attempts have resorted to architectural choices that are hard to scale (Zhang et al., 2021a) or cannot be utilized when text and code are intertwined (Kim et al., 2021), which is the most common mode of operation for Code-LMs. Thus, there is a need for an objective that fosters both semantic and syntactic code understanding while being easy to incorporate into modern autoregressive pre-training of Code-LMs.

Surmounting the code *data-wall*. The choice of model and corpus size in language model pre-training has traditionally been guided by *scaling laws* (Kaplan et al., 2020; Hoffmann et al., 2022) which provide a recipe on how to best trade the two off under a fixed computational budget. However, the widespread adoption of LMs has warranted a revision of scaling laws to account for inference costs and latency and favours smaller models trained on more data (Sardana et al., 2024). With quality filtering in place, transformer-based models have demonstrated the ability to keep learning on orders of magnitude more data than what was considered optimal according to scaling laws (Gadre et al., 2024). This finding has subsequently been validated for Code-LMs in both language modelling (Rozière et al., 2023) and downstream task performance (Liu et al., 2024; Li et al., 2023b; Allal et al., 2023).

Consequently, scaling up high-quality corpora for LM pre-training has hit the *data-wall* [1] — the exhaustion of openly accessible, unique and high-quality data (Bi et al., 2024). Recent LM releases (Dubey et al., 2024) may have already hit the limit of clean and openly crawled data (Penedo et al., 2024) ($\approx$15 trillion tokens). Code-LM developers face an even more alarming data-constrained reality, with the most extensive corpus of freely licensed source code containing under a trillion tokens by most measures (Lozhkov et al., 2024). Attempts to circumvent this problem include repeating data (Muennighoff et al., 2023), synthesizing data (Gunasekar et al., 2023; Wei et al., 2024) and obtaining code-adjacent data from natural language corpora using domain-classifier verdicts (Liu et al., 2024). Further attempts have sought to obtain more data through the interplay between model-generated code and programming language toolchains (Ding et al., 2024; Zheng et al., 2024). Nonetheless, the best means to source the next trillion tokens of code data remains an open question.

The case for (de-)obfuscation as an objective. We posit that using obfuscated code in Code-LM pre-training enables: **1)** Crafting objectives that better disentangle syntactic and semantic understanding of code; **2)** A way out of the code data bottleneck. Obfuscation refers to syntactically mangling source code files in a manner that preserves their semantic correctness, primarily as a measure to prevent adversaries from reverse-engineering its function (Schrittwieser et al., 2016). This is mainly achieved by substituting variable, function, class and namespace identifiers with uninformative, generic names that make it harder for readers to discern semantic intent (Lawrie et al., 2006). We provide complete examples of obfuscated code in Appendix B.3.

Specifically, we introduce a translation objective for pairs of the original source code and corresponding obfuscated code for a part of the pre-training corpus, including training the Code-LM explicitly on the structure of the obfuscated code itself, in contrast to prior work that leverages de-obfuscation objectives (Lachaux et al., 2021). Stripping away these natural language components in code (e.g., variable and class names) that reveal semantic intent (Petrescu et al., 2023) forces the Code-LM to reason about the algorithmic meaning of the code from the syntactic structure alone, without allowing it to exploit semantic shortcuts. Simultaneously, the translation objective grounds the Code-LM's understanding of the obfuscated code in the original source code, ensuring no modifications to the Code-LM are needed for inference. Additionally, obfuscation can be performed stochastically,

[1] situational-awareness.ai/from-gpt-4-to-agi/#The_data_wall

thus providing an easy way to scale up code pre-training corpora with diverse semantic-preserving transformations in a way existing rule-based techniques cannot match.

Contributions and research questions. In this work, we: **1)** create `ObscuraX`, a source-to-obfuscated-code translation pairs dataset containing approximately 55M pairs in seven programming languages; **2)** conduct a systematic investigation of grounding Code-LMs in obfuscated code, demonstrating sizeable and consistent empirical gains across a broad range of tasks and programming languages; **3)** train `ObscuraCoder`, a suite of base Code-LMs pre-trained on a data mix containing `ObscuraX`, ranging from in size 255M to 2.8B parameters.

We structure our inquiry into the benefits of grounding Code-LMs in obfuscated code around the following research questions:

- **RQ1:** Does training on obfuscated code help improve the performance of Code-LMs on tasks requiring (a) syntactic and (b) semantic code understanding?
- **RQ2:** Can supplementing identifier obfuscation with import obfuscation improve Code-LMs on library-oriented [2] code generation?
- **RQ3:** Are there any (undesirable) side-effects of obfuscation training, i.e. can it lead to performance regression on tasks on which causal LM excel, e.g., code-completion?
- **RQ4:** How does the relative performance of obfuscation-grounded training differ and scale with increasing model size?

2 Related Work

We provide a concise overview of four relevant lines of existing work: **1)** syntactically- and semantically-aware code representation learning; **2)** code translation as a pre-training objective; **3)** code representation learning based on programming language toolchains; and, most related, **4)** code (de-)obfuscation.

Syntactically- and semantically-aware code representation learning. The conventional autoregressive objective used to train Code-LMs mimics their natural language counterparts, offering the convenience of reusing the same data preparation and training pipelines. This, however, fails to capture all the nuances of programming language syntax (Velasco et al., 2024), with resulting Code-LMs displaying uncertainty over certain syntactic structures such as exception handling and type inference (Palacio et al., 2023). Prior bids to address this issue at training time have resorted to ensuring syntactically faithful generations from Code-LMs using external signals, via reinforcement learning (Parsert & Polgreen, 2024) or by training dedicated syntax verification models (Yang et al., 2023b), *inter alia*. Other approaches include (i) targeting input embeddings by deploying differentiable adversarial perturbations for improved robustness (Li et al., 2022a), (ii) incorporating auxiliary syntax tree position (Guo et al., 2022c) and node-type information (Park et al., 2023; Takerngsaksiri et al., 2024; Saberi & Fard, 2023) and (iii) test-time interventions such as syntactic grammar constrained decoding (Yang et al., 2023a; Shen et al., 2024).

Similarly, while causal LM-ing imparts some understanding of code semantics (Ahmed et al., 2023b), code representations learnt this way often fail on tests of semantic equivalence for complex code fragments (Troshin & Chirkova, 2022). Existing work has thus sought to explicitly encode the semantics of the source code by leveraging data flow graphs (Chae et al., 2017), control flow graphs (David et al., 2020), program dependence graphs (Bichsel et al., 2016) and compiler intermediate representations (Ben-Nun et al., 2018). Attempts along these lines usually employ task-specific graph-based (Du & Yu, 2023; Liu et al., 2023c; Pei et al., 2024; Guo et al., 2021), tree-based (Jiang et al., 2022b), tensor-based (Yang et al., 2023c) or hybrid (Shi et al., 2023; Wang, 2019) architectures and objectives. Other work seeks to improve semantic understanding by training Code-LMs on runtime execution traces (Wang & Su, 2019; Wang et al., 2023a; Liu et al., 2023a). In contrast, we show that teaching a model the code structure via obfuscated code while simultaneously training it to deobfuscate that code improves Code-LMs' syntactic and semantic code understanding abilities.

Translation as a pre-training objective. Most mainstream Code-LMs (Lozhkov et al., 2024; Liu et al., 2024; Mishra et al., 2024; Zhao et al., 2024)—by virtue of being trained on GitHub code—are at least implicitly grounded in comment text- to-code translation objectives. However, they have

[2] In this work, library-oriented refers to non-object-oriented Code Function Call APIs, as per the `Gorilla OpenFunctions` taxonomy. Typical examples are external packages like Numpy, NetworkX, Seaborn, etc.

a mixed record on code-to-code translation (Pan et al., 2024). Attempts to improve code-to-code translation have trained Code-LMs on parallel data, usually sourced from programming contests (Zhu et al., 2022b;a). Subsequent work tried to generalize parallel data collection implicitly via, e.g., back-translation (Ahmad et al., 2023; Chen & Lampouras, 2023; Rozière et al., 2020) with applications in code repair (Drain et al., 2021; Silva et al., 2023) and code explanation (Mahbub et al., 2023).

Translation objectives have also been employed in training to improve Code-LMs' application-specific representations. These include grounding in (i) synthetic code outlines for multi-step generation (Shi et al., 2024), (ii) syntax tree leaf nodes for retrieval (Phan & Jannesari, 2023), (iii) compiled binary code for vulnerability detection (Ahmad & Luo, 2024), (iv) pseudo-code for code comprehension (Oda et al., 2015) and (v) compiler intermediate representations for improved multilingual performance (Szafraniec et al., 2023). We extend this body of work by demonstrating the benefits of obfuscation-based translation.

Programming language toolchain grounded representation learning. There is an abundance of work that grounds code in artefacts that originate from various stages of compilation. Amongst the frontend artefacts, attempts to leverage syntax are the most common, most often encoding the (linearization of) syntactic trees with recurrent (Jiang et al., 2022a), convolutional (Mou et al., 2016), graph-based (Zhang et al., 2022) and transformer-based models (Guo et al., 2022a). Other approaches (i) use syntactic trees as a search prior for graph-based decoding (Brockschmidt et al., 2019). (ii) use syntactic tree leaf element boundaries to inform masking (Wang et al., 2021b) and tokenization (Gong et al., 2024), (iii) predict (heuristically selected) paths from the tree as an auxiliary pre-training objective (Tipirneni et al., 2024) and, (iv) leverage syntactic trees for data augmentation: heuristic generation of semantic-preserving transformations, leveraged for contrastive learning (Jain et al., 2021; Wang et al., 2021a; Bahrami et al., 2021). Further compiler frontend artefacts such as data flow graphs (Brauckmann et al., 2020) and control flow graphs (Nair et al., 2020) have also been leveraged to ground program understanding. Other work (Shojaee et al., 2023; Le et al., 2022) uses comparisons of data flow and control flow graphs between the generated and reference code to shape a reward function for reinforcement learning-guided generation.

Compiler intermediate representations have also been leveraged for grounding, as (i) a source of meaning-preserving transformations (Li et al., 2022c) and as (ii) a means to improve multilingual performance of Code-LMs (Paul et al., 2024). Further from the developer, compiler backend artefacts such as diagnostics (Ahmed et al., 2023a) and textual feedback (Ren et al., 2024; Liu et al., 2023b) have been utilized to reduce functionally erroneous Code-LMs outputs. In this work, we leverage obfuscated code, typically produced by the compilers' backend, for improved grounding and dual-channel (i.e., syntax vs. semantics) disentanglement. However, aiming for customizability and broad applicability across programming languages, we write our own custom obfuscator in this work.

Code (de-)obfuscation. Code obfuscation is the semantic-preserving modification of source code to hide its execution intent. Neural approaches mainly focus on control flow (Ma et al., 2014), data (Yala et al., 2022) and identifier (Zhou et al., 2022; Datta, 2021) obfuscation but have limited adoption due to their tendency to introduce performance regressions and execution faults (Skolka et al., 2019).

In contrast, due to its underspecified and formally intractable nature (Takang et al., 1996), code de-obfuscation is a natural task for neural methods. In particular, probabilistic (Raychev et al., 2019; Alon et al., 2018), recurrent (Bavishi et al., 2018; Lacomis et al., 2019) and transformer-based (David et al., 2020) models have all been employed for code de-obfuscation. More recently, DOBF (Lachaux et al., 2021) pioneered de-obfuscation as a pre-training objective for encoder-decoder Code-LMs, rendering it more effective than traditional sequence-to-sequence denoising objectives (Raffel et al., 2020) on many-shot translation and retrieval tasks. DOBF, however, only trains the Code-LM on the identifier map and does not backpropagate gradients w.r.t. the source or the obfuscated code; instead, it relies on initialization with a model trained on other objectives to maximize its code understanding abilities. In contrast, our `ObscuraCoder` models are trained from scratch on a bidirectional translation objective while mixing in language modelling on unpaired source and obfuscated code samples. To the best of our knowledge, we are the first to show that (de-)obfuscation translation is a viable pre-training objective for improving modern multilingual decoder-only Code-LMs. Explicitly modelling the structure of the original and obfuscated code, `ObscuraCoder` yields significant improvements in semantic robustness and zero-shot code completion compared to both vanilla autoregressive LMs and (a decoder-only variant of) DOBF. Moreover, we are the first to show that trading off semantic correctness (for a fraction of the obfuscated samples) by mangling import statements in addition to identifiers leads to large gains in library-oriented code generation.

3 OBSCURAX: SOURCE TO OBFUSCATED CODE TRANSLATION PAIRS

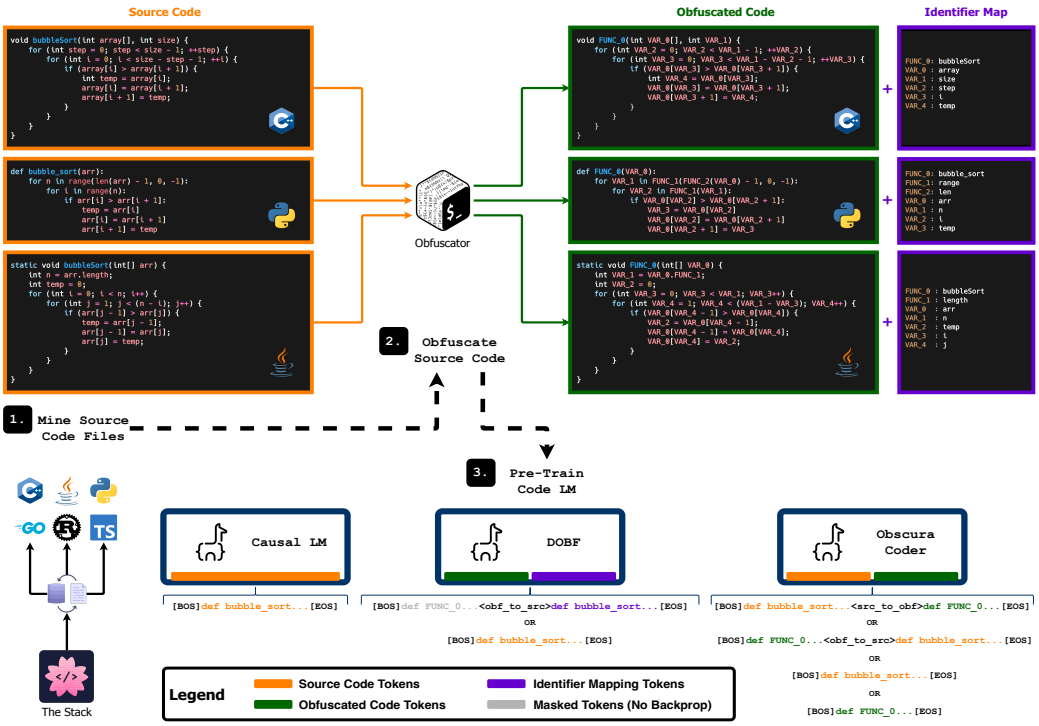

Figure 1: An overview of the `ObscuraX` data sourcing (Step 1 and 2) and the `ObscuraCoder` pre-training objective, contrasted against that of causal LM and DOBF (Step3). Notably, `ObscuraCoder` does not mask the obfuscated tokens and additionally trains on samples comprising only source or obfuscated data.

Seeking to test the hypotheses we posit in Section 1, we construct a multilingual source-code-to-obfuscated-code parallel dataset. We initiate this effort by acquiring source code files containing fewer than 2000 lines of code from the Stack corpus (Kocetkov et al., 2023) in seven languages — C, C++, Go, Java, Python, Rust and TypeScript. We ensure to retain all code sourced from research repositories [3] and accepted programming contest solutions (Puri et al., 2021), owing to their tendency to be self-contained. The resultant corpus is MinHash (Broder, 1997) de-duplicated with the surviving source files' syntax trees extracted using Tree-sitter [4]. Well-known for its fault-tolerant approach to parsing, Tree-sitter affords us a reliable way to handle open-domain source code of vague provenance whose executability and correctness are not assured.

The syntax trees obtained contain the span indices of the code's constituent syntactic elements. However, the targeted obfuscation of specific categories of syntactic structures requires reliably mapping this category information to the sub-trees of the syntax tree. Hence, we write a customized obfuscator that leverages Tree-sitter's own tree query language [5]. We build on top of queries employed by popular IDEs [6] to account for the inherent complexity in accurately tracking syntactic structure compositions and their scopes for the multi-paradigm programming languages in `ObscuraX`.

The consequent mapping between spans in the source files and syntactic element category is employed to create a mapping from the original variable, function and class names to their obfuscated counterparts. This is stochastic and is influenced by the obfuscation proportion hyper-parameter p_{obf}, which informs the proportion of the original elements we wish to obfuscate. In practice, p_{obf} is uniformly varied up to 0.9, with higher values avoided to provide the model with some information to learn the deobfuscation in a principled manner. We limit obfuscation to 150 members of a specific syntactic category to reasonably restrict the samples' de-obfuscation difficulty level. The processed

[3] 🤗 `AlgorithmicResearchGroup/arxiv_research_code`
[4] git `tree-sitter/tree-sitter`
[5] git `tree-sitter-grammars/tree-sitter-query`
[6] git `nvim-treesitter/nvim-treesitter`

counterparts are formatted as VAR_{n}, FUNC_{n} and CLASS_{n} respectively, where n ranges from 0 to 149. Finally, for cases where the same name is shared by different syntactic structures or by syntactic structures of the same variety but different scope (e.g. global and local variables), we preserve their shared semantic grounding using a catch-all category formatted as ID_{n}.

The described process is fully correctness-preserving, allowing the curation of learning objectives that better disentangle the syntactic and semantic channels. However, to improve library-oriented code generation, we sacrifice this property for ≈25% of the samples and obfuscate the imports as IMPORT_{n}. This, in turn, facilitates the Code-LM's understanding of API usage during de-obfuscation, which can be vital for its ability on infrequently invoked libraries or rare use-cases (Rabin et al., 2023). Our resultant dataset, which we dub ObscuraX, comprises ≈55M samples and is the largest multilingual collection of source code to obfuscated code translation pairs yet. Figure 1 details a high-level view of how ObscuraX is a critical part of the ObscuraCoder pre-training pipeline and also lists some examples. Refer to Appendix B.3 for more detailed samples in all languages.

4 Experimental Setup

Pre-training data recipe. We compile two comparable pre-training corpora to facilitate a fair comparison between ObscuraCoder and standard autoregressive pre-training. We begin by obtaining 105B tokens of high-quality filtered and de-duplicated text data, with the sourcing biased towards informative and instructional content from sources such as books, wikis, news articles and research papers following existing literature (Penedo et al., 2024). This is further augmented with 15B tokens of code-adjacent text and code-text data such as library documentation, coding tutorials, GitHub commits and issues. We dub this 120B-token corpus **filtered-code-text**. We then collect source code data from the Stack V1 (Kocetkov et al., 2023) corpus. Specifically, we select only the splits of languages from ObscuraX, i.e., C, C++, Go, Java, Python, Rust and TypeScript. We also select the Markdown split because of its importance for Code-LMs' instruction-following abilities (Luo et al., 2024). We subject this code corpus to further quality filtering and de-duplication, obtaining the final 76B-token corpus we term **filtered-source-code**.

We organize the causal LM pre-training corpus into two phases: (1) 90B tokens sampled from filtered-code-text, and (2) two copies of filtered-source code (152B tokens) plus the remaining 30B tokens of filtered-code-text randomly shuffled; for a total training data of ≈272B tokens. By contrast, the ObscuraCoder pre-training corpus reuses the first phase of causal LM training but compiles the second phase as a random shuffle of (1) 64B tokens from filtered-source code, (2) 30B tokens of obfuscated code from ObscuraX, (3) 58B tokens of translation pairs from ObscuraX and (4) the (same) remaining 30B tokens of filtered-code-text; also totalling ≈272B tokens. The translation pairs from ObscuraX are separated by two sentinel tokens which we add to the Code-LM's vocabulary—<src_to_obf> and <obf_to_src>—used for the respective translation directions (see Figure 1), which are each sampled for 50% of the instances.

Both the corpora are designed to follow existing Code-LM best practices of initially priming the model on text-only data before training on a code-text mixture dominated by source code data (Guo et al., 2024; Rozière et al., 2023; Hernandez et al., 2021; Tao et al., 2024). Similarly, when shuffling the various splits during corpora creation, we ensure no source code fragment is included more than three times, thus staying within the optimal learning regime (Muennighoff et al., 2023). Finally, we ensure that our corpora are n-gram decontaminated w.r.t. downstream tasks on which we evaluate (Chen et al., 2021). Appendix B.2 further details our data sourcing and filtering pipeline.

Pre-training details. For direct comparability between vanilla Code-LM pre-training and ObscuraCoder, we train both using the Llama architecture (Rozière et al., 2023) in four model sizes: 255M, 491M, 1.2B and 2.8B parameters. While most frontier models are pre-trained on corpora roughly an order of magnitude larger than ours, we argue that the scale of our pre-training runs suffice to demonstrate the merits of obfuscation grounding, as they are nearly an order of magnitude greater that what would be deemed optimal by the scaling-laws (Hoffmann et al., 2022).

All training runs are performed using an open-source implementation [7] of the Megatron-LM (Shoeybi et al., 2019) kernels and leverage Deepspeed Stage-2 (Rajbhandari et al., 2020) sharding on BF16 precision. We use the Adam optimizer (Kingma & Ba, 2015), with a learning rate of 5e-4 and a cosine annealed schedule that terminates at 5% of the peak learning rate. The models are trained

[7] git EleutherAI/gpt-neox

using FlashAttention-2 (Dao, 2024) with a sequence length of 2048 tokens and a batch size of 256 for 520K steps. We use a custom byte-level BPE tokenizer (Wang et al., 2020) with a vocabulary size of 49152 tokens in total that we train on a 5B token corpus comprising of code and code-adjacent text data. The tokenizer is augmented with special tokens pertaining to the outputs of our obfuscator — `ID_{n}`, `CLASS_{n}`, `FUNC_{n}`, `VAR_{n}` and `IMPORT_{n}` — `n` ranging from 0 to 149.

Inference details. For inference, we resort to a Paged Attention (Kwon et al., 2023) enabled fork of an open-source evaluation harness.[8] We conduct our evaluation using model checkpoints loaded in half-precision (for efficiency) with nucleus sampling (p=0.9). All inference runs are conducted on Nvidia A100 80GB GPUs with 95% of the GPU VRAM explicitly reserved for vLLMs GPU pages. We further set aside 64GB of RAM as a CPU swap, allowing for offloading pages to the CPU during bursts of long sequences. We additionally limit the continuous batching parameter to 32.

5 Experimental Research Questions & Results

Our evaluation comprises a mix of five zero-shot and fine-tuning tasks, selected to provide answers to the three research questions from Section 1. For each task, we compare the performance of `ObscuraCoder` against an equally-sized autoregressive LM. We further contextualize the sample-efficiency benefits of our obfuscation objective by including comparisons to seven frontier Code-LMs from the Deepseek-Coder (Guo et al., 2024), CodeGemma (Zhao et al., 2024), Phi (Gunasekar et al., 2023) and StarCoder (Lozhkov et al., 2024; Li et al., 2023b) families that are under 3B parameters in size and pre-trained on corpora between 5x to 22x larger than the one used to train `ObscuraCoder`.

For fine-tuning tasks, all models are trained for three epochs using a cosine scheduler with a peak learning rate of 5e-5 using LoRA (Xu et al., 2024) modules coupled with trainable embeddings. We follow prior work (Poth et al., 2023) and train LoRA modules with rank 64 for classification tasks and 256 for open-ended generation. Unless otherwise specified, we use greedy decoding at inference.

Model	Pre-Training Token Count	Defect		ReCode			BigCodeBench		
		Acc.	F1-Score	Format	Syntax	Function	Pass@1	Pass@10	Pass@25
CausalLM 255M	272B	62.39	61.11	11.37	8.98	3.13	2.12	3.87	4.32
ObscuraCoder 255M	272B	62.80 **+0.41**	62.14 **+1.03**	14.17 **+2.80**	11.80 **+2.82**	4.95 **+1.82**	2.64 **+0.52**	4.45 **+0.58**	4.97 **+0.65**
CausalLM 491M	272B	63.36	62.88	15.04	11.30	5.22	2.83	6.29	8.91
ObscuraCoder 491M	272B	64.27 **+0.91**	63.72 **+0.84**	23.88 **+8.84**	20.04 **+8.74**	8.01 **+2.78**	3.02 **+0.19**	7.09 **+0.80**	11.97 **+3.06**
CausalLM 1.2B	272B	63.49	63.59	27.65	21.01	11.30	7.98	14.18	23.77
ObscuraCoder 1.2B	272B	64.49 **+1.00**	64.76 **+1.17**	36.37 **+8.73**	27.86 **+6.85**	16.66 **+5.36**	9.67 **+1.69**	17.89 **+3.71**	28.04 **+4.27**
CausalLM 2.8B	272B	64.73	64.36	34.70	26.20	14.72	10.79	22.87	31.95
ObscuraCoder 2.8B	272B	65.58 **+0.85**	65.42 **+1.06**	45.29 **+10.59**	35.97 **+9.77**	20.11 **+5.38**	14.77 **+3.98**	31.94 **+9.07**	48.78 **+16.83**
StarCoderBase 1B	1.0T	64.96	64.68	28.97	25.92	13.84	5.83	13.58	22.77
DeepSeekCoder 1.3B	2.0T	65.21	64.59	48.04	44.42	25.27	19.57	34.18	47.84
StarCoderBase 3B	1.0T	65.82	64.92	37.21	30.25	17.49	8.26	26.79	37.74
StarCoder2 3B	3.3T	66.37	65.55	53.82	47.66	24.58	16.38	36.11	48.81
StableCode 3B	5.6T	62.31	61.71	50.09	43.60	24.15	12.98	31.75	49.44
CodeGemma 2B	3.5T	61.93	60.57	10.31	9.62	7.04	17.39	33.12	43.66
Phi-2	1.4T	64.56	63.97	59.99	53.71	32.71	21.83	38.54	53.78

Table 1: Results table for **RQ1** and **RQ2** comparing syntactic understanding for code defect detection on CodeXGLUE (Lu et al., 2021), semantic understanding for robust code completion on ReCode (Wang et al., 2023b) and library-oriented code generation on BigCodeBench (Zhuo et al., 2024). For detailed split-level breakdowns in ReCode results refer to Tables 7 to 9 in Appendix C.

RQ1: Obfuscation translation → better syntactic and semantic understanding?

We first test if our obfuscation-based training leads to a better understanding of complex semantic and syntactic structures, which are crucial for solving complex tasks like detecting vulnerable code.

CodeXGLUE Code defect detection. Detecting code defects requires integrating information from both channels due to the inherent dual-channel nature of software bugs. While some bugs, such as insecure argument usage, demand a grasp of syntactic structures (Yamaguchi et al., 2014; Ray et al., 2016) and are easier to detect when code syntax is simplified (Thummalapenta & Xie, 2011), others, like memory leaks and null pointer references, require an understanding of control and data flow semantics (Zhou et al., 2019; Wang et al., 2016). We benchmark `ObscuraCoder` and baseline CodeLMs on a binary defect detection classification task from CodeXGLUE (Lu et al., 2021).

[8] git `bigcode-project/bigcode-evaluation-harness`

ReCode. We next employ five differently seeded ReCode (Wang et al., 2023b) transformations of HumanEval (Chen et al., 2021). These examine a Code-LM's (zero-shot) robustness to semantically-preserving syntactic perturbations based on its greedy decoding completions on code generation prompts mangled using an extensive battery of Format, Function, and Syntax transforms, thus explicitly testing the model's understanding of the interplay between code syntax and semantics.

Results. Table 1 summarizes the results of the above two tasks. We observe that obfuscation-based objectives of `ObscuraCoder` consistently bring significant performance gains on top of causal language modelling for all model sizes and both tasks. The gains are particularly large on the ReCode task, keeping with prior work, which reports that translation between semantically equivalent programs benefits understanding of code intent (Guizzo et al., 2024). The `ObscuraCoder` models frequently match or surpass the performance of the next-size CausalLMs, and `ObscuraCoder 2.8B` beats several frontier models on semantic code understanding.

`RQ2`: Obfuscating imports → better library-oriented code generation?

BigCodeBench. We investigate the effects of our de-obfuscation objective—and in particular, the choice to obfuscate imports and package names—on Code-LMs' abilities in library-oriented code generation. Such obfuscation, we hypothesized, should have forced Code-LMs to understand what library APIs do and when to invoke them. Prior work (Zhang et al., 2021b) points to the demanding nature of package de-obfuscation. Because of this, we evaluate our Code-LMs zero-shot on the BigCodeBench (Zhuo et al., 2024) benchmark [9] in the completion setting, thus testing models' competence in API usage. We sample 50 generations per problem and report the `pass@k` performance for $\texttt{k} \in \{1, 10, 25\}$. The `pass@1` performance emulates usage scenarios where correctness is paramount: we thus decode with temperature sampling with a low temperature of 0.1. In contrast, `pass@10` and `pass@25` estimates correspond to scenarios where creativity and diversity of generations are more important: we thus use a higher sampling temperature of 0.8.

Results. The results in the BigCodeBench column of Table 1 demonstrate substantial gains in library-oriented code generation performance facilitated by obfuscation-grounded pre-training that includes obfuscation of imports and package names. We believe that our design choice to represent obfuscated packages, which are often multi-token strings, with a single (special) token is one key driver of these gains, in line with suggestions from prior work (Hadi et al., 2022). Particularly encouraging is the observation that `ObscuraCoder`'s library-oriented generation gains (over the vanilla autoregressive LM-ing) widen with increasing model size. `ObscuraCoder` represents a significant advancement in addressing API misuse, a major obstacle to wider Code-LM adoption (Wang et al., 2024b). Promisingly, `ObscuraCoder 2.8B` is competitive with the best frontier open-source models in this challenging setting that most closely mimics real-world usage.

`RQ3`: Are there negative side-effects of obfuscation-based pre-training?

We next test whether improvements in dual-channel understanding and library-oriented code generation come at a cost, i.e., performance deterioration, for zero-shot multilingual code completion, which is still the most common application of Code-LMs.

Multipl-E. We first carry out multlingual evaluation of `pass@k` performance for $\texttt{k} \in \{1, 10, 100\}$ in five languages (C++, Java, Python, Rust and TypeScript) using the Multipl-E benchmark (Cassano et al., 2023). We sample 200 generations per problem. Like in BigCodeBench evaluation, we set the sampling temperature to 0.1 for `pass@1` and to 0.8 for `pass@10` and `pass@100`.

CommitChronicle. As a further measure of multilingual code competence, we evaluate Code-LMs on CommitChronicle (Eliseeva et al., 2023), a fine-tuning code-change summarization benchmark in seven languages (C, C++, Go, Java, Python, Rust and TypeScript). Summarizing code changes is a good proxy for multilingual code understanding due to its dual nature w.r.t. code completion (Wei et al., 2019) and the tendency of the capabilities in each task—text-to-code and code-to-text—to reinforce the other. We construct language-specific splits by filtering the original dataset and partitioning 75%, 15% and 10% of the data into train, validation and test splits, respectively. In fine-tuning, we ensure that autoregressive losses are backpropagated only for the target summaries.

Results. Table 2 points to the strong performance of `ObscuraCoder` on both multilingual benchmarks. Not only do our (de-)obfuscation pre-training objectives not hurt multilingual generation

[9] As early adopters, we ran `v0.1.0` of the benchmark, where 13 network usage problems were found to contain unreliable tests and were discarded. We report all results on the remaining 1127 problems.

Model	Pre-Training Token Count	Multipl-E			Commit Chronicle		
		Pass@1	Pass@10	Pass@100	ROUGE-1	ROUGE-2	ROUGE-L
CausalLM 255M	272B	4.29	7.13	14.36	32.76	10.15	31.14
ObscuraCoder 255M	272B	5.93 +1.65	9.66 +2.53	18.20 +3.84	33.40 +0.65	10.64 +0.49	31.70 +0.56
CausalLM 491M	272B	5.84	10.64	20.69	34.08	11.02	32.40
ObscuraCoder 491M	272B	8.76 +2.92	14.33 +3.69	25.86 +5.17	34.85 +0.78	11.44 +0.42	33.03 +0.63
CausalLM 1.2B	272B	12.07	19.50	34.10	35.65	11.96	33.79
ObscuraCoder 1.2B	272B	18.34 +6.27	29.34 +9.83	47.79 +13.69	37.00 +1.35	13.20 +1.24	35.34 +1.55
CausalLM 2.8B	272B	16.70	28.83	47.53	36.72	12.89	35.07
ObscuraCoder 2.8B	272B	23.55 +6.85	39.41 +10.59	61.93 +14.39	38.07 +1.35	14.15 +1.26	36.60 +1.53
StarCoderBase 1B	1.0T	13.12	22.63	37.78	37.05	13.19	35.30
DeepSeekCoder 1.3B	2.0T	26.81	47.46	71.70	35.86	12.95	34.42
StarCoderBase 3B	1.0T	19.21	33.48	57.60	38.65	14.44	36.88
StarCoder2 3B	3.3T	27.38	49.63	72.86	38.75	14.58	36.96
StableCode 3B	5.6T	26.73	47.93	68.88	38.08	14.35	36.69
CodeGemma 2B	3.5T	22.71	37.98	62.14	36.14	13.29	35.28
Phi-2	1.4T	22.27	39.96	57.30	35.24	12.16	33.61

Table 2: Results table for **RQ3** comparing multilingual code completion averages on Multipl-E (Cassano et al., 2023) (C++, Java, Python, Rust, TypeScript) and multilingual commit summarization averages on CommitChronicle (Eliseeva et al., 2023) (C, C++, Go, Java, Python, Rust, TypeScript). For detailed language-wise breakdowns in Multipl-E results refer to Tables 10 to 12 and CommitChronicle results refer to Tables 13 to 15 in Appendix C.

performance compared to causal LM-ing, but they seem to improve it. Moreover, similar to the results on BigCodeBench, the gains from obfuscation pre-training over autoregressive LM-ing generally increase with the model size. Prior work (Bavarian et al., 2022; Wang et al., 2021b) refers to incorporation of additional objectives during pre-training as being "for free" when these objectives do not incur any deterioration in autoregressive performance. Owing to its pre-training objective, `ObscuraCoder` goes beyond this and actually displays strong evidence of positive transfer from improved syntactic, semantic and API understanding to multilingual code completion and summarization.

RQ4: Do the benefits of obfuscation-grounding scale?

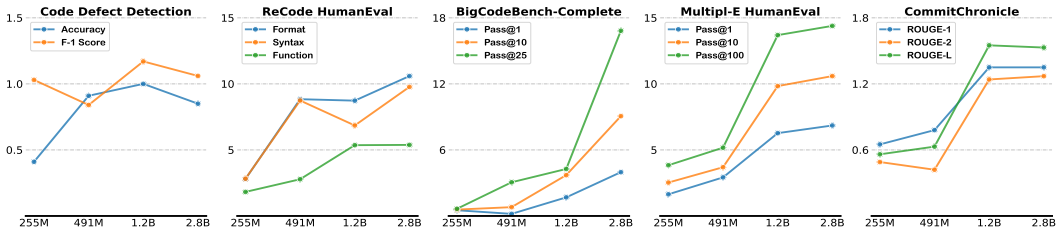

Figure 2: Downstream performance gains of `ObscuraCoder` viz-a-viz Causal Code-LMs.

Due to computational constraints, we only train models up to 2.8B parameters in size. We attempt to extrapolate the effects of obfuscation pre-training to parameter sizes resembling those of the most widely adopted Code-LMs [10] by investigating the scaling trends of downstream task performance.

Figure 2 summarizes the performance of `ObscuraCoder` across all five downstream tasks as a function of model size. Expectedly, scaling model size brings more modest gains in fine-tuning setups (CodeXGLUE Defect Detection and CommitChronicle) than in zero-shot evaluations: we observe saturation in fine-tuning performance also for the larger frontier model (see Table 1 and Table 2). Scaling trends are very prominent in zero-shot tasks, especially library-oriented (BigCodeBench) and multilingual (Multipl-E) generation: this is encouraging, as these tasks correspond to Code-LMs' most prevalent modes of use. Finally, we observe the strongest scaling trends in settings that reward diverse code generation: high pass rate estimates on high-temperature generations. This is promising for generate-then-rerank settings, where being able to choose from a deep bank of valid solutions drives further improvements to end-user utility (Li et al., 2022b; Chen et al., 2023) and opens room for optimizing non-functional axes (Waghjale et al., 2024).

Generally, our results strongly agree with prior work that establishes the benefits of information-preserving transforms in LM pre-training (Yuan & Liu, 2022; Maini et al., 2024). Specifically, we establish that Code-LMs can substantially benefit from (de-)obfuscation-based translation objectives.

[10] At the time of writing, this ranges from approx. 3x to 150x larger than the largest `ObscuraCoder` model.

6 ABLATIONS

How Does `ObscuraCoder` Compare Against DOBF? We attempt an explicit head-to-head comparison between the de-obfuscation objective pioneered by DOBF (Lachaux et al., 2021) and our obfuscation-grounded translation objective. However, the originally released DOBF model is an encoder-decoder model with 126M parameters, making it half the size of our smallest model. Hence, to facilitate a fair comparison, we train a decoder-only adaptation of DOBF on a modified version of the `ObscuraCoder` corpus of ≈272B tokens outlined in Section 4. Specifically, the 30B tokens of obfuscated code and the 58B tokens of translation pairs are replaced by 88B tokens of source code to identifier map translation pairs sourced from `ObscuraX` (see Figure 1 for the DOBF objective).

We train size-matched versions of DOBF on the above-detailed corpus. Staying faithful to the original de-obfuscation objective, we mask the obfuscated code from the loss computation and only back-propagate gradients for the identifier maps targets.[11] [12] Results in Table 3 show that `ObscuraCoder` has an advantage over DOBF across the board, with particularly prominent gaps on on zero-shot tasks. Promisingly, the performance gap appears to widen with increasing model size.

Model	Defect	ReCode			BigCodeBench		Multipl-E		CC
	F-1 Score	Format	Syntax	Function	Pass@1	Pass@25	Pass@1	Pass@100	ROUGE-2
ObscuraCoder 255M	62.14	14.17	11.80	4.95	2.64	4.97	5.93	18.20	10.64
DOBF 255M	62.22 **+0.08**	14.00 **-0.17**	11.67 **-0.13**	4.97 **+0.02**	2.68 **+0.04**	4.79 **-0.18**	4.62 **-1.31**	16.85 **-1.35**	10.36 **-0.28**
CausalLM-CP 255M	62.03 **-0.11**	11.05 **-3.12**	8.27 **-3.53**	2.81 **-2.14**	1.86 **-0.78**	4.21 **-0.76**	4.00 **-1.93**	15.10 **-3.10**	10.08 **-0.56**
ObscuraCoder 491M	63.72	23.88	20.04	8.01	3.02	11.97	8.76	25.86	11.44
DOBF 491M	63.65 **-0.07**	15.93 **-7.95**	12.70 **-7.33**	6.91 **-1.10**	2.89 **-0.13**	10.06 **-1.91**	6.76 **-2.00**	20.32 **-5.54**	11.15 **-0.29**
CausalLM-CP 491M	62.27 **-1.45**	15.59 **-8.29**	11.91 **-8.13**	6.18 **-1.83**	2.46 **-0.56**	8.12 **-3.85**	6.89 **-1.87**	22.04 **-3.82**	10.92 **-0.52**
ObscuraCoder 1.2B	64.76	36.37	27.86	16.66	9.67	28.04	18.34	47.79	13.20
DOBF 1.2B	64.67 **-0.09**	30.06 **-6.31**	23.65 **-4.21**	13.07 **-3.59**	9.14 **-0.53**	25.95 **-2.09**	15.25 **-3.09**	40.30 **-7.49**	12.52 **-0.68**
CausalLM-CP 1.2B	63.46 **-1.30**	26.58 **-9.79**	20.71 **-7.15**	11.38 **-5.28**	6.78 **-2.89**	19.84 **-8.20**	13.34 **-5.00**	37.77 **-10.02**	12.16 **-1.04**
ObscuraCoder 2.8B	65.42	45.29	35.97	20.11	14.77	48.78	23.55	61.93	14.15
DOBF 2.8B	65.37 **-0.05**	37.92 **-7.36**	29.10 **-6.87**	16.69 **-3.42**	13.18 **-1.59**	43.65 **-5.13**	20.32 **-3.23**	51.27 **-10.66**	13.39 **-0.76**
CausalLM-CP 2.8B	63.94 **-1.48**	35.65 **-9.64**	26.86 **-9.11**	15.85 **-4.26**	10.68 **-4.09**	33.74 **-15.04**	17.92 **-5.63**	47.73 **-14.20**	13.24 **-0.91**

Table 3: Results table for the comparative ablations of `ObscuraCoder` compared to a comparatively pre-trained DOBF model and a comparatively pre-trained CausalLM model, continually pre-trained on 8B tokens (CausalLM-CP) using obfuscation-grounded translation.

Does Post-Hoc Obfuscation Training Suffice? Finally, we test the effectiveness of our obfuscation translation training when it is post-hoc applied on an already pre-trained causal Code-LM. To this end, we subsample 8B tokens of translation pairs from `ObscuraX` and continue pre-training our causal LMs on obfuscation translation. The resulting models, dubbed CausalLM-CP, are compared against `ObscuraCoder` in Table 3. Unfortunately, we find that post-hoc obfuscation training does not confer performance gains to causally pre-trained LMs. This points to the importance of early obfuscation-grounding of code representations.

7 CONCLUSION

This work investigates the effects of obfuscation-grounding Code-LMs' representations across multiple programming languages. We create `ObscuraX`, an ≈119B-token source-code-to-obfuscated-code parallel dataset containing ≈55M training instances across seven languages. We then pre-train `ObscuraCoder`, a suite of models ranging from 255M to 2.8B parameters in size, on a ≈272B-token corpus that samples from `ObscuraX` and demonstrate that obfuscation grounding leads to substantial gains in syntactic and semantic understanding of code. We further uncover broader benefits of adding obfuscation translation to CodeLMs' pre-training mix: improved library-oriented code generation, multilingual code completion and summarization. Our results render obfuscation-based code translation a viable objective for bypassing the code data bottleneck. We hope our promising results catalyze the incorporation of obfuscated code into the pre-training recipes of frontier Code-LMs.

[11] While this leads to fewer gradient back-propagations for the DOBF model, we maintain that this is an inherent disadvantage that training with this objective in the current data-constrained regime incurs.

[12] The mainline GPTNeoX library does not support custom loss masking. For DOBF training, we employ a hitherto un-merged pull request containing the feature: `EleutherAI/gpt-neox/pull/1240`

REPRODUCIBILITY STATEMENT

In the interest of reproducibility, the authors provide supplementary material containing anonymised experiment training and inference code, along with the Docker environments to execute them. Similarly, multiple `ObscuraX` samples in every language are also included for reference. While the `ObscuraCoder` models and the full `ObscuraX` dataset are too large to be shared without compromising anonymity, the authors commit to releasing these artefacts on acceptance. Seeking to aid replication efforts, we include detailed information about the construction of `ObscuraX` in Section 3 along with instructions to replicate our pre-training corpora in Section 4 and Appendix B.2. We also outline the exact model training and architectural attributes in Appendix A.

REFERENCES

Julien Abadji, Pedro Javier Ortiz Suárez, Laurent Romary, and Benoît Sagot. Towards a cleaner document-oriented multilingual crawled corpus. In *LREC*, pp. 4344–4355. European Language Resources Association, 2022.

Iftakhar Ahmad and Lannan Luo. Unsupervised binary code translation with application to code similarity detection and vulnerability discovery. *CoRR*, abs/2404.19025, 2024. doi: 10.48550/ARXIV.2404.19025. URL https://doi.org/10.48550/arXiv.2404.19025.

Wasi Uddin Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. Summarize and generate to back-translate: Unsupervised translation of programming languages. In Andreas Vlachos and Isabelle Augenstein (eds.), *Proceedings of the 17th Conference of the European Chapter of the Association for Computational Linguistics, EACL 2023, Dubrovnik, Croatia, May 2-6, 2023*, pp. 1520–1534. Association for Computational Linguistics, 2023. doi: 10.18653/V1/2023.EACL-MAIN.112. URL https://doi.org/10.18653/v1/2023.eacl-main.112.

Toufique Ahmed, Noah Rose Ledesma, and Premkumar T. Devanbu. Synshine: Improved fixing of syntax errors. *IEEE Trans. Software Eng.*, 49(4):2169–2181, 2023a. doi: 10.1109/TSE.2022.3212635. URL https://doi.org/10.1109/TSE.2022.3212635.

Toufique Ahmed, Dian Yu, Chengxuan Huang, Cathy Wang, Prem Devanbu, and Kenji Sagae. Towards understanding what code language models learned. *CoRR*, abs/2306.11943, 2023b. doi: 10.48550/ARXIV.2306.11943. URL https://doi.org/10.48550/arXiv.2306.11943.

Loubna Ben Allal, Raymond Li, Denis Kocetkov, Chenghao Mou, Christopher Akiki, Carlos Muñoz Ferrandis, Niklas Muennighoff, Mayank Mishra, Alex Gu, Manan Dey, Logesh Kumar Umapathi, Carolyn Jane Anderson, Yangtian Zi, Joel Lamy-Poirier, Hailey Schoelkopf, Sergey Troshin, Dmitry Abulkhanov, Manuel Romero, Michael Lappert, Francesco De Toni, Bernardo García del Río, Qian Liu, Shamik Bose, Urvashi Bhattacharyya, Terry Yue Zhuo, Ian Yu, Paulo Villegas, Marco Zocca, Sourab Mangrulkar, David Lansky, Huu Nguyen, Danish Contractor, Luis Villa, Jia Li, Dzmitry Bahdanau, Yacine Jernite, Sean Hughes, Daniel Fried, Arjun Guha, Harm de Vries, and Leandro von Werra. Santacoder: don't reach for the stars! *CoRR*, abs/2301.03988, 2023. doi: 10.48550/ARXIV.2301.03988. URL https://doi.org/10.48550/arXiv.2301.03988.

Miltiadis Allamanis. The adverse effects of code duplication in machine learning models of code. In *Onward!*, pp. 143–153. ACM, 2019.

Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. A general path-based representation for predicting program properties. In Jeffrey S. Foster and Dan Grossman (eds.), *Proceedings of the 39th ACM SIGPLAN Conference on Programming Language Design and Implementation, PLDI 2018, Philadelphia, PA, USA, June 18-22, 2018*, pp. 404–419. ACM, 2018. doi: 10.1145/3192366.3192412. URL https://doi.org/10.1145/3192366.3192412.

Mehdi Bahrami, N. C. Shrikanth, Yuji Mizobuchi, Lei Liu, Masahiro Fukuyori, Wei-Peng Chen, and Kazuki Munakata. Augmentedcode: Examining the effects of natural language resources in code retrieval models. *CoRR*, abs/2110.08512, 2021. URL https://arxiv.org/abs/2110.08512.

Mohammad Bavarian, Heewoo Jun, Nikolas Tezak, John Schulman, Christine McLeavey, Jerry Tworek, and Mark Chen. Efficient training of language models to fill in the middle. *CoRR*, abs/2207.14255, 2022. doi: 10.48550/ARXIV.2207.14255. URL https://doi.org/10.48550/arXiv.2207.14255.

Rohan Bavishi, Michael Pradel, and Koushik Sen. Context2name: A deep learning-based approach to infer natural variable names from usage contexts. *CoRR*, abs/1809.05193, 2018. URL http://arxiv.org/abs/1809.05193.

Tal Ben-Nun, Alice Shoshana Jakobovits, and Torsten Hoefler. Neural code comprehension: A learnable representation of code semantics. In Samy Bengio, Hanna M. Wallach, Hugo Larochelle, Kristen Grauman, Nicolò Cesa-Bianchi, and Roman Garnett (eds.), *Advances in Neural Information Processing Systems 31: Annual Conference on Neural Information Processing Systems 2018, NeurIPS 2018, December 3-8, 2018, Montréal, Canada*, pp. 3589–3601, 2018. URL https://proceedings.neurips.cc/paper/2018/hash/17c3433fecc21b57000debdf7ad5c930-Abstract.html.

Xiao Bi, Deli Chen, Guanting Chen, Shanhuang Chen, Damai Dai, Chengqi Deng, Honghui Ding, Kai Dong, Qiushi Du, Zhe Fu, Huazuo Gao, Kaige Gao, Wenjun Gao, Ruiqi Ge, Kang Guan, Daya Guo, Jianzhong Guo, Guangbo Hao, Zhewen Hao, Ying He, Wenjie Hu, Panpan Huang, Erhang Li, Guowei Li, Jiashi Li, Yao Li, Y. K. Li, Wenfeng Liang, Fangyun Lin, Alex X. Liu, Bo Liu, Wen Liu, Xiaodong Liu, Xin Liu, Yiyuan Liu, Haoyu Lu, Shanghao Lu, Fuli Luo, Shirong Ma, Xiaotao Nie, Tian Pei, Yishi Piao, Junjie Qiu, Hui Qu, Tongzheng Ren, Zehui Ren, Chong Ruan, Zhangli Sha, Zhihong Shao, Junxiao Song, Xuecheng Su, Jingxiang Sun, Yaofeng Sun, Minghui Tang, Bingxuan Wang, Peiyi Wang, Shiyu Wang, Yaohui Wang, Yongji Wang, Tong Wu, Y. Wu, Xin Xie, Zhenda Xie, Ziwei Xie, Yiliang Xiong, Hanwei Xu, R. X. Xu, Yanhong Xu, Dejian Yang, Yuxiang You, Shuiping Yu, Xingkai Yu, B. Zhang, Haowei Zhang, Lecong Zhang, Liyue Zhang, Mingchuan Zhang, Minghua Zhang, Wentao Zhang, Yichao Zhang, Chenggang Zhao, Yao Zhao, Shangyan Zhou, Shunfeng Zhou, Qihao Zhu, and Yuheng Zou. Deepseek LLM: scaling open-source language models with longtermism. *CoRR*, abs/2401.02954, 2024. doi: 10.48550/ARXIV.2401.02954. URL https://doi.org/10.48550/arXiv.2401.02954.

Benjamin Bichsel, Veselin Raychev, Petar Tsankov, and Martin T. Vechev. Statistical deobfuscation of android applications. In Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi (eds.), *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pp. 343–355. ACM, 2016. doi: 10.1145/2976749.2978422. URL https://doi.org/10.1145/2976749.2978422.

Alexander Brauckmann, Andrés Goens, Sebastian Ertel, and Jerónimo Castrillón. Compiler-based graph representations for deep learning models of code. In Louis-Noël Pouchet and Alexandra Jimborean (eds.), *CC '20: 29th International Conference on Compiler Construction, San Diego, CA, USA, February 22-23, 2020*, pp. 201–211. ACM, 2020. doi: 10.1145/3377555.3377894. URL https://doi.org/10.1145/3377555.3377894.

Marc Brockschmidt, Miltiadis Allamanis, Alexander L. Gaunt, and Oleksandr Polozov. Generative code modeling with graphs. In *7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019*. OpenReview.net, 2019. URL https://openreview.net/forum?id=Bke4KsA5FX.

Andrei Z. Broder. On the resemblance and containment of documents. In Bruno Carpentieri, Alfredo De Santis, Ugo Vaccaro, and James A. Storer (eds.), *Compression and Complexity of SEQUENCES 1997, Positano, Amalfitan Coast, Salerno, Italy, June 11-13, 1997, Proceedings*, pp. 21–29. IEEE, 1997. doi: 10.1109/SEQUEN.1997.666900. URL https://doi.org/10.1109/SEQUEN.1997.666900.

Nghi D. Q. Bui, Yijun Yu, and Lingxiao Jiang. Self-supervised contrastive learning for code retrieval and summarization via semantic-preserving transformations. In Fernando Diaz, Chirag Shah, Torsten Suel, Pablo Castells, Rosie Jones, and Tetsuya Sakai (eds.), *SIGIR '21: The 44th International ACM SIGIR Conference on Research and Development in Information Retrieval, Virtual Event, Canada, July 11-15, 2021*, pp. 511–521. ACM, 2021. doi: 10.1145/3404835.3462840. URL https://doi.org/10.1145/3404835.3462840.

Casey Casalnuovo, Kevin Lee, Hulin Wang, Prem Devanbu, and Emily Morgan. Do people prefer "natural" code? *CoRR*, abs/1910.03704, 2019. URL http://arxiv.org/abs/1910.03704.

Casey Casalnuovo, Earl T. Barr, Santanu Kumar Dash, Prem Devanbu, and Emily Morgan. A theory of dual channel constraints. In Gregg Rothermel and Doo-Hwan Bae (eds.), *ICSE-NIER 2020: 42nd International Conference on Software Engineering, New Ideas and Emerging Results, Seoul, South Korea, 27 June - 19 July, 2020*, pp. 25–28. ACM, 2020. doi: 10.1145/3377816.3381720. URL https://doi.org/10.1145/3377816.3381720.

Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q. Feldman, Arjun Guha, Michael Greenberg, and Abhinav Jangda. Multipl-e: A scalable and polyglot approach to benchmarking neural code generation. *IEEE Trans. Software Eng.*, 49(7):3675–3691, 2023. doi: 10.1109/TSE.2023.3267446. URL https://doi.org/10.1109/TSE.2023.3267446.

Kwonsoo Chae, Hakjoo Oh, Kihong Heo, and Hongseok Yang. Automatically generating features for learning program analysis heuristics for c-like languages. *Proc. ACM Program. Lang.*, 1(OOPSLA): 101:1–101:25, 2017. doi: 10.1145/3133925. URL https://doi.org/10.1145/3133925.

Saikat Chakraborty, Toufique Ahmed, Yangruibo Ding, Premkumar T. Devanbu, and Baishakhi Ray. Natgen: generative pre-training by "naturalizing" source code. In Abhik Roychoudhury, Cristian Cadar, and Miryung Kim (eds.), *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*, pp. 18–30. ACM, 2022. doi: 10.1145/3540250.3549162. URL https://doi.org/10.1145/3540250.3549162.

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. Codet: Code generation with generated tests. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net, 2023. URL https://openreview.net/forum?id=ktrw68Cmu9c.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harri Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. Evaluating large language models trained on code. *CoRR*, abs/2107.03374, 2021. URL https://arxiv.org/abs/2107.03374.

Pinzhen Chen and Gerasimos Lampouras. Exploring data augmentation for code generation tasks. In Andreas Vlachos and Isabelle Augenstein (eds.), *Findings of the Association for Computational Linguistics: EACL 2023, Dubrovnik, Croatia, May 2-6, 2023*, pp. 1497–1505. Association for Computational Linguistics, 2023. doi: 10.18653/V1/2023.FINDINGS-EACL.114. URL https://doi.org/10.18653/v1/2023.findings-eacl.114.

Tri Dao. Flashattention-2: Faster attention with better parallelism and work partitioning. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024. URL https://openreview.net/forum?id=mZn2Xyh9Ec.

Siddhartha Datta. Deepobfuscode: Source code obfuscation through sequence-to-sequence networks. In Kohei Arai (ed.), *Intelligent Computing - Proceedings of the 2021 Computing Conference, Volume 2, SAI 2021, Virtual Event, 15-16 July, 2021*, volume 284 of *Lecture Notes in Networks and Systems*, pp. 637–647. Springer, 2021. doi: 10.1007/978-3-030-80126-7_45. URL https://doi.org/10.1007/978-3-030-80126-7_45.

Yaniv David, Uri Alon, and Eran Yahav. Neural reverse engineering of stripped binaries using augmented control flow graphs. *Proc. ACM Program. Lang.*, 4(OOPSLA):225:1–225:28, 2020. doi: 10.1145/3428293. URL https://doi.org/10.1145/3428293.

Yangruibo Ding, Benjamin Steenhoek, Kexin Pei, Gail E. Kaiser, Wei Le, and Baishakhi Ray. TRACED: execution-aware pre-training for source code. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*, pp. 36:1–36:12. ACM, 2024. doi: 10.1145/3597503.3608140. URL https://doi.org/10.1145/3597503.3608140.

Dawn Drain, Colin B. Clement, Guillermo Serrato, and Neel Sundaresan. Deepdebug: Fixing python bugs using stack traces, backtranslation, and code skeletons. *CoRR*, abs/2105.09352, 2021. URL https://arxiv.org/abs/2105.09352.

Yali Du and Zhongxing Yu. Pre-training code representation with semantic flow graph for effective bug localization. In Satish Chandra, Kelly Blincoe, and Paolo Tonella (eds.), *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2023, San Francisco, CA, USA, December 3-9, 2023*, pp. 579–591. ACM, 2023. doi: 10.1145/3611643.3616338. URL https://doi.org/10.1145/3611643.3616338.

Abhimanyu Dubey, Abhinav Jauhri, Abhinav Pandey, Abhishek Kadian, Ahmad Al-Dahle, Aiesha Letman, Akhil Mathur, Alan Schelten, Amy Yang, Angela Fan, Anirudh Goyal, Anthony Hartshorn, Aobo Yang, Archi Mitra, Archie Sravankumar, Artem Korenev, Arthur Hinsvark, Arun Rao, Aston Zhang, Aurélien Rodriguez, Austen Gregerson, Ava Spataru, Baptiste Rozière, Bethany Biron, Binh Tang, Bobbie Chern, Charlotte Caucheteux, Chaya Nayak, Chloe Bi, Chris Marra, Chris McConnell, Christian Keller, Christophe Touret, Chunyang Wu, Corinne Wong, Cristian Canton Ferrer, Cyrus Nikolaidis, Damien Allonsius, Daniel Song, Danielle Pintz, Danny Livshits, David Esiobu, Dhruv Choudhary, Dhruv Mahajan, Diego Garcia-Olano, Diego Perino, Dieuwke Hupkes, Egor Lakomkin, Ehab AlBadawy, Elina Lobanova, Emily Dinan, Eric Michael Smith, Filip Radenovic, Frank Zhang, Gabriel Synnaeve, Gabrielle Lee, Georgia Lewis Anderson, Graeme Nail, Grégoire Mialon, Guan Pang, Guillem Cucurell, Hailey Nguyen, Hannah Korevaar, Hu Xu, Hugo Touvron, Iliyan Zarov, Imanol Arrieta Ibarra, Isabel M. Kloumann, Ishan Misra, Ivan Evtimov, Jade Copet, Jaewon Lee, Jan Geffert, Jana Vranes, Jason Park, Jay Mahadeokar, Jeet Shah, Jelmer van der Linde, Jennifer Billock, Jenny Hong, Jenya Lee, Jeremy Fu, Jianfeng Chi, Jianyu Huang, Jiawen Liu, Jie Wang, Jiecao Yu, Joanna Bitton, Joe Spisak, Jongsoo Park, Joseph Rocca, Joshua Johnstun, Joshua Saxe, Junteng Jia, Kalyan Vasuden Alwala, Kartikeya Upasani, Kate Plawiak, Ke Li, Kenneth Heafield, Kevin Stone, and et al. The llama 3 herd of models. *CoRR*, abs/2407.21783, 2024. doi: 10.48550/ARXIV.2407.21783. URL https://doi.org/10.48550/arXiv.2407.21783.

Omer Dunay, Daniel Cheng, Adam Tait, Parth Thakkar, Peter C. Rigby, Andy Chiu, Imad Ahmad, Arun Ganesan, Chandra Shekhar Maddila, Vijayaraghavan Murali, Ali Tayyebi, and Nachiappan Nagappan. Multi-line ai-assisted code authoring. In Marcelo d'Amorim (ed.), *Companion Proceedings of the 32nd ACM International Conference on the Foundations of Software Engineering, FSE 2024, Porto de Galinhas, Brazil, July 15-19, 2024*, pp. 150–160. ACM, 2024. doi: 10.1145/3663529.3663836. URL https://doi.org/10.1145/3663529.3663836.

Aleksandra Eliseeva, Yaroslav Sokolov, Egor Bogomolov, Yaroslav Golubev, Danny Dig, and Timofey Bryksin. From commit message generation to history-aware commit message completion. In *38th IEEE/ACM International Conference on Automated Software Engineering, ASE 2023, Luxembourg, September 11-15, 2023*, pp. 723–735. IEEE, 2023. doi: 10.1109/ASE56229.2023.00078. URL https://doi.org/10.1109/ASE56229.2023.00078.

Alexander Frömmgen, Jacob Austin, Peter Choy, Nimesh Ghelani, Lera Kharatyan, Gabriela Surita, Elena Khrapko, Pascal Lamblin, Pierre-Antoine Manzagol, Marcus Revaj, Maxim Tabachnyk, Daniel Tarlow, Kevin Villela, Daniel Zheng, Satish Chandra, and Petros Maniatis. Resolving code review comments with machine learning. In *Proceedings of the 46th International Conference on Software Engineering: Software Engineering in Practice, ICSE-SEIP 2024, Lisbon, Portugal, April 14-20, 2024*, pp. 204–215. ACM, 2024. doi: 10.1145/3639477.3639746. URL https://doi.org/10.1145/3639477.3639746.

Samir Yitzhak Gadre, Georgios Smyrnis, Vaishaal Shankar, Suchin Gururangan, Mitchell Wortsman, Rulin Shao, Jean Mercat, Alex Fang, Jeffrey Li, Sedrick Keh, Rui Xin, Marianna Nezhurina, Igor Vasiljevic, Jenia Jitsev, Alexandros G. Dimakis, Gabriel Ilharco, Shuran Song, Thomas Kollar, Yair Carmon, Achal Dave, Reinhard Heckel, Niklas Muennighoff, and Ludwig Schmidt. Language models scale reliably with over-training and on downstream tasks. *CoRR*, abs/2403.08540, 2024. doi: 10.48550/ARXIV.2403.08540. URL https://doi.org/10.48550/arXiv.2403.08540.

Linyuan Gong, Mostafa Elhoushi, and Alvin Cheung. AST-T5: structure-aware pretraining for code generation and understanding. In *ICML*. OpenReview.net, 2024.

Giovani Guizzo, Jie M. Zhang, Federica Sarro, Christoph Treude, and Mark Harman. Mutation analysis for evaluating code translation. *Empir. Softw. Eng.*, 29(1):19, 2024. doi: 10.1007/S10664-023-10385-W. URL https://doi.org/10.1007/s10664-023-10385-w.

Suriya Gunasekar, Yi Zhang, Jyoti Aneja, Caio César Teodoro Mendes, Allie Del Giorno, Sivakanth Gopi, Mojan Javaheripi, Piero Kauffmann, Gustavo de Rosa, Olli Saarikivi, Adil Salim, Shital Shah, Harkirat Singh Behl, Xin Wang, Sébastien Bubeck, Ronen Eldan, Adam Tauman Kalai, Yin Tat Lee, and Yuanzhi Li. Textbooks are all you need. *CoRR*, abs/2306.11644, 2023. doi: 10.48550/ARXIV.2306.11644. URL https://doi.org/10.48550/arXiv.2306.11644.

Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin B. Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. Graphcodebert: Pre-training code representations with data flow. In *9th International Conference on Learning Representations, ICLR 2021, Virtual Event, Austria, May 3-7, 2021*. OpenReview.net, 2021. URL https://openreview.net/forum?id=jLoC4ez43PZ.

Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. Unixcoder: Unified cross-modal pre-training for code representation. In Smaranda Muresan, Preslav Nakov, and Aline Villavicencio (eds.), *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2022, Dublin, Ireland, May 22-27, 2022*, pp. 7212–7225. Association for Computational Linguistics, 2022a. doi: 10.18653/V1/2022.ACL-LONG.499. URL https://doi.org/10.18653/v1/2022.acl-long.499.

Daya Guo, Alexey Svyatkovskiy, Jian Yin, Nan Duan, Marc Brockschmidt, and Miltiadis Allamanis. Learning to complete code with sketches. In *The Tenth International Conference on Learning Representations, ICLR 2022, Virtual Event, April 25-29, 2022*. OpenReview.net, 2022b. URL https://openreview.net/forum?id=q79uMSC6ZBT.

Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. Deepseek-coder: When the large language model meets programming - the rise of code intelligence. *CoRR*, abs/2401.14196, 2024. doi: 10.48550/ARXIV.2401.14196. URL https://doi.org/10.48550/arXiv.2401.14196.

Juncai Guo, Jin Liu, Yao Wan, Li Li, and Pingyi Zhou. Modeling hierarchical syntax structure with triplet position for source code summarization. In Smaranda Muresan, Preslav Nakov, and Aline Villavicencio (eds.), *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2022, Dublin, Ireland, May 22-27, 2022*, pp. 486–500. Association for Computational Linguistics, 2022c. doi: 10.18653/V1/2022.ACL-LONG.37. URL https://doi.org/10.18653/v1/2022.acl-long.37.

Mohammad Abdul Hadi, Imam Nur Bani Yusuf, Ferdian Thung, Kien Gia Luong, Lingxiao Jiang, Fatemeh H. Fard, and David Lo. On the effectiveness of pretrained models for API learning. In Ayushi Rastogi, Rosalia Tufano, Gabriele Bavota, Venera Arnaoudova, and Sonia Haiduc (eds.), *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension, ICPC 2022, Virtual Event, May 16-17, 2022*, pp. 309–320. ACM, 2022. doi: 10.1145/3524610.3527886. URL https://doi.org/10.1145/3524610.3527886.

Kenneth Heafield. Kenlm: Faster and smaller language model queries. In Chris Callison-Burch, Philipp Koehn, Christof Monz, and Omar Zaidan (eds.), *Proceedings of the Sixth Workshop*

on Statistical Machine Translation, WMT@EMNLP 2011, Edinburgh, Scotland, UK, July 30-31, 2011, pp. 187–197. Association for Computational Linguistics, 2011. URL https://aclanthology.org/W11-2123/.

Danny Hernandez, Jared Kaplan, Tom Henighan, and Sam McCandlish. Scaling laws for transfer. *CoRR*, abs/2102.01293, 2021. URL https://arxiv.org/abs/2102.01293.

Abram Hindle, Earl T. Barr, Mark Gabel, Zhendong Su, and Premkumar T. Devanbu. On the naturalness of software. *Commun. ACM*, 59(5):122–131, 2016. doi: 10.1145/2902362. URL https://doi.org/10.1145/2902362.

Jordan Hoffmann, Sebastian Borgeaud, Arthur Mensch, Elena Buchatskaya, Trevor Cai, Eliza Rutherford, Diego de Las Casas, Lisa Anne Hendricks, Johannes Welbl, Aidan Clark, Tom Hennigan, Eric Noland, Katie Millican, George van den Driessche, Bogdan Damoc, Aurelia Guy, Simon Osindero, Karen Simonyan, Erich Elsen, Jack W. Rae, Oriol Vinyals, and Laurent Sifre. Training compute-optimal large language models. *CoRR*, abs/2203.15556, 2022. doi: 10.48550/ARXIV.2203.15556. URL https://doi.org/10.48550/arXiv.2203.15556.

Naman Jain, Tianjun Zhang, Wei-Lin Chiang, Joseph E. Gonzalez, Koushik Sen, and Ion Stoica. Llm-assisted code cleaning for training accurate code generators. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024. URL https://openreview.net/forum?id=maRYffiUpI.

Paras Jain, Ajay Jain, Tianjun Zhang, Pieter Abbeel, Joseph Gonzalez, and Ion Stoica. Contrastive code representation learning. In Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih (eds.), *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, 2021*, pp. 5954–5971. Association for Computational Linguistics, 2021. doi: 10.18653/V1/2021.EMNLP-MAIN.482. URL https://doi.org/10.18653/v1/2021.emnlp-main.482.

Hui Jiang, Linfeng Song, Yubin Ge, Fandong Meng, Junfeng Yao, and Jinsong Su. An AST structure enhanced decoder for code generation. *IEEE ACM Trans. Audio Speech Lang. Process.*, 30: 468–476, 2022a. doi: 10.1109/TASLP.2021.3138717. URL https://doi.org/10.1109/TASLP.2021.3138717.

Yuan Jiang, Xiaohong Su, Christoph Treude, and Tiantian Wang. Hierarchical semantic-aware neural code representation. *J. Syst. Softw.*, 191:111355, 2022b. doi: 10.1016/J.JSS.2022.111355. URL https://doi.org/10.1016/j.jss.2022.111355.

Jared Kaplan, Sam McCandlish, Tom Henighan, Tom B. Brown, Benjamin Chess, Rewon Child, Scott Gray, Alec Radford, Jeffrey Wu, and Dario Amodei. Scaling laws for neural language models. *CoRR*, abs/2001.08361, 2020. URL https://arxiv.org/abs/2001.08361.

Seohyun Kim, Jinman Zhao, Yuchi Tian, and Satish Chandra. Code prediction by feeding trees to transformers. In *43rd IEEE/ACM International Conference on Software Engineering, ICSE 2021, Madrid, Spain, 22-30 May 2021*, pp. 150–162. IEEE, 2021. doi: 10.1109/ICSE43902.2021.00026. URL https://doi.org/10.1109/ICSE43902.2021.00026.

Diederik P. Kingma and Jimmy Ba. Adam: A method for stochastic optimization. In Yoshua Bengio and Yann LeCun (eds.), *3rd International Conference on Learning Representations, ICLR 2015, San Diego, CA, USA, May 7-9, 2015, Conference Track Proceedings*, 2015. URL http://arxiv.org/abs/1412.6980.

Donald E. Knuth. Literate programming. *Comput. J.*, 27(2):97–111, 1984. doi: 10.1093/COMJNL/27.2.97. URL https://doi.org/10.1093/comjnl/27.2.97.

Denis Kocetkov, Raymond Li, Loubna Ben Allal, Jia Li, Chenghao Mou, Yacine Jernite, Margaret Mitchell, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Dzmitry Bahdanau, Leandro von Werra, and Harm de Vries. The stack: 3 TB of permissively licensed source code. *Trans. Mach. Learn. Res.*, 2023, 2023. URL https://openreview.net/forum?id=pxpbTdUEpD.

Woosuk Kwon, Zhuohan Li, Siyuan Zhuang, Ying Sheng, Lianmin Zheng, Cody Hao Yu, Joseph Gonzalez, Hao Zhang, and Ion Stoica. Efficient memory management for large language model serving with pagedattention. In Jason Flinn, Margo I. Seltzer, Peter Druschel, Antoine Kaufmann, and Jonathan Mace (eds.), *Proceedings of the 29th Symposium on Operating Systems Principles, SOSP 2023, Koblenz, Germany, October 23-26, 2023*, pp. 611–626. ACM, 2023. doi: 10.1145/3600006.3613165. URL https://doi.org/10.1145/3600006.3613165.

Marie-Anne Lachaux, Baptiste Rozière, Marc Szafraniec, and Guillaume Lample. DOBF: A deobfuscation pre-training objective for programming languages. In Marc'Aurelio Ranzato, Alina Beygelzimer, Yann N. Dauphin, Percy Liang, and Jennifer Wortman Vaughan (eds.), *Advances in Neural Information Processing Systems 34: Annual Conference on Neural Information Processing Systems 2021, NeurIPS 2021, December 6-14, 2021, virtual*, pp. 14967–14979, 2021. URL https://proceedings.neurips.cc/paper/2021/hash/7d6548bdc0082aacc950ed35e91fcccb-Abstract.html.

Jeremy Lacomis, Pengcheng Yin, Edward J. Schwartz, Miltiadis Allamanis, Claire Le Goues, Graham Neubig, and Bogdan Vasilescu. DIRE: A neural approach to decompiled identifier naming. In *34th IEEE/ACM International Conference on Automated Software Engineering, ASE 2019, San Diego, CA, USA, November 11-15, 2019*, pp. 628–639. IEEE, 2019. doi: 10.1109/ASE.2019.00064. URL https://doi.org/10.1109/ASE.2019.00064.

Dawn J. Lawrie, Christopher Morrell, Henry Feild, and David W. Binkley. What's in a name? A study of identifiers. In *14th International Conference on Program Comprehension (ICPC 2006), 14-16 June 2006, Athens, Greece*, pp. 3–12. IEEE Computer Society, 2006. doi: 10.1109/ICPC.2006.51. URL https://doi.org/10.1109/ICPC.2006.51.

Hung Le, Yue Wang, Akhilesh Deepak Gotmare, Silvio Savarese, and Steven Chu-Hong Hoi. Coderl: Mastering code generation through pretrained models and deep reinforcement learning. In Sanmi Koyejo, S. Mohamed, A. Agarwal, Danielle Belgrave, K. Cho, and A. Oh (eds.), *Advances in Neural Information Processing Systems 35: Annual Conference on Neural Information Processing Systems 2022, NeurIPS 2022, New Orleans, LA, USA, November 28 - December 9, 2022*, 2022. URL http://papers.nips.cc/paper_files/paper/2022/hash/8636419dea1aa9fbd25fc4248e702da4-Abstract-Conference.html.

Katherine Lee, Daphne Ippolito, Andrew Nystrom, Chiyuan Zhang, Douglas Eck, Chris Callison-Burch, and Nicholas Carlini. Deduplicating training data makes language models better. In *ACL (1)*, pp. 8424–8445. Association for Computational Linguistics, 2022.

Jia Li, Ge Li, Yongming Li, and Zhi Jin. Structured chain-of-thought prompting for code generation. *ACM Transactions on Software Engineering and Methodology*, 2023a. URL https://api.semanticscholar.org/CorpusID:258615421.

Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, Qian Liu, Evgenii Zheltonozhskii, Terry Yue Zhuo, Thomas Wang, Olivier Dehaene, Mishig Davaadorj, Joel Lamy-Poirier, João Monteiro, Oleh Shliazhko, Nicolas Gontier, Nicholas Meade, Armel Zebaze, Ming-Ho Yee, Logesh Kumar Umapathi, Jian Zhu, Benjamin Lipkin, Muhtasham Oblokulov, Zhiruo Wang, Rudra Murthy V, Jason T. Stillerman, Siva Sankalp Patel, Dmitry Abulkhanov, Marco Zocca, Manan Dey, Zhihan Zhang, Nour Fahmy, Urvashi Bhattacharyya, Wenhao Yu, Swayam Singh, Sasha Luccioni, Paulo Villegas, Maxim Kunakov, Fedor Zhdanov, Manuel Romero, Tony Lee, Nadav Timor, Jennifer Ding, Claire Schlesinger, Hailey Schoelkopf, Jan Ebert, Tri Dao, Mayank Mishra, Alex Gu, Jennifer Robinson, Carolyn Jane Anderson, Brendan Dolan-Gavitt, Danish Contractor, Siva Reddy, Daniel Fried, Dzmitry Bahdanau, Yacine Jernite, Carlos Muñoz Ferrandis, Sean Hughes, Thomas Wolf, Arjun Guha, Leandro von Werra, and Harm de Vries. Starcoder: may the source be with you! *Trans. Mach. Learn. Res.*, 2023, 2023b. URL https://openreview.net/forum?id=KoFOg41haE.

Yiyang Li, Hongqiu Wu, and Hai Zhao. Semantic-preserving adversarial code comprehension. In Nicoletta Calzolari, Chu-Ren Huang, Hansaem Kim, James Pustejovsky, Leo Wanner, Key-Sun Choi, Pum-Mo Ryu, Hsin-Hsi Chen, Lucia Donatelli, Heng Ji, Sadao Kurohashi, Patrizia Paggio, Nianwen Xue, Seokhwan Kim, Younggyun Hahm, Zhong He, Tony Kyungil Lee, Enrico Santus,

Francis Bond, and Seung-Hoon Na (eds.), *Proceedings of the 29th International Conference on Computational Linguistics, COLING 2022, Gyeongju, Republic of Korea, October 12-17, 2022*, pp. 3017–3028. International Committee on Computational Linguistics, 2022a. URL https://aclanthology.org/2022.coling-1.267.

Yujia Li, David H. Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. Competition-level code generation with alphacode. *CoRR*, abs/2203.07814, 2022b. doi: 10.48550/ARXIV.2203.07814. URL https://doi.org/10.48550/arXiv.2203.07814.

Zongjie Li, Pingchuan Ma, Huaijin Wang, Shuai Wang, Qiyi Tang, Sen Nie, and Shi Wu. Unleashing the power of compiler intermediate representation to enhance neural program embeddings. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*, pp. 2253–2265. ACM, 2022c. doi: 10.1145/3510003.3510217. URL https://doi.org/10.1145/3510003.3510217.

Aixin Liu, Bei Feng, Bin Wang, Bingxuan Wang, Bo Liu, Chenggang Zhao, Chengqi Deng, Chong Ruan, Damai Dai, Daya Guo, Dejian Yang, Deli Chen, Dongjie Ji, Erhang Li, Fangyun Lin, Fuli Luo, Guangbo Hao, Guanting Chen, Guowei Li, Hao Zhang, Hanwei Xu, Hao Yang, Haowei Zhang, Honghui Ding, Huajian Xin, Huazuo Gao, Hui Li, Hui Qu, J. L. Cai, Jian Liang, Jianzhong Guo, Jiaqi Ni, Jiashi Li, Jin Chen, Jingyang Yuan, Junjie Qiu, Junxiao Song, Kai Dong, Kaige Gao, Kang Guan, Lean Wang, Lecong Zhang, Lei Xu, Leyi Xia, Liang Zhao, Liyue Zhang, Meng Li, Miaojun Wang, Mingchuan Zhang, Minghua Zhang, Minghui Tang, Mingming Li, Ning Tian, Panpan Huang, Peiyi Wang, Peng Zhang, Qihao Zhu, Qinyu Chen, Qiushi Du, R. J. Chen, R. L. Jin, Ruiqi Ge, Ruizhe Pan, Runxin Xu, Ruyi Chen, S. S. Li, Shanghao Lu, Shangyan Zhou, Shanhuang Chen, Shaoqing Wu, Shengfeng Ye, Shirong Ma, Shiyu Wang, Shuang Zhou, Shuiping Yu, Shunfeng Zhou, Size Zheng, Tao Wang, Tian Pei, Tian Yuan, Tianyu Sun, W. L. Xiao, Wangding Zeng, Wei An, Wen Liu, Wenfeng Liang, Wenjun Gao, Wentao Zhang, X. Q. Li, Xiangyue Jin, Xianzu Wang, Xiao Bi, Xiaodong Liu, Xiaohan Wang, Xiaojin Shen, Xiaokang Chen, Xiaosha Chen, Xiaotao Nie, and Xiaowen Sun. Deepseek-v2: A strong, economical, and efficient mixture-of-experts language model. *CoRR*, abs/2405.04434, 2024. doi: 10.48550/ARXIV.2405.04434. URL https://doi.org/10.48550/arXiv.2405.04434.

Chenxiao Liu, Shuai Lu, Weizhu Chen, Daxin Jiang, Alexey Svyatkovskiy, Shengyu Fu, Neel Sundaresan, and Nan Duan. Code execution with pre-trained language models. In Anna Rogers, Jordan L. Boyd-Graber, and Naoaki Okazaki (eds.), *Findings of the Association for Computational Linguistics: ACL 2023, Toronto, Canada, July 9-14, 2023*, pp. 4984–4999. Association for Computational Linguistics, 2023a. doi: 10.18653/V1/2023.FINDINGS-ACL.308. URL https://doi.org/10.18653/v1/2023.findings-acl.308.

Jiate Liu, Yiqin Zhu, Kaiwen Xiao, Qiang Fu, Xiao Han, Wei Yang, and Deheng Ye. RLTF: reinforcement learning from unit test feedback. *Trans. Mach. Learn. Res.*, 2023, 2023b. URL https://openreview.net/forum?id=hjYmsV6nXZ.

Shangqing Liu, Xiaofei Xie, Jing Kai Siow, Lei Ma, Guozhu Meng, and Yang Liu. Graphsearchnet: Enhancing gnns via capturing global dependencies for semantic code search. *IEEE Trans. Software Eng.*, 49(4):2839–2855, 2023c. doi: 10.1109/TSE.2022.3233901. URL https://doi.org/10.1109/TSE.2022.3233901.

Anton Lozhkov, Raymond Li, Loubna Ben Allal, Federico Cassano, Joel Lamy-Poirier, Nouamane Tazi, Ao Tang, Dmytro Pykhtar, Jiawei Liu, Yuxiang Wei, Tianyang Liu, Max Tian, Denis Kocetkov, Arthur Zucker, Younes Belkada, Zijian Wang, Qian Liu, Dmitry Abulkhanov, Indraneil Paul, Zhuang Li, Wen-Ding Li, Megan Risdal, Jia Li, Jian Zhu, Terry Yue Zhuo, Evgenii Zheltonozhskii, Nii Osae Osae Dade, Wenhao Yu, Lucas Krauß, Naman Jain, Yixuan Su, Xuanli He, Manan Dey, Edoardo Abati, Yekun Chai, Niklas Muennighoff, Xiangru Tang, Muhtasham Oblokulov, Christopher Akiki, Marc Marone, Chenghao Mou, Mayank Mishra, Alex Gu, Binyuan Hui, Tri Dao, Armel Zebaze, Olivier Dehaene, Nicolas Patry, Canwen Xu, Julian J. McAuley, Han Hu, Torsten Scholak, Sébastien Paquet, Jennifer Robinson, Carolyn Jane Anderson, Nicolas Chapados,

and et al. Starcoder 2 and the stack v2: The next generation. *CoRR*, abs/2402.19173, 2024. doi: 10.48550/ARXIV.2402.19173. URL https://doi.org/10.48550/arXiv.2402.19173.

Shuai Lu, Daya Guo, Shuo Ren, Junjie Huang, Alexey Svyatkovskiy, Ambrosio Blanco, Colin B. 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. Codexglue: A machine learning benchmark dataset for code understanding and generation. In Joaquin Vanschoren and Sai-Kit Yeung (eds.), *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks 2021, December 2021, virtual*, 2021. URL https://datasets-benchmarks-proceedings.neurips.cc/paper/2021/hash/c16a5320fa475530d9583c34fd356ef5-Abstract-round1.html.

Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. Wizardcoder: Empowering code large language models with evol-instruct. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024. URL https://openreview.net/forum?id=UnUwSIgK5W.

Haoyu Ma, Xinjie Ma, Weijie Liu, Zhipeng Huang, Debin Gao, and Chunfu Jia. Control flow obfuscation using neural network to fight concolic testing. In Jing Tian, Jiwu Jing, and Mudhakar Srivatsa (eds.), *International Conference on Security and Privacy in Communication Networks - 10th International ICST Conference, SecureComm 2014, Beijing, China, September 24-26, 2014, Revised Selected Papers, Part I*, volume 152 of *Lecture Notes of the Institute for Computer Sciences, Social Informatics and Telecommunications Engineering*, pp. 287–304. Springer, 2014. doi: 10.1007/978-3-319-23829-6_21. URL https://doi.org/10.1007/978-3-319-23829-6_21.

Wei Ma, Mengjie Zhao, Xiaofei Xie, Qiang Hu, Shangqing Liu, Jiexin Zhang, Wenhan Wang, and Yang Liu. Unveiling code pre-trained models: Investigating syntax and semantics capacities. *ACM Transactions on Software Engineering and Methodology*, 2022. URL https://api.semanticscholar.org/CorpusID:258556996.

Parvez Mahbub, Ohiduzzaman Shuvo, and Mohammad Masudur Rahman. Explaining software bugs leveraging code structures in neural machine translation. In *45th IEEE/ACM International Conference on Software Engineering, ICSE 2023, Melbourne, Australia, May 14-20, 2023*, pp. 640–652. IEEE, 2023. doi: 10.1109/ICSE48619.2023.00063. URL https://doi.org/10.1109/ICSE48619.2023.00063.

Pratyush Maini, Skyler Seto, Richard He Bai, David Grangier, Yizhe Zhang, and Navdeep Jaitly. Rephrasing the web: A recipe for compute and data-efficient language modeling. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar (eds.), *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2024, Bangkok, Thailand, August 11-16, 2024*, pp. 14044–14072. Association for Computational Linguistics, 2024. doi: 10.18653/V1/2024.ACL-LONG.757. URL https://doi.org/10.18653/v1/2024.acl-long.757.

Max Marion, Ahmet Üstün, Luiza Pozzobon, Alex Wang, Marzieh Fadaee, and Sara Hooker. When less is more: Investigating data pruning for pretraining llms at scale. *CoRR*, abs/2309.04564, 2023.

Mayank Mishra, Matt Stallone, Gaoyuan Zhang, Yikang Shen, Aditya Prasad, Adriana Meza Soria, Michele Merler, Parameswaran Selvam, Saptha Surendran, Shivdeep Singh, Manish Sethi, Xuan-Hong Dang, Pengyuan Li, Kun-Lung Wu, Syed Zawad, Andrew Coleman, Matthew White, Mark Lewis, Raju Pavuluri, Yan Koyfman, Boris Lublinsky, Maximilien de Bayser, Ibrahim Abdelaziz, Kinjal Basu, Mayank Agarwal, Yi Zhou, Chris Johnson, Aanchal Goyal, Hima Patel, S. Yousaf Shah, Petros Zerfos, Heiko Ludwig, Asim Munawar, Maxwell Crouse, Pavan Kapanipathi, Shweta Salaria, Bob Calio, Sophia Wen, Seetharami Seelam, Brian Belgodere, Carlos A. Fonseca, Amith Singhee, Nirmit Desai, David D. Cox, Ruchir Puri, and Rameswar Panda. Granite code models: A family of open foundation models for code intelligence. *CoRR*, abs/2405.04324, 2024. doi: 10.48550/ARXIV.2405.04324. URL https://doi.org/10.48550/arXiv.2405.04324.

Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. Convolutional neural networks over tree structures for programming language processing. In Dale Schuurmans and Michael P. Wellman (eds.), *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence, February 12-17, 2016, Phoenix, Arizona, USA*, pp. 1287–1293. AAAI Press, 2016. doi: 10.1609/AAAI.V30I1.10139. URL https://doi.org/10.1609/aaai.v30i1.10139.

Aaron Mueller, Albert Webson, Jackson Petty, and Tal Linzen. In-context learning generalizes, but not always robustly: The case of syntax. In Kevin Duh, Helena Gómez-Adorno, and Steven Bethard (eds.), *Proceedings of the 2024 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers), NAACL 2024, Mexico City, Mexico, June 16-21, 2024*, pp. 4761–4779. Association for Computational Linguistics, 2024. doi: 10.18653/V1/2024.NAACL-LONG.267. URL https://doi.org/10.18653/v1/2024.naacl-long.267.

Niklas Muennighoff, Alexander M. Rush, Boaz Barak, Teven Le Scao, Nouamane Tazi, Aleksandra Piktus, Sampo Pyysalo, Thomas Wolf, and Colin A. Raffel. Scaling data-constrained language models. In Alice Oh, Tristan Naumann, Amir Globerson, Kate Saenko, Moritz Hardt, and Sergey Levine (eds.), *Advances in Neural Information Processing Systems 36: Annual Conference on Neural Information Processing Systems 2023, NeurIPS 2023, New Orleans, LA, USA, December 10 - 16, 2023*, 2023. URL http://papers.nips.cc/paper_files/paper/2023/hash/9d89448b63ce1e2e8dc7af72c984c196-Abstract-Conference.html.

Vijayaraghavan Murali, Chandra Shekhar Maddila, Imad Ahmad, Michael Bolin, Daniel Cheng, Negar Ghorbani, Renuka Fernandez, Nachiappan Nagappan, and Peter C. Rigby. Ai-assisted code authoring at scale: Fine-tuning, deploying, and mixed methods evaluation. *Proc. ACM Softw. Eng.*, 1(FSE):1066–1085, 2024. doi: 10.1145/3643774. URL https://doi.org/10.1145/3643774.

Shounak Naik, Rajaswa Patil, Swati Agarwal, and Veeky Baths. Probing semantic grounding in language models of code with representational similarity analysis. In Weitong Chen, Lina Yao, Taotao Cai, Shirui Pan, Tao Shen, and Xue Li (eds.), *Advanced Data Mining and Applications - 18th International Conference, ADMA 2022, Brisbane, QLD, Australia, November 28-30, 2022, Proceedings, Part II*, volume 13726 of *Lecture Notes in Computer Science*, pp. 395–406. Springer, 2022. doi: 10.1007/978-3-031-22137-8_29. URL https://doi.org/10.1007/978-3-031-22137-8_29.

Aravind Ashok Nair, Avijit Roy, and Karl Meinke. funcgnn: A graph neural network approach to program similarity. In Maria Teresa Baldassarre, Filippo Lanubile, Marcos Kalinowski, and Federica Sarro (eds.), *ESEM '20: ACM / IEEE International Symposium on Empirical Software Engineering and Measurement, Bari, Italy, October 5-7, 2020*, pp. 10:1–10:11. ACM, 2020. doi: 10.1145/3382494.3410675. URL https://doi.org/10.1145/3382494.3410675.

Yusuke Oda, Hiroyuki Fudaba, Graham Neubig, Hideaki Hata, Sakriani Sakti, Tomoki Toda, and Satoshi Nakamura. Learning to generate pseudo-code from source code using statistical machine translation (T). In Myra B. Cohen, Lars Grunske, and Michael Whalen (eds.), *30th IEEE/ACM International Conference on Automated Software Engineering, ASE 2015, Lincoln, NE, USA, November 9-13, 2015*, pp. 574–584. IEEE Computer Society, 2015. doi: 10.1109/ASE.2015.36. URL https://doi.org/10.1109/ASE.2015.36.

David N. Palacio, Alejandro Velasco, Daniel Rodríguez-Cárdenas, Kevin Moran, and Denys Poshyvanyk. Evaluating and explaining large language models for code using syntactic structures. *CoRR*, abs/2308.03873, 2023. doi: 10.48550/ARXIV.2308.03873. URL https://doi.org/10.48550/arXiv.2308.03873.

Rangeet Pan, Ali Reza Ibrahimzada, Rahul Krishna, Divya Sankar, Lambert Pouguem Wassi, Michele Merler, Boris Sobolev, Raju Pavuluri, Saurabh Sinha, and Reyhaneh Jabbarvand. Lost in translation: A study of bugs introduced by large language models while translating code. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering, ICSE 2024, Lisbon, Portugal, April 14-20, 2024*, pp. 82:1–82:13. ACM, 2024. doi: 10.1145/3597503.3639226. URL https://doi.org/10.1145/3597503.3639226.

Youngmi Park, Ahjeong Park, and Chulyun Kim. Alsi-transformer: Transformer-based code comment generation with aligned lexical and syntactic information. *IEEE Access*, 11:39037–39047, 2023. doi: 10.1109/ACCESS.2023.3268638. URL https://doi.org/10.1109/ACCESS.2023.3268638.

Julian Parsert and Elizabeth Polgreen. Reinforcement learning and data-generation for syntax-guided synthesis. In Michael J. Wooldridge, Jennifer G. Dy, and Sriraam Natarajan (eds.), *Thirty-Eighth AAAI Conference on Artificial Intelligence, AAAI 2024, Thirty-Sixth Conference on Innovative Applications of Artificial Intelligence, IAAI 2024, Fourteenth Symposium on Educational Advances in Artificial Intelligence, EAAI 2014, February 20-27, 2024, Vancouver, Canada*, pp. 10670–10678. AAAI Press, 2024. doi: 10.1609/AAAI.V38I9.28938. URL https://doi.org/10.1609/aaai.v38i9.28938.

Indraneil Paul, Goran Glavas, and Iryna Gurevych. Ircoder: Intermediate representations make language models robust multilingual code generators. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar (eds.), *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2024, Bangkok, Thailand, August 11-16, 2024*, pp. 15023–15041. Association for Computational Linguistics, 2024. URL https://aclanthology.org/2024.acl-long.802.

Kexin Pei, Weichen Li, Qirui Jin, Shuyang Liu, Scott Geng, Lorenzo Cavallaro, Junfeng Yang, and Suman Jana. Exploiting code symmetries for learning program semantics. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*. OpenReview.net, 2024. URL https://openreview.net/forum?id=OLvgrLtv6J.

Guilherme Penedo, Quentin Malartic, Daniel Hesslow, Ruxandra Cojocaru, Hamza Alobeidli, Alessandro Cappelli, Baptiste Pannier, Ebtesam Almazrouei, and Julien Launay. The refinedweb dataset for falcon LLM: outperforming curated corpora with web data only. In *NeurIPS*, 2023.

Guilherme Penedo, Hynek Kydlícek, Loubna Ben Allal, Anton Lozhkov, Margaret Mitchell, Colin Raffel, Leandro von Werra, and Thomas Wolf. The fineweb datasets: Decanting the web for the finest text data at scale. *CoRR*, abs/2406.17557, 2024. doi: 10.48550/ARXIV.2406.17557. URL https://doi.org/10.48550/arXiv.2406.17557.

Constantin Cezar Petrescu, Sam Smith, Rafail Giavrimis, and Santanu Kumar Dash. Do names echo semantics? A large-scale study of identifiers used in c++'s named casts. *J. Syst. Softw.*, 202:111693, 2023. doi: 10.1016/J.JSS.2023.111693. URL https://doi.org/10.1016/j.jss.2023.111693.

Hung Phan and Ali Jannesari. Evaluating and optimizing the effectiveness of neural machine translation in supporting code retrieval models: A study on the CAT benchmark. In Ingo Frommholz, Frank Hopfgartner, Mark Lee, Michael Oakes, Mounia Lalmas, Min Zhang, and Rodrygo L. T. Santos (eds.), *Proceedings of the 32nd ACM International Conference on Information and Knowledge Management, CIKM 2023, Birmingham, United Kingdom, October 21-25, 2023*, pp. 2055–2064. ACM, 2023. doi: 10.1145/3583780.3614869. URL https://doi.org/10.1145/3583780.3614869.

Clifton Poth, Hannah Sterz, Indraneil Paul, Sukannya Purkayastha, Leon Engländer, Timo Imhof, Ivan Vulic, Sebastian Ruder, Iryna Gurevych, and Jonas Pfeiffer. Adapters: A unified library for parameter-efficient and modular transfer learning. In Yansong Feng and Els Lefever (eds.), *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing, EMNLP 2023 - System Demonstrations, Singapore, December 6-10, 2023*, pp. 149–160. Association for Computational Linguistics, 2023. doi: 10.18653/V1/2023.EMNLP-DEMO.13. URL https://doi.org/10.18653/v1/2023.emnlp-demo.13.

Raghu Prabhakar, Ram Sivaramakrishnan, Darshan Gandhi, Yun Du, Mingran Wang, Xiangyu Song, Kejie Zhang, Tianren Gao, Angela Wang, Karen Li, Yongning Sheng, Joshua Brot, Denis Sokolov, Apurv Vivek, Calvin Leung, Arjun Sabnis, Jiayu Bai, Tuowen Zhao, Mark Gottscho, David Jackson, Mark Luttrell, Manish K. Shah, Edison Chen, Kaizhao Liang, Swayambhoo Jain, Urmish Thakker, Dawei Huang, Sumti Jairath, Kevin J. Brown, and Kunle Olukotun. Sambanova SN40L: scaling the AI memory wall with dataflow and composition of experts. *CoRR*, abs/2405.07518,

2024. doi: 10.48550/ARXIV.2405.07518. URL https://doi.org/10.48550/arXiv.2405.07518.

Ruchir Puri, David S. Kung, Geert Janssen, Wei Zhang, Giacomo Domeniconi, Vladimir Zolotov, Julian Dolby, Jie Chen, Mihir R. Choudhury, Lindsey Decker, Veronika Thost, Luca Buratti, Saurabh Pujar, Shyam Ramji, Ulrich Finkler, Susan Malaika, and Frederick Reiss. Codenet: A large-scale AI for code dataset for learning a diversity of coding tasks. In Joaquin Vanschoren and Sai-Kit Yeung (eds.), *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks 2021, December 2021, virtual*, 2021. URL https://datasets-benchmarks-proceedings.neurips.cc/paper/2021/hash/a5bfc9e07964f8dddeb95fc584cd965d-Abstract-round2.html.

Md. Rafiqul Islam Rabin, Nghi D. Q. Bui, Ke Wang, Yijun Yu, Lingxiao Jiang, and Mohammad Amin Alipour. On the generalizability of neural program models with respect to semantic-preserving program transformations. *Inf. Softw. Technol.*, 135:106552, 2021. doi: 10.1016/J.INFSOF.2021.106552. URL https://doi.org/10.1016/j.infsof.2021.106552.

Md. Rafiqul Islam Rabin, Aftab Hussain, Mohammad Amin Alipour, and Vincent J. Hellendoorn. Memorization and generalization in neural code intelligence models. *Inf. Softw. Technol.*, 153: 107066, 2023. doi: 10.1016/J.INFSOF.2022.107066. URL https://doi.org/10.1016/j.infsof.2022.107066.

Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *J. Mach. Learn. Res.*, 21:140:1–140:67, 2020. URL https://jmlr.org/papers/v21/20-074.html.

Musfiqur Rahman, Dharani Palani, and Peter C. Rigby. Natural software revisited. In Joanne M. Atlee, Tevfik Bultan, and Jon Whittle (eds.), *Proceedings of the 41st International Conference on Software Engineering, ICSE 2019, Montreal, QC, Canada, May 25-31, 2019*, pp. 37–48. IEEE / ACM, 2019. doi: 10.1109/ICSE.2019.00022. URL https://doi.org/10.1109/ICSE.2019.00022.

Samyam Rajbhandari, Jeff Rasley, Olatunji Ruwase, and Yuxiong He. Zero: memory optimizations toward training trillion parameter models. In Christine Cuicchi, Irene Qualters, and William T. Kramer (eds.), *Proceedings of the International Conference for High Performance Computing, Networking, Storage and Analysis, SC 2020, Virtual Event / Atlanta, Georgia, USA, November 9-19, 2020*, pp. 20. IEEE/ACM, 2020. doi: 10.1109/SC41405.2020.00024. URL https://doi.org/10.1109/SC41405.2020.00024.

Baishakhi Ray, Vincent J. Hellendoorn, Saheel Godhane, Zhaopeng Tu, Alberto Bacchelli, and Premkumar T. Devanbu. On the "naturalness" of buggy code. In Laura K. Dillon, Willem Visser, and Laurie A. Williams (eds.), *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, pp. 428–439. ACM, 2016. doi: 10.1145/2884781.2884848. URL https://doi.org/10.1145/2884781.2884848.

Veselin Raychev, Martin T. Vechev, and Andreas Krause. Predicting program properties from 'big code'. *Commun. ACM*, 62(3):99–107, 2019. doi: 10.1145/3306204. URL https://doi.org/10.1145/3306204.

Houxing Ren, Mingjie Zhan, Zhongyuan Wu, Aojun Zhou, Junting Pan, and Hongsheng Li. Reflectioncoder: Learning from reflection sequence for enhanced one-off code generation. *CoRR*, abs/2405.17057, 2024. doi: 10.48550/ARXIV.2405.17057. URL https://doi.org/10.48550/arXiv.2405.17057.

Baptiste Rozière, Marie-Anne Lachaux, Lowik Chanussot, and Guillaume Lample. Unsupervised translation of programming languages. In Hugo Larochelle, Marc'Aurelio Ranzato, Raia Hadsell, Maria-Florina Balcan, and Hsuan-Tien Lin (eds.), *Advances in Neural Information Processing Systems 33: Annual Conference on Neural Information Processing Systems 2020, NeurIPS 2020, December 6-12, 2020, virtual*, 2020. URL https://proceedings.neurips.cc/paper/2020/hash/ed23fbf18c2cd35f8c7f8de44f85c08d-Abstract.html.

Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton-Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. Code llama: Open foundation models for code. *CoRR*, abs/2308.12950, 2023. doi: 10.48550/ARXIV.2308.12950. URL https://doi.org/10.48550/arXiv.2308.12950.

Iman Saberi and Fatemeh H. Fard. Model-agnostic syntactical information for pre-trained programming language models. In *20th IEEE/ACM International Conference on Mining Software Repositories, MSR 2023, Melbourne, Australia, May 15-16, 2023*, pp. 183–193. IEEE, 2023. doi: 10.1109/MSR59073.2023.00036. URL https://doi.org/10.1109/MSR59073.2023.00036.

Nikhil Sardana, Jacob Portes, Sasha Doubov, and Jonathan Frankle. Beyond chinchilla-optimal: Accounting for inference in language model scaling laws. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*. OpenReview.net, 2024. URL https://openreview.net/forum?id=0bmXrtTDUu.

Sebastian Schrittwieser, Stefan Katzenbeisser, Johannes Kinder, Georg Merzdovnik, and Edgar R. Weippl. Protecting software through obfuscation: Can it keep pace with progress in code analysis? *ACM Comput. Surv.*, 49(1):4:1–4:37, 2016. doi: 10.1145/2886012. URL https://doi.org/10.1145/2886012.

Jay Shah, Ganesh Bikshandi, Ying Zhang, Vijay Thakkar, Pradeep Ramani, and Tri Dao. Flashattention-3: Fast and accurate attention with asynchrony and low-precision. *CoRR*, abs/2407.08608, 2024. doi: 10.48550/ARXIV.2407.08608. URL https://doi.org/10.48550/arXiv.2407.08608.

Yiheng Shen, Xiaolin Ju, Xiang Chen, and Guang Yang. Bash comment generation via data augmentation and semantic-aware codebert. *Autom. Softw. Eng.*, 31(1):30, 2024. doi: 10.1007/S10515-024-00431-2. URL https://doi.org/10.1007/s10515-024-00431-2.

Chaochen Shi, Borui Cai, Yao Zhao, Longxiang Gao, Keshav Sood, and Yong Xiang. Coss: Leveraging statement semantics for code summarization. *IEEE Trans. Software Eng.*, 49(6):3472–3486, 2023. doi: 10.1109/TSE.2023.3256362. URL https://doi.org/10.1109/TSE.2023.3256362.

Kensen Shi, Deniz Altinbüken, Saswat Anand, Mihai Christodorescu, Katja Grünwedel, Alexa Koenings, Sai Naidu, Anurag Pathak, Marc Rasi, Fredde Ribeiro, Brandon Ruffin, Siddhant Sanyam, Maxim Tabachnyk, Sara Toth, Roy Tu, Tobias Welp, Pengcheng Yin, Manzil Zaheer, Satish Chandra, and Charles Sutton. Natural language outlines for code: Literate programming in the LLM era. *CoRR*, abs/2408.04820, 2024. doi: 10.48550/ARXIV.2408.04820. URL https://doi.org/10.48550/arXiv.2408.04820.

Yucen Shi, Ying Yin, Zhengkui Wang, David Lo, Tao Zhang, Xin Xia, Yuhai Zhao, and Bowen Xu. How to better utilize code graphs in semantic code search? In Abhik Roychoudhury, Cristian Cadar, and Miryung Kim (eds.), *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, ESEC/FSE 2022, Singapore, Singapore, November 14-18, 2022*, pp. 722–733. ACM, 2022. doi: 10.1145/3540250.3549087. URL https://doi.org/10.1145/3540250.3549087.

Mohammad Shoeybi, Mostofa Patwary, Raul Puri, Patrick LeGresley, Jared Casper, and Bryan Catanzaro. Megatron-lm: Training multi-billion parameter language models using model parallelism. *CoRR*, abs/1909.08053, 2019. URL http://arxiv.org/abs/1909.08053.

Parshin Shojaee, Aneesh Jain, Sindhu Tipirneni, and Chandan K. Reddy. Execution-based code generation using deep reinforcement learning. *Trans. Mach. Learn. Res.*, 2023, 2023. URL https://openreview.net/forum?id=0XBuaxqEcG.

André Silva, João F. Ferreira, He Ye, and Martin Monperrus. MUFIN: improving neural repair models with back-translation. *CoRR*, abs/2304.02301, 2023. doi: 10.48550/ARXIV.2304.02301. URL https://doi.org/10.48550/arXiv.2304.02301.

Philippe Skolka, Cristian-Alexandru Staicu, and Michael Pradel. Anything to hide? studying minified and obfuscated code in the web. In Ling Liu, Ryen W. White, Amin Mantrach, Fabrizio Silvestri, Julian J. McAuley, Ricardo Baeza-Yates, and Leila Zia (eds.), *The World Wide Web Conference, WWW 2019, San Francisco, CA, USA, May 13-17, 2019*, pp. 1735–1746. ACM, 2019. doi: 10.1145/3308558.3313752. URL https://doi.org/10.1145/3308558.3313752.

Tao Sun, Linzheng Chai, Jian Yang, Yuwei Yin, Hongcheng Guo, Jiaheng Liu, Bing Wang, Liqun Yang, and Zhoujun Li. Unicoder: Scaling code large language model via universal code. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar (eds.), *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2024, Bangkok, Thailand, August 11-16, 2024*, pp. 1812–1824. Association for Computational Linguistics, 2024. URL https://aclanthology.org/2024.acl-long.100.

Marc Szafraniec, Baptiste Rozière, Hugh Leather, Patrick Labatut, François Charton, and Gabriel Synnaeve. Code translation with compiler representations. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net, 2023. URL https://openreview.net/forum?id=XomEU3eNeSQ.

Armstrong A. Takang, Penny A. Grubb, and Robert D. Macredie. The effects of comments and identifier names on program comprehensibility: an experimental investigation. *J. Program. Lang.*, 4 (3):143–167, 1996. URL http://compscinet.dcs.kcl.ac.uk/JP/jp040302.abs.html.

Wannita Takerngsaksiri, Chakkrit Tantithamthavorn, and Yuan-Fang Li. Syntax-aware on-the-fly code completion. *Inf. Softw. Technol.*, 165:107336, 2024. doi: 10.1016/J.INFSOF.2023.107336. URL https://doi.org/10.1016/j.infsof.2023.107336.

Tianhua Tao, Junbo Li, Bowen Tan, Hongyi Wang, William Marshall, Bhargav M Kanakiya, Joel Hestness, Natalia Vassilieva, Zhiqiang Shen, Eric P. Xing, and Zhengzhong Liu. Crystal: Illuminating LLM abilities on language and code. In *First Conference on Language Modeling*, 2024. URL https://openreview.net/forum?id=kWnlCVcp6o.

Suresh Thummalapenta and Tao Xie. Alattin: mining alternative patterns for defect detection. *Autom. Softw. Eng.*, 18(3-4):293–323, 2011. doi: 10.1007/S10515-011-0086-Z. URL https://doi.org/10.1007/s10515-011-0086-z.

Sindhu Tipirneni, Ming Zhu, and Chandan K. Reddy. Structcoder: Structure-aware transformer for code generation. *ACM Trans. Knowl. Discov. Data*, 18(3):70:1–70:20, 2024. doi: 10.1145/3636430. URL https://doi.org/10.1145/3636430.

Sergey Troshin and Nadezhda Chirkova. Probing pretrained models of source codes. In Jasmijn Bastings, Yonatan Belinkov, Yanai Elazar, Dieuwke Hupkes, Naomi Saphra, and Sarah Wiegreffe (eds.), *Proceedings of the Fifth BlackboxNLP Workshop on Analyzing and Interpreting Neural Networks for NLP, BlackboxNLP@EMNLP 2022, Abu Dhabi, United Arab Emirates (Hybrid), December 8, 2022*, pp. 371–383. Association for Computational Linguistics, 2022. doi: 10.18653/V1/2022.BLACKBOXNLP-1.31. URL https://doi.org/10.18653/v1/2022.blackboxnlp-1.31.

Alejandro Velasco, David N. Palacio, Daniel Rodríguez-Cárdenas, and Denys Poshyvanyk. Which syntactic capabilities are statistically learned by masked language models for code? In *Proceedings of the 2024 ACM/IEEE 44th International Conference on Software Engineering: New Ideas and Emerging Results, NIER@ICSE 2024, Lisbon, Portugal, April 14-20, 2024*, pp. 72–76. ACM, 2024. doi: 10.1145/3639476.3639768. URL https://doi.org/10.1145/3639476.3639768.

Siddhant Waghjale, Vishruth Veerendranath, Zora Zhiruo Wang, and Daniel Fried. ECCO: can we improve model-generated code efficiency without sacrificing functional correctness? *CoRR*, abs/2407.14044, 2024. doi: 10.48550/ARXIV.2407.14044. URL https://doi.org/10.48550/arXiv.2407.14044.

Changhan Wang, Kyunghyun Cho, and Jiatao Gu. Neural machine translation with byte-level subwords. In *The Thirty-Fourth AAAI Conference on Artificial Intelligence, AAAI 2020, The Thirty-Second Innovative Applications of Artificial Intelligence Conference, IAAI 2020, The Tenth*

AAAI Symposium on Educational Advances in Artificial Intelligence, EAAI 2020, New York, NY, USA, February 7-12, 2020, pp. 9154–9160. AAAI Press, 2020. doi: 10.1609/AAAI.V34I05.6451. URL https://doi.org/10.1609/aaai.v34i05.6451.

Deze Wang, Zhouyang Jia, Shanshan Li, Yue Yu, Yun Xiong, Wei Dong, and Xiangke Liao. Bridging pre-trained models and downstream tasks for source code understanding. In *44th IEEE/ACM 44th International Conference on Software Engineering, ICSE 2022, Pittsburgh, PA, USA, May 25-27, 2022*, pp. 287–298. ACM, 2022. doi: 10.1145/3510003.3510062. URL https://doi.org/10.1145/3510003.3510062.

Evan Wang, Federico Cassano, Catherine Wu, Yunfeng Bai, Will Song, Vaskar Nath, Ziwen Han, Sean Hendryx, Summer Yue, and Hugh Zhang. Planning in natural language improves llm search for code generation. *CoRR*, abs/2409.03733, 2024a. URL https://api.semanticscholar.org/CorpusID:272423808.

Huaijin Wang, Pingchuan Ma, Shuai Wang, Qiyi Tang, Sen Nie, and Shi Wu. sem2vec: Semantics-aware assembly tracelet embedding. *ACM Trans. Softw. Eng. Methodol.*, 32(4):90:1–90:34, 2023a. doi: 10.1145/3569933. URL https://doi.org/10.1145/3569933.

Ke Wang. Learning scalable and precise representation of program semantics. *CoRR*, abs/1905.05251, 2019. URL http://arxiv.org/abs/1905.05251.

Ke Wang and Zhendong Su. Learning blended, precise semantic program embeddings. *CoRR*, abs/1907.02136, 2019. URL http://arxiv.org/abs/1907.02136.

Shiqi Wang, Zheng Li, Haifeng Qian, Chenghao Yang, Zijian Wang, Mingyue Shang, Varun Kumar, Samson Tan, Baishakhi Ray, Parminder Bhatia, Ramesh Nallapati, Murali Krishna Ramanathan, Dan Roth, and Bing Xiang. Recode: Robustness evaluation of code generation models. In Anna Rogers, Jordan L. Boyd-Graber, and Naoaki Okazaki (eds.), *Proceedings of the 61st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), ACL 2023, Toronto, Canada, July 9-14, 2023*, pp. 13818–13843. Association for Computational Linguistics, 2023b. doi: 10.18653/V1/2023.ACL-LONG.773. URL https://doi.org/10.18653/v1/2023.acl-long.773.

Song Wang, Taiyue Liu, and Lin Tan. Automatically learning semantic features for defect prediction. In Laura K. Dillon, Willem Visser, and Laurie A. Williams (eds.), *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, pp. 297–308. ACM, 2016. doi: 10.1145/2884781.2884804. URL https://doi.org/10.1145/2884781.2884804.

Xin Wang, Yasheng Wang, Fei Mi, Pingyi Zhou, Yao Wan, Xiao Liu, Li Li, Hao Wu, Jin Liu, and Xin Jiang. Syncobert: Syntax-guided multi-modal contrastive pre-training for code representation. *CoRR*, abs/2108.04556, 2021a. URL https://api.semanticscholar.org/CorpusID:237450712.

Yue Wang, Weishi Wang, Shafiq R. Joty, and Steven C. H. Hoi. Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In Marie-Francine Moens, Xuanjing Huang, Lucia Specia, and Scott Wen-tau Yih (eds.), *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021, Virtual Event / Punta Cana, Dominican Republic, 7-11 November, 2021*, pp. 8696–8708. Association for Computational Linguistics, 2021b. doi: 10.18653/V1/2021.EMNLP-MAIN.685. URL https://doi.org/10.18653/v1/2021.emnlp-main.685.

Zhijie Wang, Zijie Zhou, Da Song, Yuheng Huang, Shengmai Chen, Lei Ma, and Tianyi Zhang. Where do large language models fail when generating code? *CoRR*, abs/2406.08731, 2024b. doi: 10.48550/ARXIV.2406.08731. URL https://doi.org/10.48550/arXiv.2406.08731.

Bolin Wei, Ge Li, Xin Xia, Zhiyi Fu, and Zhi Jin. Code generation as a dual task of code summarization. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett (eds.), *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pp.

6559–6569, 2019. URL https://proceedings.neurips.cc/paper/2019/hash/e52ad5c9f751f599492b4f087ed7ecfc-Abstract.html.

Yuxiang Wei, Zhe Wang, Jiawei Liu, Yifeng Ding, and Lingming Zhang. Magicoder: Empowering code generation with oss-instruct. In *Forty-first International Conference on Machine Learning, ICML 2024, Vienna, Austria, July 21-27, 2024*. OpenReview.net, 2024. URL https://openreview.net/forum?id=XUeoOBid3x.

Yuhui Xu, Lingxi Xie, Xiaotao Gu, Xin Chen, Heng Chang, Hengheng Zhang, Zhengsu Chen, Xiaopeng Zhang, and Qi Tian. Qa-lora: Quantization-aware low-rank adaptation of large language models. In *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*. OpenReview.net, 2024. URL https://openreview.net/forum?id=WvFoJccpo8.

Adam Yala, Victor Quach, Homa Esfahanizadeh, Rafael G. L. D'Oliveira, Ken R. Duffy, Muriel Médard, Tommi S. Jaakkola, and Regina Barzilay. Syfer: Neural obfuscation for private data release. *CoRR*, abs/2201.12406, 2022. URL https://arxiv.org/abs/2201.12406.

Fabian Yamaguchi, Nico Golde, Daniel Arp, and Konrad Rieck. Modeling and discovering vulnerabilities with code property graphs. In *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014*, pp. 590–604. IEEE Computer Society, 2014. doi: 10.1109/SP.2014.44. URL https://doi.org/10.1109/SP.2014.44.

An Yang, Baosong Yang, Binyuan Hui, Bo Zheng, Bowen Yu, Chang Zhou, Chengpeng Li, Chengyuan Li, Dayiheng Liu, Fei Huang, Guanting Dong, Haoran Wei, Huan Lin, Jialong Tang, Jialin Wang, Jian Yang, Jianhong Tu, Jianwei Zhang, Jianxin Ma, Jianxin Yang, Jin Xu, Jingren Zhou, Jinze Bai, Jinzheng He, Junyang Lin, Kai Dang, Keming Lu, Keqin Chen, Kexin Yang, Mei Li, Mingfeng Xue, Na Ni, Pei Zhang, Peng Wang, Ru Peng, Rui Men, Ruize Gao, Runji Lin, Shijie Wang, Shuai Bai, Sinan Tan, Tianhang Zhu, Tianhao Li, Tianyu Liu, Wenbin Ge, Xiaodong Deng, Xiaohuan Zhou, Xingzhang Ren, Xinyu Zhang, Xipin Wei, Xuancheng Ren, Xuejing Liu, Yang Fan, Yang Yao, Yichang Zhang, Yu Wan, Yunfei Chu, Yuqiong Liu, Zeyu Cui, Zhenru Zhang, Zhifang Guo, and Zhihao Fan. Qwen2 technical report. *CoRR*, abs/2407.10671, 2024. doi: 10.48550/ARXIV.2407.10671. URL https://doi.org/10.48550/arXiv.2407.10671.

Guang Yang, Yu Zhou, Xiang Chen, Xiangyu Zhang, Yiran Xu, Tingting Han, and Taolue Chen. A syntax-guided multi-task learning approach for turducken-style code generation. *Empir. Softw. Eng.*, 28(6):141, 2023a. doi: 10.1007/S10664-023-10372-1. URL https://doi.org/10.1007/s10664-023-10372-1.

Guang Yang, Yu Zhou, Xiangyu Zhang, Xiang Chen, Tingting Han, and Taolue Chen. Assessing and improving syntactic adversarial robustness of pre-trained models for code translation. *CoRR*, abs/2310.18587, 2023b. doi: 10.48550/ARXIV.2310.18587. URL https://doi.org/10.48550/arXiv.2310.18587.

Jia Yang, Cai Fu, Fengyang Deng, Ming Wen, Xiaowei Guo, and Chuanhao Wan. Toward interpretable graph tensor convolution neural network for code semantics embedding. *ACM Trans. Softw. Eng. Methodol.*, 32(5):115:1–115:40, 2023c. doi: 10.1145/3582574. URL https://doi.org/10.1145/3582574.

Weizhe Yuan and Pengfei Liu. restructured pre-training. *CoRR*, abs/2206.11147, 2022. doi: 10.48550/ARXIV.2206.11147. URL https://doi.org/10.48550/arXiv.2206.11147.

Daoguang Zan, Bei Chen, Dejian Yang, Zeqi Lin, Minsu Kim, Bei Guan, Yongji Wang, Weizhu Chen, and Jian-Guang Lou. CERT: continual pre-training on sketches for library-oriented code generation. In Luc De Raedt (ed.), *Proceedings of the Thirty-First International Joint Conference on Artificial Intelligence, IJCAI 2022, Vienna, Austria, 23-29 July 2022*, pp. 2369–2375. ijcai.org, 2022. doi: 10.24963/IJCAI.2022/329. URL https://doi.org/10.24963/ijcai.2022/329.

Jingfeng Zhang, Haiwen Hong, Yin Zhang, Yao Wan, Ye Liu, and Yulei Sui. Disentangled code representation learning for multiple programming languages. In Chengqing Zong, Fei Xia, Wenjie Li, and Roberto Navigli (eds.), *Findings of the Association for Computational*

Linguistics: ACL/IJCNLP 2021, Online Event, August 1-6, 2021, volume ACL/IJCNLP 2021 of *Findings of ACL*, pp. 4454–4466. Association for Computational Linguistics, 2021a. doi: 10.18653/V1/2021.FINDINGS-ACL.391. URL https://doi.org/10.18653/v1/2021.findings-acl.391.

Kechi Zhang, Wenhan Wang, Huangzhao Zhang, Ge Li, and Zhi Jin. Learning to represent programs with heterogeneous graphs. In Ayushi Rastogi, Rosalia Tufano, Gabriele Bavota, Venera Arnaoudova, and Sonia Haiduc (eds.), *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension, ICPC 2022, Virtual Event, May 16-17, 2022*, pp. 378–389. ACM, 2022. doi: 10.1145/3524610.3527905. URL https://doi.org/10.1145/3524610.3527905.

Xiaolu Zhang, Frank Breitinger, Engelbert Luechinger, and Stephen O'Shaughnessy. Android application forensics: A survey of obfuscation, obfuscation detection and deobfuscation techniques and their impact on investigations. *Digit. Investig.*, 39:301285, 2021b. doi: 10.1016/J.FSIDI.2021.301285. URL https://doi.org/10.1016/j.fsidi.2021.301285.

Heri Zhao, Jeffrey Hui, Joshua Howland, Nam Nguyen, Siqi Zuo, Andrea Hu, Christopher A. Choquette-Choo, Jingyue Shen, Joe Kelley, Kshitij Bansal, Luke Vilnis, Mateo Wirth, Paul Michel, Peter Choy, Pratik Joshi, Ravin Kumar, Sarmad Hashmi, Shubham Agrawal, Zhitao Gong, Jane Fine, Tris Warkentin, Ale Jakse Hartman, Bin Ni, Kathy Korevec, Kelly Schaefer, and Scott Huffman. Codegemma: Open code models based on gemma. *CoRR*, abs/2406.11409, 2024. doi: 10.48550/ARXIV.2406.11409. URL https://doi.org/10.48550/arXiv.2406.11409.

Tianyu Zheng, Ge Zhang, Tianhao Shen, Xueling Liu, Bill Yuchen Lin, Jie Fu, Wenhu Chen, and Xiang Yue. Opencodeinterpreter: Integrating code generation with execution and refinement. In Lun-Wei Ku, Andre Martins, and Vivek Srikumar (eds.), *Findings of the Association for Computational Linguistics, ACL 2024, Bangkok, Thailand and virtual meeting, August 11-16, 2024*, pp. 12834–12859. Association for Computational Linguistics, 2024. URL https://aclanthology.org/2024.findings-acl.762.

Tong Zhou, Shaolei Ren, and Xiaolin Xu. Obfunas: A neural architecture search-based DNN obfuscation approach. In Tulika Mitra, Evangeline F. Y. Young, and Jinjun Xiong (eds.), *Proceedings of the 41st IEEE/ACM International Conference on Computer-Aided Design, ICCAD 2022, San Diego, California, USA, 30 October 2022 - 3 November 2022*, pp. 81:1–81:9. ACM, 2022. doi: 10.1145/3508352.3549429. URL https://doi.org/10.1145/3508352.3549429.

Yaqin Zhou, Shangqing Liu, Jing Kai Siow, Xiaoning Du, and Yang Liu. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. In Hanna M. Wallach, Hugo Larochelle, Alina Beygelzimer, Florence d'Alché-Buc, Emily B. Fox, and Roman Garnett (eds.), *Advances in Neural Information Processing Systems 32: Annual Conference on Neural Information Processing Systems 2019, NeurIPS 2019, December 8-14, 2019, Vancouver, BC, Canada*, pp. 10197–10207, 2019. URL https://proceedings.neurips.cc/paper/2019/hash/49265d2447bc3bbfe9e76306ce40a31f-Abstract.html.

Ming Zhu, Aneesh Jain, Karthik Suresh, Roshan Ravindran, Sindhu Tipirneni, and Chandan K. Reddy. Xlcost: A benchmark dataset for cross-lingual code intelligence. *CoRR*, abs/2206.08474, 2022a. doi: 10.48550/ARXIV.2206.08474. URL https://doi.org/10.48550/arXiv.2206.08474.

Ming Zhu, Karthik Suresh, and Chandan K. Reddy. Multilingual code snippets training for program translation. In *Thirty-Sixth AAAI Conference on Artificial Intelligence, AAAI 2022, Thirty-Fourth Conference on Innovative Applications of Artificial Intelligence, IAAI 2022, The Twelveth Symposium on Educational Advances in Artificial Intelligence, EAAI 2022 Virtual Event, February 22 - March 1, 2022*, pp. 11783–11790. AAAI Press, 2022b. doi: 10.1609/AAAI.V36I10.21434. URL https://doi.org/10.1609/aaai.v36i10.21434.

Terry Yue Zhuo, Minh Chien Vu, Jenny Chim, Han Hu, Wenhao Yu, Ratnadira Widyasari, Imam Nur Bani Yusuf, Haolan Zhan, Junda He, Indraneil Paul, Simon Brunner, Chen Gong, Thong Hoang, Armel Randy Zebaze, Xiaoheng Hong, Wen-Ding Li, Jean Kaddour, Ming Xu, Zhihan Zhang, Prateek Yadav, Naman Jain, Alex Gu, Zhoujun Cheng, Jiawei Liu, Qian Liu, Zijian Wang,

David Lo, Binyuan Hui, Niklas Muennighoff, Daniel Fried, Xiaoning Du, Harm de Vries, and Leandro von Werra. Bigcodebench: Benchmarking code generation with diverse function calls and complex instructions. *CoRR*, abs/2406.15877, 2024. doi: 10.48550/ARXIV.2406.15877. URL https://doi.org/10.48550/arXiv.2406.15877.

A Model Architecture & Training Details

Attribute	ObscuraCoder 255M	ObscuraCoder 491M	ObscuraCoder 1.2B	ObscuraCoder 2.8B
Training Attributes				
Scheduler Type	Cosine			
Scheduler Warmup Prop.	0.05			
Optimizer Type	AdamW			
Peak LR	5e-4			
Terminal LR	2.5e-5			
Beta	{0.9, 0.95}			
Epsilon	1e-8			
Gradient Clipping	1.0			
Weight Decay	0.1			
Deepspeed variant	Stage-2			
Model Datatype	bfloat16			
Softmax Datatype	float32			
FP32 AllReduce	True			
Pipeline Parallel	False			
Activation Checkpointing	False			
Global Batch Size	256			
Training Steps	520000			
Contiguous Gradients	True			
Round Robin Gradients	True			
Scattered Reduce	True			
Partitioned Allgather	True			
Architecture Attributes				
Norm Type	RMS			
Norm Epsilon	1e-6			
Pos. Embed. Type	Rotary			
Pos. Embed. Period	1000000			
Pos. Embed. Prop.	1.0			
Sequence Length	2048			
MLP Type	Llama			
Activation Function	SiLU			
Weight Tying	False			
Attention Variant	Flash Attention-2			
Tokenizer Variant	Byte-Level BPE			
Tokenizer Vocab. Size	49152			
QK Norm	False			
Layer Count	12	12	20	22
QKV Heads Count	8	12	16	24
Hidden Size	1024	1536	2048	3072
Intermediate Size	2816	4096	5632	8192
Non Embed. Params.	204M	415M	1128M	2643M
Total Params.	255M	491M	1229M	2794M

Table 4: Training and architectural attributes of the `ObscuraCoder` suite of models.

B Dataset Details

B.1 ObscuraX Collection

Language	Sample Count	Token Count		
		Original	Obfuscated	Total
C	5,986,186	11,619,163,328	10,962,297,717	22,581,461,045
C++	5,374,108	9,470,872,459	8,961,976,794	18,432,849,253
Go	5,188,074	5,063,743,140	4,903,380,220	9,967,123,360
Java	18,523,273	18,064,491,641	17,411,757,778	35,476,249,419
Python	13,600,833	11,183,172,044	10,718,436,155	21,901,608,199
Rust	1,263,892	1,533,690,227	1,485,340,352	3,019,030,579
Typescript	5,403,241	3,855,020,996	3,726,795,270	7,581,816,266
Total:	55,339,607	60,790,153,835	58,169,984,286	118,960,138,121

Table 5: Language-wise sample and token count breakdown of the `ObscuraX` dataset.

B.1.1 Dataset Limitations & Failed Collection Attempts

Limitations. `ObscuraX` is collected with the stated objective of allowing Code-LMs to better disentangle code syntax and semantics using obfuscation as a semantic-preserving augmentation. However, there exist some corner cases beyond the import obfuscation alluded to in Section 3, where the correctness of the original code is not preserved.

One such scenario is wildcard or global module-level imports, where our obfuscator treats the imported members no differently from user-defined ones. This can affect correctness in cases such as

`*` imports in Python or when header-only libraries are used in languages like C++. In practice, we find the occurrence of these to be limited and hence do not tailor our obfuscator for them. Another scenario is the potential obfuscation of standard macros in languages like Rust and C. These are pre-defined by the language standard and encapsulate platform-level functionality. However, their use is exceedingly rare, and we avoid any special handling of these structures.

Failed collection attempts. Before building a custom obfuscator, we attempted to re-purpose existing open-source semantic highlighting toolchains for obfuscation. We detail two failed efforts below:

- git `github/semantic`: The semantic highlighting package made public by GitHub was a natural first attempt. However, the framework seeks to map elements in all programming languages to a closed set of Haskell constructs, which can be limiting. Furthermore, it does not expose a Tree-sitter-like query layer for targeted extraction of information.
- git `microsoft/language-server-protocol`: We attempted to instrument the semantic highlighting mechanism of popular IDEs for obfuscation, by directly extracting all spans relating to specific identifiers. However, in practice, the language servers of many languages are immature and slow. We were further hamstrung by the fact that most language servers work reliably only at the repository level, which was at odds with the file-level sourcing of `ObscuraX`.

B.2 Pre-Training Corpus Collection

`filtered-code-text` collection. The `filtered-code-text` collection consists of a text-only split consisting of roughly 105B tokens and a code-adjacent split comprised of a further 15B tokens. The text-only split's sourcing is biased towards technical, academic and educational content. We list the constituent HuggingFace sources of the text-only split along with the at-source filtering procedures (if any) below:

- 🤗 `allenai/c4`: We subset for the `realnewslike` split.
- 🤗 `Skylion007/openwebtext`
- 🤗 `allenai/peS2o`: We use regex-based removal of citation text and bibliography information.
- 🤗 `datajuicer/redpajama-book-refined-by-data-juicer`
- 🤗 `datajuicer/redpajama-pile-stackexchange-refined-by-data-juicer`
- 🤗 `datajuicer/the-pile-uspto-refined-by-data-juicer`
- 🤗 `datajuicer/redpajama-wiki-refined-by-data-juicer`
- 🤗 `datajuicer/the-pile-nih-refined-by-data-juicer`
- 🤗 `SciPhi/textbooks-are-all-you-need-lite`
- 🤗 `wentingzhao/math-textbooks`
- 🤗 `airtrain-ai/fineweb-edu-fortified`: We further bias for educational content by selecting samples with an educational `score` greater than 3.5 and a repeat `count` greater than 2.

The code-adjacent split focuses on organic and synthetic data sources where text and code appear interspersed, with the text fulfilling a descriptive or pedagogical role. The following lists the constituent HuggingFace sources of this split along with the at-source filtering procedures (if any):

- 🤗 `vikp/textbook_quality_programming`
- 🤗 `bigcode/stack-exchange-preferences-20230914-clean`: We select QA pairs where the answer has been accepted by the community and the answerer has a reputation of $\geq$3. We also use QA pairs from only the following sub-domains: `stackoverflow`, `serverfault`, `askubuntu`, `superuser`, and `stackexchange`.
- 🤗 `vikp/code_with_explanations`
- 🤗 `nampdn-ai/devdocs.io`
- 🤗 `open-phi/programming _books_llama`
- 🤗 `SivilTaram/starcoder2-documentation`
- 🤗 `bigcode/coding_tutorials`

- 🤗 `bigcode/commitpackft`
- 🤗 `bigcode/the-stack-github-issues`

The two splits are merged, shuffled and subsequently subjected to the following steps of filtering:

- We follow recent work (Marion et al., 2023) and prune our corpus using a perplexity filter to improve pre-training efficiency. We leverage a KenLM (Heafield, 2011) model trained on the EN OSCAR (Abadji et al., 2022) corpus and discard documents with a length-normalized perplexity of more than 325 and less than 7.
- We follow established practice (Penedo et al., 2023) and discard samples with any <=4-gram whose repeats constitute greater than 15% of the respective n-gram count.
- We remove samples whose English language score is lower than 99%. We employ the Lingua [13] language detector to make this determination.
- Finally, we perform an aggressive MinHash (Broder, 1997) de-duplication using a shingle size of 8 and a similarity threshold of 0.5. This allows us to efficiently approximate frontier model performance with constrained compute (Lee et al., 2022).

`filtered-source-code` collection. The `filtered-source-code` collection is the result of pruning an already filtered version [14] of the Stack (Kocetkov et al., 2023). We subset for the seven language splits, which we evaluate on — C, C++, Go, Java, Python, Rust, and TypeScript — and supplement it with the Markdown split. On the resultant data, we further apply the following filters:

- For files forked more than 25 times, we retain them if the average line length is less than 140, the maximum line length is less than 500, and the alphanumeric fraction is more than 25%.
- For files forked between 10 and 25 times, we retain them if the average line length is less than 120, the maximum line length is less than 200, and the alphanumeric fraction is more than 35%.
- For files forked less than 10 times, we retain them if the average line length is less than 100, the maximum line length is less than 200, and the alphanumeric fraction is more than 40%.
- We only retain samples from conventionally used file extensions and drop samples from valid but uncommon extensions.
- Seeking to avoid the deleterious effects of near-duplicate data on Code-LMs (Allamanis, 2019), we subject the resultant data to an aggressive MinHash (Broder, 1997) de-duplication, using a shingle size of 20 and a similarity threshold of 0.75.

Language	Chosen Extensions	Sample Count	Token Count
C	{.c, .h}	8,149,577	14,381,796,257
C++	{.cpp, .c++, .cc, .cxx, .h++, .hh, .hpp, .hxx}	5,923,165	10,912,556,820
Go	{.go}	4,507,348	9,967,123,360
Java	{.java}	19,324,891	16,249,736,121
Markdown	{.md, .markdown}	12,623,416	5,614,520,807
Python	{.py}	12,304,123	12,868,526,213
Rust	{.rs}	1,314,569	1,932,035,717
Typescript	{.ts, .tsx}	8,207,133	4,437,582,117
Total:		72,354,222	76,363,877,412

Table 6: Language-wise chosen extensions along with sample and token count breakdown of the `filtered-source-code` collection used to construct `ObscuraCoder`, DOBF, and Causal LM pre-training corpora.

B.3 Obfuscated Code Examples

For the reader's reference, we provide complete `ObscuraX` samples in all supported languages along with the accompanying metadata.

[13] git `pemistahl/lingua-py`
[14] 🤗 `bigcode/the-stack-dedup`

Original Python Code

```python
from __future__ import print_function, division, absolute_import
import sys, os.path

_importlist = "VisitorBase parse check".split()

_cd = os.path.join(os.path.dirname(__file__))
version_specific_path = None
common_path = os.path.join(_cd, 'common')

def _get_asdl_depending_on_version():

    global version_specific_path
    major, minor = sys.version_info[0], sys.version_info[1]
    prefix = __name__.rsplit('.', 1)
    base = (prefix[0] + '.') if len(prefix) > 1 else ''
    dname = 'py%d_%d' % (major, minor)
    version_specific_path = os.path.join(_cd, dname)

    use_abs_import = 0

    mname = base + dname + '.asdl'
    try:
        mod = __import__(mname, fromlist=_importlist, level=use_abs_import)
    except ImportError:
        dname = 'common'
        mname = base + dname + '.asdl'
        mod = __import__(mname, fromlist=_importlist)
    for i in _importlist:
        globals()[i] = getattr(mod, i)
```

Obfuscated Python Code

```python
from __future__ import IMPORT_0, IMPORT_1, IMPORT_2
import sys, os.IMPORT_3

VAR_0 = "VisitorBase parse check".FUNC_0()

VAR_1 = os.IMPORT_3.FUNC_1(os.IMPORT_3.FUNC_2(VAR_2))
VAR_3 = None
common_path = os.IMPORT_3.FUNC_1(VAR_1, 'common')

def _get_asdl_depending_on_version():

    global VAR_3
    VAR_4, VAR_5 = sys.VAR_6[0], sys.VAR_6[1]
    VAR_7 = VAR_8.FUNC_3('.', 1)
    base = (VAR_7[0] + '.') if FUNC_4(VAR_7) > 1 else ''
    VAR_9 = 'py%d_%d' % (VAR_4, VAR_5)
    VAR_3 = os.IMPORT_3.FUNC_1(VAR_1, VAR_9)

    VAR_10 = 0

    VAR_11 = base + VAR_9 + '.asdl'
    try:
        VAR_12 = FUNC_5(VAR_11, VAR_13=VAR_0, VAR_14=VAR_10)
    except VAR_15:
        VAR_9 = 'common'
        VAR_11 = base + VAR_9 + '.asdl'
        VAR_12 = FUNC_5(VAR_11, VAR_13=VAR_0)
    for VAR_16 in VAR_0:
        FUNC_6()[VAR_16] = FUNC_7(VAR_12, VAR_16)
```

Metadata

Obfuscation proportion: 85.6%

'IMPORT_0': 'print_function', 'IMPORT_1': 'division', 'IMPORT_2': 'absolute_import', 'IMPORT_3': 'path', 'VAR_0': '_importlist', 'FUNC_0': 'split', 'VAR_1': '_cd', 'FUNC_1': 'join', 'FUNC_2': 'dirname', 'VAR_2': '__file__', 'VAR_3': 'version_specific_path', 'VAR_4': 'major', 'VAR_5': 'minor', 'VAR_6': 'version_info', 'VAR_7': 'prefix', 'VAR_8': '__name__', 'FUNC_3': 'rsplit', 'FUNC_4': 'len', 'VAR_9': 'dname', 'VAR_10': 'use_abs_import', 'VAR_11': 'mname', 'VAR_12': 'mod', 'FUNC_5': '__import__', 'VAR_13': 'fromlist', 'VAR_14': 'level', 'VAR_15': 'ImportError', 'VAR_16': 'i', 'FUNC_6': 'globals', 'FUNC_7': 'getattr', 'FUNC_8': 'load', 'VAR_17': 'filename', 'VAR_18': 'srcfile', 'FUNC_9': 'exists', 'VAR_19': 'asdl', 'FUNC_10': 'parse', 'FUNC_11': 'check'.

Figure 3: An example of original and obfuscated Python code in `ObscuraX` along with accompanying metadata stating the obfuscation proportion and the identifier map.

Original Java Code

```java
package com.dji.GSDemo.GoogleMap;

import android.os.Bundle;
import android.support.v7.app.AppCompatActivity;
import android.content.Intent;
import android.widget.TextView;


public class showDetail extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.show_details);

        TextView result = (TextView) findViewById(R.id.textView2);
        TextView resultData = (TextView) findViewById(R.id.textView3);
        Intent in = getIntent();
        result.setText(in.getStringExtra("name"));
        resultData.setText(in.getStringExtra("data"));
    }
}
```

Obfuscated Java Code

```java
package IMPORT_0.dji.GSDemo.GoogleMap;

import android.os.Bundle;
import android.IMPORT_1.v7.app.AppCompatActivity;
import android.content.IMPORT_2;
import android.widget.TextView;


public class showDetail extends AppCompatActivity {
    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.show_details);

        TextView result = (TextView) findViewById(R.id.textView2);
        TextView resultData = (TextView) findViewById(R.id.textView3);
        IMPORT_2 VAR_0 = getIntent();
        result.FUNC_0(VAR_0.getStringExtra("name"));
        resultData.FUNC_0(VAR_0.getStringExtra("data"));
    }
}
```

Metadata

Obfuscation proportion: 12.0%

'IMPORT_0': 'com', 'IMPORT_1': 'support', 'IMPORT_2': 'Intent', 'VAR_0': 'in', 'FUNC_0': 'setText'.

Figure 4: An example of original and obfuscated Java code in `ObscuraX` along with accompanying metadata stating the obfuscation proportion and the identifier map.

Original C++ Code

```cpp
#include <iostream>
#include <stdio.h>
#include <string.h>
using namespace std;
int n,m;
int data;
int tree[1005];
char str[10][10];
int lowbit(int i)
{
    return i&(-i);
}
void add(int i,int x)
{
    while(i<=n)
    {
        tree[i]+=x;
        i+=lowbit(i);
    }
}
int sum(int i)
{
    int ans=0;
    while(i>0)
    {
        ans+=tree[i];
        i-=lowbit(i);
    }
    return ans;
}
```

Obfuscated C++ Code

```cpp
#include <iostream>
#include <IMPORT_0>
#include <string.h>
using namespace std;
int VAR_0,VAR_1;
int data;
int tree[1005];
char str[10][10];
int lowbit(int VAR_2)
{
    return VAR_2&(-VAR_2);
}
void add(int VAR_2,int VAR_3)
{
    while(VAR_2<=VAR_0)
    {
        tree[VAR_2]+=VAR_3;
        VAR_2+=lowbit(VAR_2);
    }
}
int sum(int VAR_2)
{
    int ans=0;
    while(VAR_2>0)
    {
        ans+=tree[VAR_2];
        VAR_2-=lowbit(VAR_2);
    }
    return ans;
}
```

Metadata

```
Obfuscation proportion: 29.9%
```

```
'IMPORT_0': 'stdio.h', 'VAR_0': 'n', 'VAR_1': 'm', 'VAR_2': 'i', 'VAR_3': 'x'.
```

Figure 5: An example of original and obfuscated C++ code in `ObscuraX` along with accompanying metadata stating the obfuscation proportion and the identifier map.

Original C Code

```c
void dshot_dma_start() {
  uint32_t time = time_micros();
  while ((dshot_dma_phase != 0 || spi_dma_is_ready(SPI_PORT1) == 0) && (
  time_micros() - time) < state.looptime * 1e6f)
    ;
  if (dshot_dma_phase != 0 || spi_dma_is_ready(SPI_PORT1) == 0)
    return;
  for (uint8_t i = 0; i < 16; i++) {
    motor_data_portA[i] = 0;
    motor_data_portB[i] = 0;
    motor_data_portC[i] = 0;
#define MOTOR_PIN(port, pin, pin_af, timer, timer_channel)    \
  if (!(dshot_packet[MOTOR_PIN_IDENT(port, pin)] & 0x8000)) { \
    if (GPIO##port == GPIOA)                                  \
      motor_data_portA[i] |= (LL_GPIO_PIN_##pin << 16);       \
    else if (GPIO##port == GPIOB)                             \
      motor_data_portB[i] |= (LL_GPIO_PIN_##pin << 16);       \
    else if (GPIO##port == GPIOC)                             \
      motor_data_portC[i] |= (LL_GPIO_PIN_##pin << 16);       \
  }
    MOTOR_PINS
#undef MOTOR_PIN
    dshot_packet[0] <<= 1;
    dshot_packet[1] <<= 1;
    dshot_packet[2] <<= 1;
    dshot_packet[3] <<= 1;
  }
  for (int i = 1, j = 0; i < 48 && j < 16; i += 3, j++) {
    portA_buffer[i] = motor_data_portA[j];
    portB_buffer[i] = motor_data_portB[j];
    portC_buffer[i] = motor_data_portC[j];
  }
  dshot_dma_phase = DSHOT_PORT_COUNT;
  TIM1->ARR = DSHOT_BIT_TIME;
  TIM1->CCR1 = 0;
  TIM1->CCR2 = DSHOT_T0H_TIME;
  TIM1->CCR3 = DSHOT_T1H_TIME;
  if (DSHOT_GPIO_A == 1)
    dshot_dma_portA();
  else if (DSHOT_GPIO_B == 1)
    dshot_dma_portB();
  else if (DSHOT_GPIO_C == 1)
    dshot_dma_portC();
}
```

Obfuscated C Code

```c
void FUNC_0() {
  uint32_t VAR_0 = FUNC_1();
  while ((VAR_1 != 0 || FUNC_2(VAR_2) == 0) && (FUNC_1() - VAR_0) < VAR_3.
  VAR_4 * 1e6f)
    ;
  if (VAR_1 != 0 || FUNC_2(VAR_2) == 0)
    return;
  for (uint8_t VAR_5 = 0; VAR_5 < 16; VAR_5++) {
    VAR_6[VAR_5] = 0;
    VAR_7[VAR_5] = 0;
    VAR_8[VAR_5] = 0;
#define FUNC_3(VAR_9, VAR_10, VAR_11, VAR_12, VAR_13)    \
  if (!(dshot_packet[MOTOR_PIN_IDENT(port, pin)] & 0x8000)) { \
    if (GPIO##port == GPIOA)                                  \
      motor_data_portA[i] |= (LL_GPIO_PIN_##pin << 16);       \
    else if (GPIO##port == GPIOB)                             \
      motor_data_portB[i] |= (LL_GPIO_PIN_##pin << 16);       \
    else if (GPIO##port == GPIOC)                             \
      motor_data_portC[i] |= (LL_GPIO_PIN_##pin << 16);       \
  }
    CLASS_0
#undef MOTOR_PIN
    VAR_14[0] <<= 1;
    VAR_14[1] <<= 1;
    VAR_14[2] <<= 1;
    VAR_14[3] <<= 1;
  }
  for (int VAR_5 = 1, VAR_15 = 0; VAR_5 < 48 && VAR_15 < 16; VAR_5 += 3,
  VAR_15++) {
    VAR_16[VAR_5] = VAR_6[VAR_15];
    VAR_17[VAR_5] = VAR_7[VAR_15];
    VAR_18[VAR_5] = VAR_8[VAR_15];
  }
  VAR_1 = VAR_19;
  VAR_20->VAR_21 = VAR_22;
  VAR_20->VAR_23 = 0;
  VAR_20->VAR_24 = DSHOT_T0H_TIME;
  VAR_20->VAR_25 = DSHOT_T1H_TIME;
  if (VAR_26 == 1)
    FUNC_4();
  else if (VAR_27 == 1)
    FUNC_5();
  else if (VAR_28 == 1)
    FUNC_6();
}
```

Metadata

Obfuscation proportion: 82.1%

'FUNC_0': 'dshot_dma_start', 'VAR_0': 'time', 'FUNC_1': 'time_micros', 'VAR_1': 'dshot_dma_phase', 'FUNC_2': 'spi_dma_is_ready', 'VAR_2': 'SPI_PORT1', 'VAR_3' : 'state', 'VAR_4': 'looptime', 'VAR_5': 'i', 'VAR_6': 'motor_data_portA', 'VAR_7': 'motor_data_portB', 'VAR_8': 'motor_data_portC', 'FUNC_3': 'MOTOR_PIN', ' VAR_9': 'port', 'VAR_10': 'pin', 'VAR_11': 'pin_af', 'VAR_12': 'timer', 'VAR_13': 'timer_channel', 'CLASS_0': 'MOTOR_PINS', 'VAR_14': 'dshot_packet', 'VAR_15' : 'j', 'VAR_16': 'portA_buffer', 'VAR_17': 'portB_buffer', 'VAR_18': 'portC_buffer', 'VAR_19': 'DSHOT_PORT_COUNT', 'VAR_20': 'TIM1', 'VAR_21': 'ARR', 'VAR_22' : 'DSHOT_BIT_TIME', 'VAR_23': 'CCR1', 'VAR_24': 'CCR2', 'VAR_25': 'CCR3', 'VAR_26': 'DSHOT_GPIO_A', 'FUNC_4': 'dshot_dma_portA', 'VAR_27': 'DSHOT_GPIO_B', ' FUNC_5': 'dshot_dma_portB', 'VAR_28': 'DSHOT_GPIO_C', 'FUNC_6': 'dshot_dma_portC'.

Figure 6: An example of original and obfuscated C code in ObscuraX along with accompanying metadata stating the obfuscation proportion and the identifier map.

Original Rust Code

```rust
/// Generate an encrypted header
/// for a resource encrypted using an hybrid crypto scheme.
///
/// A random symmetric key is generated for the specified symmetric scheme,
/// encrypted using the public key of the ABE scheme and policy attributes
/// then pre-pended to the symmetrically encrypted metadata
pub fn encrypt_hybrid_header<A, S>(
    policy: &Policy,
    public_key: &A::MasterPublicKey,
    attributes: &[Attribute],
    meta_data: Option<Metadata>,
) -> Result<EncryptedHeader<S>, FormatErr>
where
    A: AbeScheme + std::marker::Sync + std::marker::Send,
    S: SymmetricCrypto,
{
    let engine = Engine::<A>::new();
    let (sk_bytes, encrypted_sk) =
        engine.generate_symmetric_key(policy, public_key, attributes, S::Key
        ::LENGTH)?;
    let symmetric_key = S::Key::try_from_bytes(sk_bytes)?;
    // convert to bytes
    // ..size
    let mut header_bytes = u32_len(&encrypted_sk)?.to_vec();
    // ...bytes
    header_bytes.extend(&encrypted_sk);
    if let Some(meta) = meta_data {
        // Nonce
        let nonce = S::Nonce::new(&mut CsRng::new());
        header_bytes.extend(nonce.to_bytes());
        // Encrypted metadata
        let encrypted_metadata = S::encrypt(&symmetric_key, &meta.to_bytes()
        ?, &nonce, None)?;
        // ... size
        header_bytes.extend(u32_len(&encrypted_metadata)?);
        // ... bytes
        header_bytes.extend(encrypted_metadata);
    }
    Ok(EncryptedHeader {
        symmetric_key,
        encrypted_header_bytes: header_bytes,
    })
}
```

Obfuscated Rust Code

```rust
/// Generate an encrypted header
/// for a resource encrypted using an hybrid crypto scheme.
///
/// A random symmetric key is generated for the specified symmetric scheme,
/// encrypted using the public key of the ABE scheme and policy attributes
/// then pre-pended to the symmetrically encrypted metadata
pub fn encrypt_hybrid_header<A, S>(
    policy: &Policy,
    public_key: &A::MasterPublicKey,
    attributes: &[Attribute],
    meta_data: Option<Metadata>,
) -> Result<EncryptedHeader<S>, FormatErr>
where
    A: AbeScheme + std::marker::Sync + std::marker::Send,
    S: SymmetricCrypto,
{
    let engine = Engine::<A>::new();
    let (sk_bytes, encrypted_sk) =
        engine.generate_symmetric_key(policy, public_key, attributes, S::Key
        ::LENGTH)?;
    let symmetric_key = S::Key::try_from_bytes(sk_bytes)?;
    // convert to bytes
    // ..size
    let mut header_bytes = u32_len(&encrypted_sk)?.to_vec();
    // ...bytes
    header_bytes.extend(&encrypted_sk);
    if let Some(meta) = meta_data {
        // Nonce
        let nonce = S::Nonce::new(&mut CsRng::new());
        header_bytes.extend(nonce.FUNC_0());
        // Encrypted metadata
        let encrypted_metadata = S::encrypt(&symmetric_key, &meta.FUNC_0()?,
         &nonce, None)?;
        // ... size
        header_bytes.extend(u32_len(&encrypted_metadata)?);
        // ... bytes
        header_bytes.extend(encrypted_metadata);
    }
    Ok(EncryptedHeader {
        symmetric_key,
        VAR_0: header_bytes,
    })
```

Metadata

```
Obfuscation proportion: 2.6%
```

```
'FUNC_0': 'to_bytes', 'VAR_0': 'encrypted_header_bytes'
```

Figure 7: An example of original and obfuscated Rust code in `ObscuraX` along with accompanying metadata stating the obfuscation proportion and the identifier map.

Original Typescript Code

```typescript
export default (data: any[], period: number = 21) => {
  let position = "BULL"

  // console.log(data)
  // get data from the end of the dataset backward (positive to negative
  number)
  const dataSet = data.slice(-Math.abs(period))

  // calculate simple moving average for the period
  const sma = dataSet.map((item: any) => item.close).reduce((prev: number,
  curr: number) => prev + curr, 0) / period

  // Bull or Bear?
  // if the last price is above the SMA = Bull
  // if the last price is below the SMA = Bear
  const lastDataPoint = data.slice(-1)[0]
  // console.log(101, lastDataPoint)

  const { startTime, close } = lastDataPoint

  if (close < sma) {
    position = "BEAR"
  }

  return { startTime, sma, position }
}
```

Obfuscated Typescript Code

```typescript
export default (data: any[], period: number = 21) => {
  let position = "BULL"

  // console.log(data)
  // get data from the end of the dataset backward (positive to negative
  number)
  const VAR_0 = data.slice(-Math.abs(period))

  // calculate simple moving average for the period
  const VAR_1 = VAR_0.FUNC_0((item: any) => item.VAR_2).reduce((prev: number
  , VAR_3: number) => prev + VAR_3, 0) / period

  // Bull or Bear?
  // if the last price is above the SMA = Bull
  // if the last price is below the SMA = Bear
  const lastDataPoint = data.slice(-1)[0]
  // console.log(101, lastDataPoint)

  const { startTime, close } = lastDataPoint

  if (VAR_2 < VAR_1) {
    position = "BEAR"
  }

  return { startTime, VAR_1, position }
}
```

Metadata

Obfuscation proportion: 25.1%

```
'VAR_0': 'dataSet', 'VAR_1': 'sma', 'FUNC_0': 'map', 'VAR_2': 'close', 'VAR_3': 'curr'.
```

Figure 8: An example of original and obfuscated TypeScript code in `ObscuraX` along with accompanying metadata stating the obfuscation proportion and the identifier map.

Original Go Code

```go
// Recv gets the replied icmp packet
func (i *Trace) Recv(id, seq int) (ICMPResp, error) {
	var (
		b    = make([]byte, 512)
		ts   = time.Now()
		resp ICMPResp
		wID  bool
		wSeq bool
		wDst bool
	)
	for {
		n, from, err := syscall.Recvfrom(i.fd, b, 0)
		if err != nil {
			du, _ := time.ParseDuration(i.wait)
			if err == syscall.EAGAIN && time.Since(ts) < du {
				continue
			}
			return resp, err
		}
		b = b[:n]
		if len(i.ip.To4()) == net.IPv4len {
			resp = icmpV4RespParser(b)
			wID = resp.typ == IPv4ICMPTypeEchoReply && id !=
			resp.id
			wSeq = seq != resp.seq
			wDst = resp.ip.dst.String() != i.ip.String()
		} else {
			resp = icmpV6RespParser(b)
			resp.src = net.IP(from.(*syscall.SockaddrInet6).Addr
			[:])
			wID = resp.typ == IPv6ICMPTypeEchoReply && id !=
			resp.id
			wSeq = seq != resp.seq
			wDst = resp.ip.dst.String() != i.ip.String()
		}
		if (i.icmp && wSeq) || wDst || wID {
			du, _ := time.ParseDuration(i.wait)
			if time.Since(ts) < du {
				continue
			}
			return resp, fmt.Errorf("wrong response")
		}
		break
	}
	return resp, nil
}
```

Obfuscated Go Code

```go
// Recv gets the replied icmp packet
func (VAR_0 *Trace) FUNC_0(VAR_1, seq int) (ICMPResp, CLASS_0) {
	var (
		b    = FUNC_1([]CLASS_1, 512)
		ts   = VAR_2.FUNC_2()
		resp ICMPResp
		wID  bool
		wSeq bool
		wDst bool
	)
	for {
		n, VAR_3, VAR_4 := syscall.FUNC_3(VAR_0.fd, b, 0)
		if VAR_4 != nil {
			du, _ := VAR_2.FUNC_4(VAR_0.wait)
			if VAR_4 == syscall.EAGAIN && VAR_2.Since(ts) < du {
				continue
			}
			return resp, VAR_4
		}
		b = b[:n]
		if len(VAR_0.ip.FUNC_5()) == net.IPv4len {
			resp = icmpV4RespParser(b)
			wID = resp.typ == IPv4ICMPTypeEchoReply && VAR_1 !=
			resp.VAR_1
			wSeq = seq != resp.seq
			wDst = resp.ip.dst.String() != VAR_0.ip.String()
		} else {
			resp = FUNC_6(b)
			resp.VAR_5 = net.IP(VAR_3.(*syscall.CLASS_2).VAR_6
			[:])
			wID = resp.typ == IPv6ICMPTypeEchoReply && VAR_1 !=
			resp.VAR_1
			wSeq = seq != resp.seq
			wDst = resp.ip.dst.String() != VAR_0.ip.String()
		}
		if (VAR_0.icmp && wSeq) || wDst || wID {
			du, _ := VAR_2.FUNC_4(VAR_0.wait)
			if VAR_2.Since(ts) < du {
				continue
			}
			return resp, fmt.Errorf("wrong response")
		}
		break
	}
	return resp, nil
}
```

Metadata

Obfuscation proportion: 42.6%

'VAR_0': 'i', 'FUNC_0': 'Recv', 'VAR_1': 'id', 'CLASS_0': 'error', 'FUNC_1': 'make', 'CLASS_1': 'byte', 'VAR_2': 'time', 'FUNC_2': 'Now', 'VAR_3': 'from', 'VAR_4': 'err', 'FUNC_3': 'Recvfrom', 'FUNC_4': 'ParseDuration', 'FUNC_5': 'To4', 'FUNC_6': 'icmpV6RespParser', 'VAR_5': 'src', 'CLASS_2': 'SockaddrInet6', 'VAR_6': 'Addr'.

Figure 9: An example of original and obfuscated Go code in `ObscuraX` along with accompanying metadata stating the obfuscation proportion and the identifier map.

C DETAILED RESULTS

For completeness, we detail the split and language-wise performance of the models on all tasks (where applicable) discussed in Section 5.

Model	Doc to Comments	Newline After Code	Newline After Doc	Newline Random	Line Split	Tab Indent
CausalLM 255M	9.75	12.19	10.99	10.99	11.51	12.77
ObscuraCoder 255M	11.58 +1.83	15.91 +3.72	14.98 +3.99	12.91 +1.92	14.97 +3.46	14.67 +1.90
CausalLM 491M	12.19	15.88	16.46	13.41	14.02	18.29
ObscuraCoder 491M	21.34 +9.15	26.21 +10.33	26.82 +10.36	21.95 +8.54	22.56 +8.54	24.39 +6.10
CausalLM 1.2B	22.89	29.55	29.55	26.87	27.43	29.59
ObscuraCoder 1.2B	33.56 +10.67	36.78 +7.23	35.36 +5.81	36.97 +10.10	36.46 +9.03	39.11 +9.52
CausalLM 2.8B	30.78	35.11	36.93	34.01	35.67	35.67
ObscuraCoder 2.8B	38.93 +8.15	46.71 +11.60	48.36 +11.43	44.74 +10.73	48.06 +12.39	44.91 +9.24
StarCoderBase 1B	25.77	31.67	31.67	27.43	28.65	28.65
DeepSeekCoder 1.3B	41.36	52.94	46.88	50.68	44.91	51.47
StarCoderBase 3B	35.36	37.35	39.63	35.67	37.07	38.17
StarCoder2 3B	49.12	60.31	49.73	51.94	54.58	57.21
StableCode 3B	44.89	50.79	55.48	48.79	50.28	50.28
CodeGemma 2B	9.56	11.71	12.97	9.14	10.53	7.92
Phi-2	58.25	59.34	68.12	61.46	49.12	63.63

Table 7: ReCode Format `pass@1` comparison between `ObscuraCoder` and comparable Causal LM models, along with frontier models for context.

Model	Dead Code Insert	For While Transform	Operand Swap	Var Renaming CB	Var Renaming Naive	Var Renaming RN
CausalLM 255M	2.63	10.43	12.84	12.43	6.11	9.45
ObscuraCoder 255M	3.66 +1.03	15.48 +5.05	15.48 +2.64	15.87 +3.44	8.37 +2.26	11.96 +2.51
CausalLM 491M	4.87	13.74	16.46	13.74	7.92	11.06
ObscuraCoder 491M	8.53 +3.66	26.37 +12.63	25.14 +8.68	26.99 +13.25	14.21 +6.29	18.97 +7.91
CausalLM 1.2B	8.12	25.87	27.11	26.96	14.77	23.21
ObscuraCoder 1.2B	10.82 +2.70	34.77 +8.90	37.91 +10.80	34.56 +7.60	19.96 +5.19	29.13 +5.92
CausalLM 2.8B	10.22	34.13	36.15	35.97	14.98	25.76
ObscuraCoder 2.8B	17.03 +6.81	44.88 +10.75	47.56 +11.41	44.98 +9.01	23.25 +8.27	38.13 +12.37
StarCoderBase 1B	8.53	32.19	29.12	32.19	28.77	24.70
DeepSeekCoder 1.3B	17.19	51.78	50.24	55.06	49.36	42.87
StarCoderBase 3B	12.87	25.59	37.97	38.67	35.79	30.61
StarCoder2 3B	16.14	53.38	57.44	59.14	50.85	48.78
StableCode 3B	14.13	52.77	56.01	49.64	49.06	39.97
CodeGemma 2B	6.79	12.12	11.67	11.49	9.54	6.11
Phi-2	18.15	64.19	63.56	62.26	59.77	54.35

Table 8: ReCode Syntax `pass@1` comparison between `ObscuraCoder` and comparable Causal LM models, along with frontier models for context.

Model	Camel Case	Butter Fingers	Swap Characters	Change Character Case	Inflectional Variation	Synonym Substitution
CausalLM 255M	4.26	1.97	2.03	2.78	2.89	4.82
ObscuraCoder 255M	5.74 +1.48	4.26 +2.29	4.78 +2.75	3.14 +0.36	5.63 +2.74	6.13 +1.31
CausalLM 491M	6.71	4.87	5.49	2.43	6.97	4.87
ObscuraCoder 491M	11.59 +4.88	6.71 +1.84	8.56 +3.07	6.11 +3.68	8.95 +1.98	6.11 +1.24
CausalLM 1.2B	15.49	11.55	10.16	8.74	11.69	10.16
ObscuraCoder 1.2B	20.08 +4.59	16.21 +4.66	17.17 +7.01	13.44 +4.70	17.17 +5.48	15.89 +5.73
CausalLM 2.8B	18.12	14.29	15.98	8.53	15.98	15.44
ObscuraCoder 2.8B	25.32 +7.20	19.26 +4.97	17.12 +1.14	16.68 +8.15	21.40 +5.42	20.86 +5.42
StarCoderBase 1B	15.07	13.85	13.09	13.41	14.19	13.41
DeepSeekCoder 1.3B	27.43	24.82	26.56	19.45	28.28	25.10
StarCoderBase 3B	20.34	17.24	16.89	15.01	16.56	18.91
StarCoder2 3B	30.65	26.57	23.61	18.11	25.39	23.17
StableCode 3B	28.97	23.48	23.11	19.35	26.88	23.11
CodeGemma 2B	8.53	5.67	7.93	4.87	8.48	6.78
Phi-2	41.36	29.14	30.22	26.41	37.44	31.67

Table 9: ReCode Function `pass@1` comparison between `ObscuraCoder` and comparable Causal LM models, along with frontier models for context.

Model	C++	Java	Python	Rust	TypeScript
CausalLM 255M	3.51	4.29	4.32	2.78	6.53
ObscuraCoder 255M	6.94 +3.43	6.27 +1.98	6.47 +2.15	3.46 +0.68	6.53 0.00
CausalLM 491M	5.74	5.74	7.20	2.74	7.79
ObscuraCoder 491M	8.56 +2.82	8.60 +2.86	10.91 +3.71	6.61 +3.87	9.13 +1.34
CausalLM 1.2M	12.87	13.04	13.89	9.71	10.82
ObscuraCoder 1.2B	19.03 +6.16	18.26 +5.22	19.79 +5.90	15.69 +5.98	18.92 +8.10
CausalLM 1.2M	17.79	16.48	19.16	14.28	15.79
ObscuraCoder 2.8B	25.08 +7.29	23.56 +7.08	26.34 +7.18	16.93 +2.65	25.86 +10.07
StarCoderBase 1B	12.05	14.11	15.06	10.22	14.18
DeepSeekCoder 1.3B	29.88	29.11	28.87	18.69	27.52
StarCoderBase 3B	20.46	18.64	20.19	16.83	19.92
StarCoder2 3B	26.78	27.89	26.49	25.56	30.16
StableCode 3B	28.04	25.31	29.84	23.03	27.44
CodeGemma 2B	27.26	22.42	20.78	28.42	14.65
Phi-2	21.75	20.88	49.92	6.87	11.94

Table 10: Multipl-E `pass@1` comparison between `ObscuraCoder` and comparable Causal LM models, along with frontier models for context.

Model	C++	Java	Python	Rust	TypeScript
CausalLM 255M	7.34	7.62	9.45	4.08	7.15
ObscuraCoder 255M	9.75 +2.41	10.06 +2.44	13.95 +4.50	5.59 +1.51	8.95 +1.80
CausalLM 491M	10.41	10.38	13.81	7.59	11.02
ObscuraCoder 491M	14.29 +3.88	13.24 +2.86	17.31 +3.50	9.94 +2.35	16.88 +5.86
CausalLM 1.2B	20.18	19.53	22.11	15.81	19.89
ObscuraCoder 1.2B	28.41 +8.23	30.84 +11.31	33.54 +11.43	22.07 +6.26	31.83 +11.94
CausalLM 2.8B	29.78	27.99	29.19	27.11	30.07
ObscuraCoder 2.8B	39.22 +9.44	40.09 +12.10	42.39 +13.20	34.60 +7.49	40.77 +10.70
StarCoderBase 1B	23.12	23.79	24.97	18.88	22.41
DeepSeekCoder 1.3B	47.17	46.83	52.92	39.20	51.17
StarCoderBase 3B	30.78	33.99	36.86	30.84	34.95
StarCoder2 3B	50.47	47.78	51.67	48.54	49.69
StableCode 3B	50.81	46.89	54.64	40.17	47.12
CodeGemma 2B	45.31	40.07	32.63	46.11	25.77
Phi-2	47.77	37.13	71.78	11.76	31.38

Table 11: Multipl-E `pass@10` comparison between `ObscuraCoder` and comparable Causal LM models, along with frontier models for context.

Model	C++	Java	Python	Rust	TypeScript
CausalLM 255M	15.97	15.22	17.08	8.93	14.62
ObscuraCoder 255M	16.93 +0.96	18.38 +3.16	25.58 +8.50	11.43 +2.50	18.70 +4.08
CausalLM 491M	21.21	21.56	25.13	11.96	23.59
ObscuraCoder 491M	28.53 +7.32	27.88 +6.32	30.04 +4.91	18.91 +6.95	23.96 +0.37
CausalLM 1.2B	34.84	37.69	36.35	28.84	32.77
ObscuraCoder 1.2B	48.38 +13.54	51.91 +14.22	49.87 +13.52	40.15 +11.31	48.64 +15.87
CausalLM 2.8B	47.71	49.44	50.12	41.42	48.98
ObscuraCoder 2.8B	64.87 +17.16	63.74 +14.30	67.41 +17.29	53.45 +12.03	60.17 +11.19
StarCoderBase 1B	39.11	38.20	40.75	30.64	40.19
DeepSeekCoder 1.3B	71.48	67.42	77.79	64.05	77.76
StarCoderBase 3B	55.65	56.41	61.27	55.91	58.77
StarCoder2 3B	69.19	67.67	76.78	77.56	73.12
StableCode 3B	68.34	65.48	74.81	66.21	69.55
CodeGemma 2B	66.17	58.85	60.77	70.27	54.63
Phi-2	62.46	57.40	83.95	26.91	55.76

Table 12: Multipl-E `pass@100` comparison between `ObscuraCoder` and comparable Causal LM models, along with frontier models for context.

Model	C	C++	Go	Java	Python	TypeScript	Rust
CausalLM 255M	32.07	32.45	33.13	33.79	33.09	32.19	32.58
ObscuraCoder 255M	32.69 **+0.62**	33.26 **+0.81**	34.04 **+0.91**	34.42 **+0.63**	33.58 **+0.49**	32.59 **+0.40**	33.24 **+0.66**
CausalLM 491M	33.37	33.69	35.50	34.87	34.16	33.12	33.83
ObscuraCoder 491M	34.07 **+0.70**	34.67 **+0.98**	36.09 **+0.59**	35.64 **+0.77**	34.81 **+0.65**	33.68 **+0.56**	35.01 **+1.18**
CausalLM 1.2B	35.28	34.97	37.19	36.61	35.32	34.59	35.59
ObscuraCoder 1.2B	36.23 **+0.95**	36.67 **+1.70**	38.97 **+1.78**	37.69 **+1.08**	36.11 **+0.79**	36.24 **+1.65**	37.09 **+1.50**
CausalLM 2.8B	36.09	36.41	38.76	37.38	36.14	35.40	36.86
ObscuraCoder 2.8B	37.58 **+1.49**	37.84 **+1.43**	39.59 **+0.83**	38.97 **+1.59**	37.73 **+1.59**	36.29 **+0.89**	38.52 **+1.56**
StarCoderBase 1B	36.11	37.02	38.75	37.40	36.88	36.07	37.13
DeepSeekCoder 1.3B	35.81	35.79	35.64	36.68	35.99	35.09	36.01
StarCoderBase 3B	38.69	38.49	40.03	39.08	38.14	37.59	38.52
StarCoder2 3B	38.09	38.78	40.48	39.14	38.39	37.45	38.89
StableCode 3B	38.75	38.11	39.34	38.26	37.34	36.88	37.88
CodeGemma 2B	36.62	35.96	35.93	36.11	35.91	35.25	37.22
Phi-2	34.97	34.56	37.02	35.86	35.18	33.97	35.10

Table 13: CommitChronicle ROUGE-1 comparison between ObscuraCoder and comparable Causal LM models, along with frontier models for context.

Model	C	C++	Go	Java	Python	TypeScript	Rust
CausalLM 255M	9.85	9.70	10.19	11.31	10.44	9.47	10.10
ObscuraCoder 255M	10.42 **+0.57**	10.23 **+0.53**	10.98 **+0.79**	11.76 **+0.45**	10.79 **+0.35**	9.79 **+0.32**	10.51 **+0.41**
CausalLM 491M	10.82	10.64	11.38	12.09	11.20	10.01	11.01
ObscuraCoder 491M	11.22 **+0.40**	10.92 **+0.28**	11.83 **+0.45**	12.51 **+0.42**	11.56 **+0.38**	10.59 **+0.58**	11.45 **+0.44**
CausalLM 1.2B	11.80	11.57	12.52	13.12	12.01	10.93	11.78
ObscuraCoder 1.2B	13.16 **+1.36**	12.95 **+1.38**	13.98 **+1.46**	14.11 **+0.99**	13.08 **+1.07**	12.04 **+1.11**	13.08 **+1.30**
CausalLM 2.8B	13.04	12.55	13.58	14.00	12.87	11.86	12.31
ObscuraCoder 2.8B	14.28 **+1.24**	13.97 **+1.42**	14.91 **+1.33**	15.24 **+1.24**	14.14 **+1.27**	12.41 **+0.55**	14.12 **+1.81**
StarCoderBase 1B	12.98	12.88	14.01	14.03	13.23	12.08	13.14
DeepSeekCoder 1.3B	13.06	12.91	12.89	13.93	12.97	11.93	12.99
StarCoderBase 3B	14.84	14.45	14.92	15.16	14.09	13.19	14.46
StarCoder2 3B	14.75	14.43	15.21	15.26	14.44	13.37	14.63
StableCode 3B	15.46	13.87	15.05	14.65	13.87	12.89	14.63
CodeGemma 2B	15.08	11.84	13.47	14.06	12.77	11.86	13.96
Phi-2	12.18	11.77	12.19	13.13	12.38	11.19	12.28

Table 14: CommitChronicle ROUGE-2 comparison between ObscuraCoder and comparable Causal LM models, along with frontier models for context.

Model	C	C++	Go	Java	Python	TypeScript	Rust
CausalLM 255M	29.85	30.69	31.79	32.11	31.63	30.77	31.14
ObscuraCoder 255M	30.47 **+0.62**	31.52 **+0.83**	32.07 **+0.28**	32.74 **+0.63**	32.11 **+0.48**	31.23 **+0.46**	31.79 **+0.65**
CausalLM 491M	31.21	31.99	33.39	33.22	32.75	31.64	32.62
ObscuraCoder 491M	31.97 **+0.76**	32.76 **+0.77**	34.01 **+0.62**	33.88 **+0.66**	33.23 **+0.48**	32.34 **+0.70**	33.02 **+0.40**
CausalLM 1.2B	33.06	33.19	35.09	34.49	33.59	33.01	34.07
ObscuraCoder 1.2B	34.71 **+1.65**	34.75 **+1.56**	36.79 **+1.70**	35.88 **+1.39**	35.19 **+1.60**	34.77 **+1.76**	35.26 **+1.19**
CausalLM 2.8B	34.46	34.78	36.21	35.64	34.98	34.36	35.09
ObscuraCoder 2.8B	36.19 **+1.73**	36.38 **+1.60**	37.74 **+1.53**	37.19 **+1.55**	36.82 **+1.84**	35.21 **+0.85**	36.67 **+1.58**
StarCoderBase 1B	34.44	34.94	36.58	35.91	35.48	34.57	35.19
DeepSeekCoder 1.3B	33.94	34.42	34.24	35.40	34.61	33.82	34.52
StarCoderBase 3B	36.54	36.71	37.63	37.32	36.76	36.12	37.11
StarCoder2 3B	36.12	36.73	38.39	37.48	37.01	35.94	37.07
StableCode 3B	37.16	36.13	37.76	36.89	36.45	35.68	36.79
CodeGemma 2B	35.87	34.78	35.68	35.43	35.29	34.86	35.04
Phi-2	33.01	32.98	34.97	34.17	34.23	32.56	33.36

Table 15: CommitChronicle ROUGE-L comparison between ObscuraCoder and comparable Causal LM models, along with frontier models for context.