DOBF: A Deobfuscation Pre-Training Objective for Programming Languages

Marie-Anne Lachaux*
Facebook AI Research

Baptiste Roziere*
Facebook AI Research and Paris-Dauphine University

Marc Szafraniec
Facebook AI Research

Guillaume Lample
Facebook AI Research

ABSTRACT

Recent advances in self-supervised learning have dramatically improved the state of the art on a wide variety of tasks. However, research in language model pre-training has mostly focused on natural languages, and it is unclear whether models like BERT and its variants provide the best pre-training when applied to other modalities, such as source code. In this paper, we introduce a new pre-training objective, DOBF, that leverages the structural aspect of programming languages and pre-trains a model to recover the original version of obfuscated source code. We show that models pre-trained with DOBF significantly outperform existing approaches on multiple downstream tasks. Incidentally, we found that our pre-trained model is able to deobfuscate fully obfuscated source files, and to suggest descriptive variable names. We will release our code on GitHub under a Creative Commons license for non-commercial use with attribution (CC-BY-NC).

ACM Reference Format:
Marie-Anne Lachaux, Baptiste Roziere, Marc Szafraniec, and Guillaume Lample. 2021. DOBF: A Deobfuscation Pre-Training Objective for Programming Languages. In *Proceedings of KDD Workshop on Programming Language Processing (PLP) (PLP).* ACM, New York, NY, USA, 12 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

Model pre-training with self-supervised methods such as BERT [18], RoBERTa [36], XLM [30] or XLNet [49], has become ubiquitous in Natural Language Processing (NLP), and led to significant improvements in many tasks. As in natural language, pre-training was shown to be effective for source code [20, 24, 41]. These studies all leverage the original MLM objective proposed by Devlin et al. [18], which was initially designed for natural languages and does not leverage the particular structure of source code. We propose a new objective based on code obfuscation which forces the model to learn the semantics of the code.

Code obfuscation consists in modifying source code in order to make it harder for humans to understand, or smaller while keeping its behaviour unchanged. Today, it is used to protect intellectual property by preventing people from understanding and modifying the code, to prevent malware detection, and to compress programs (e.g. Javascript code) to reduce network payload sizes. Moreover, C compilers discard variable names, and current rule-based and neural-based decompilers generate obfuscated C code with uninformative variable names [21]. Obfuscators typically apply several transformations to the code. While some operations can be reversed (e.g. dead code injection), the obfuscation of identifier names—renaming every variable, method and class with uninformative names—is irreversible and has a substantial impact on code comprehension [22, 31, 43].

We propose to pre-train a model to revert the obfuscation function, by training a sequence-to-sequence (seq2seq) model to convert obfuscated functions, where names of functions and variables have been replaced by uninformative names, back to their original forms. Suggesting proper variable and function names is a difficult task that requires to understand what the program does. In the context of source code, it is a more sensible, but also a more difficult task than MLM. Indeed, we observe (c.f. Figure 1) that predicting the content of randomly masked tokens is usually quite simple, as it often boils down to making syntax related predictions, or copying variable instances that have not been masked. These simple predictions provide little training signal. Instead, with our new objective, the model needs to learn the semantics of the code to suggest appropriate variable names.

In this paper, we make the following contributions:

- We present DOBF, a new pre-training objective based on deobfuscation, and show its effectiveness on multiple programming languages.
- We show that DOBF significantly outperform MLM (e.g. BERT) on multiple tasks such as code search, code summarization or unsupervised code translation. It also outperforms CodeBERT and GraphCodeBERT on most tasks.
- We show that models trained with DOBF have interesting applications and can be used to suggest appropriate and informative variable names. Besides the model is able to successfully deobfuscate files with fully obfuscated identifiers.

2 RELATED WORK

Code Generation Pre-training. Recent studies showed that pre-training methods developed for natural language processing are also effective for programming languages. For instance, Feng et al. [20] proposed CodeBERT, a RoBERTa-based model trained on source code using the MLM and RTD objectives. With GraphCodeBERT [24],

*Both authors contributed equally to the paper.

PLP, 2021, Singapore

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

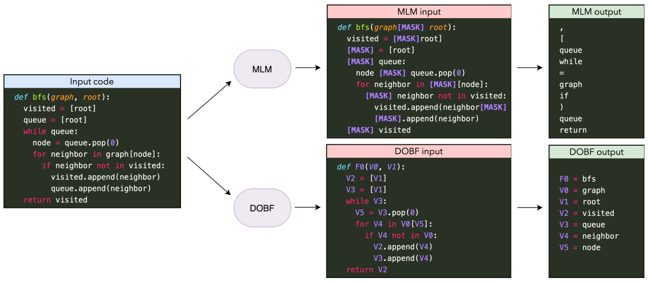

Figure 1: Illustration of the MLM and DOBF objectives. Given an input function, the masked language modeling (MLM) task randomly samples tokens to mask out. With source code, a large fraction of these tokens are related to the language syntax (e.g. commas, parentheses, etc.) that are trivial for the model to predict, and provide a poor training signal. Instead, we propose to obfuscate the code by masking the name of functions and variables, and to train the model to recover the original function by deobfuscating the code (DOBF). When a variable is masked out, we mask all occurrences of this variable with the same mask symbol (e.g. all occurrences of "visited" are replaced by "V0") to prevent the model from copying names. The DOBF objective is more difficult and provides a better learning signal.

the MLM objective is complemented by an edge-prediction objective, in which the model predicts edges in the data flow graph to make the model understand the structure of the code. In Jain et al. [26], a model is trained on javascript code using a contrastive loss ensuring that the representations are robust to some semantic-preserving transformations. They showed that their model performs well on downstream code generation tasks and outperforms previous pre-training approaches. Kanade et al. [27] applied MLM and the next sentence prediction objectives to pre-train models on Python code. More recently, Roziere et al. [41] applied used the MLM, denoising and back-translation objectives to train multilingual embeddings of source code and translate between Python, Java, and C++ in a fully unsupervised way. In this paper, we propose to use a code-specific objective to better pre-train models designed to be fine-tuned on code generation tasks: code deobfuscation. Machine learning is frequently used on tasks involving programming languages, including code completion [28, 33, 35, 42], bug detection and code repair [15, 38, 46], code summarization [8, 25], clone detection [2, 47, 48], code search [13, 23] and code translation [14, 41]. Most of these tasks can benefit from pre-trained models that capture the semantics of the code.

Code deobfuscation. Empirical studies show that naming conventions and the use of informative identifier names make code more understandable, easier to maintain and lead to fewer bugs [12, 34, 43]. It motivated other works studying deobfuscation of identifier names and identifier name proposal using n-grams [4, 5], probabilistic models [7, 11, 40, 44], and recurrent neural networks [10, 29]. Alon et al. [7] extract features from Abstract Syntax Tree (AST) paths and train a Conditional Random Field to predict variable and method names, and infer types for several languages. DIRE [29] uses a commercial decompiler to obtain C code with uninformative identifier names from binaries. They also use AST features, which go through a Graph Neural Network trained jointly with a LSTM model on the sequence of C tokens to retrieve relevant identifier names. More recently, David et al. [17] used a transformer together with augmented representations obtained from static analysis to infer procedure names in stripped binary files. These models are already used to understand obfuscated and compiled source code. However, none of these studies investigated the use of deobfuscation for model pre-training.

3 MODEL

3.1 MLM for Programming Languages

A countless number of pre-training objectives have been introduced in the literature [16, 18, 19, 32, 36]. Most of them rely on hyper-parameters and seemingly arbitrary decisions (Should we mask individual tokens or spans? Which fraction of them? What do we do with masked out tokens? etc.). These choices are typically based on intuition and validated empirically on natural language processing tasks. However, source code is much more structured than natural language, which makes predicting masked tokens much easier for programming languages.

The first row in Figure 1 shows an example of input / output for the MLM objective. We can see that the majority of tokens are composed of Python keywords or symbols related to syntax: `, [ while = if ) return`. These symbols are easy to recover, and a model will quickly learn to predict them with perfect accuracy. Retrieving the obfuscated `graph` token is also relatively simple: the model only needs to retrieve the most relevant variable in the scope. More generally, retrieving an identifier name is often easy when given its full context, including its definition and usages. We suspect that the MLM objective is too simple in programming languages.

3.2 Deobfuscation Objective

Instead of MLM, we propose a new pre-training objective, DOBF, that leverages the particular structure of programming languages. We obfuscate code snippets by replacing class, function and variable names with special tokens, and train a model to recover the original names. When an identifier is selected, all of its instances in the code are replaced by the same special token. This differs from MLM where the name of a variable can appear multiple times while being masked a single time. For instance, in Figure 1, DOBF will replace the two occurrences of `node` by the same symbol `V5`, while MLM will only mask one of these occurrences. As a result, the fraction of meaningful tokens masked by the objective is language independent: for more verbose languages (e.g. Java), the less informative syntax-related tokens will not be masked out by the DOBF objective.

Each identifier is replaced with probability $p_{obf} \in [0, 1]$. We ensure that the original input is modified: if no identifier is replaced, we draw a random one to obfuscate. When $p_{obf} = 0$, we always obfuscate exactly one random identifier in the input. When $p_{obf} = 1$, we obfuscate all the identifiers defined in the file. We ensure that the obfuscated code has the same behavior as the original. The second row in Figure 1 shows an example of obfuscated code with $p_{obf} = 1$, where we obfuscate a function `bfs` which implements a breadth-first search. The function `append` is not obfuscated as it is a standard Python function not defined in the file. The model is given the obfuscated code as input and has to output a dictionary mapping special tokens to their initial values.

Finding informative names for obfuscated identifiers requires the model to learn a deep understanding of code semantics, which is desirable for a pre-training task. MLM will mask only some of the occurrences of the identifiers and leave the other ones unchanged so that the model can simply copy identifier names. In Figure 1, with MLM masking, the model can simply notice that a variable named `queue` is called on the fourth line. Since the variable is not defined, the model can easily guess that `queue` has to be defined on the third line, and infer the value of the corresponding `[MASK]` token. With the deobfuscation objective, the model needs to analyze code patterns and understand their semantics to infer that, since its elements are popped with `.pop(0)`, the variable `V3` implements a queue. If its elements were popped with `.pop()`, our model would name it `stack` instead of `queue` (c.f. Figure 7 in the appendix).

We train a seq2seq model with attention, composed of an encoder and a decoder using a transformer architecture [45] to map an obfuscated code into a dictionary represented as a sequence of tokens[1]. At inference time, the model is able to suggest meaningful class, function and variable names for a piece of code with an arbitrary number of obfuscated identifiers.

4 EXPERIMENTS

We train DOBF with the deobfuscation objective. First, we evaluate our model on two straightforward deobfuscation applications. Then, we show its performance on multiple downstream tasks.

4.1 Deobfuscation

We evaluate our model on two applications of the deobfuscation task: when $p_{obf} = 0$ (the model has to retrieve a single identifier name), and $p_{obf} = 1$ (the model has to retrieve all the identifier names). Retrieving a single identifier is relevant to suggest relevant variable names (in code editors for instance) while the performance when $p_{obf} = 1$ is relevant when retrieving all the variable names in a decompiled or obfuscated file. We evaluate the ability of our model to retrieve identifier names from the original non-obfuscated code. We report the accuracy, which is the percentage of recovered tokens that exactly match the ground truth. Following [5–7, 9], we also report the *subtoken score*, a more flexible metric which computes the precision, recall, and F1 scores for retrieving the original case-insensitive subtokens.

4.2 Fine-tuning on downstream tasks

In order to evaluate DOBF as a pre-training model, we fine-tune DOBF on TransCoder and on three tasks from CodeXGLUE [1], a benchmark for programming languages. We only consider the Java and Python tasks with an encoder in the model architecture for which the training, validation, and test sets are publicly available.

CodeXGLUE Clone Detection This task is a binary classification problem where the model has to predict whether two code snippets are semantically equivalent. It is evaluated using the F1 score. This task is available in Java.

[1]In the obfuscated example given in Figure 1, the model is trained to generate the sequence: `FUNC_0 bfs | VAR_0 graph | VAR_1 root | VAR_2 visited | VAR_3 queue | VAR_4 neighbor | VAR_5 node`.

CodeXGLUE Code Summarization Given a code snippet, the model is trained to generate the corresponding documentation in natural language. The architecture is a sequence-to-sequence transformer model evaluated using BLEU score [39]. The dataset includes both Java and Python source code.

CodeXGLUE NL Code Search Given a code search query in natural language the model has to retrieve the most semantically related code within a collection of code snippets. This is a ranking problem evaluated using the Mean Reciprocal Rank (MRR) metric.

TransCoder TransCoder [41] is an unsupervised machine translation model which translates functions and methods between C++, Java, and Python. A single seq2seq model is trained for all languages. In the original work, TransCoder is pre-trained with MLM, and trained with denoising auto-encoding and back-translation. TransCoder is evaluated using the Computational Accuracy metric, which computes the percentage of correct solutions according to series of unit tests. We only consider a single model output (CA@1), with beam sizes of 1 and 10.

4.3 Experimental details

Model Architecture We train two models with different sizes in order to provide fair comparisons to our baselines (CodeBERT and TransCoder). We train one model with 12 layers, 12 attention heads, and a hidden dimensionality of 768 and one model with 6 layers, 8 attention heads, and a hidden dimensionality of 1024.

Training dataset As in Roziere et al. [41], we use the GitHub public dataset available on Google BigQuery and select all Python and Java files within the available projects. Following Lopes et al. [37] and Allamanis [3], we remove duplicate files. We also ensure that each fork belongs to the same split as its source repository. We show some statistics about this dataset in Table 3.

Training details We train DOBF to translate obfuscated files into lists of identifier names. We try different initialization schemes: training from scratch and with a Python-Java MLM following Roziere et al. [41]. We train DOBF with three different obfuscation probability parameters: $p_{obf} \in \{0, 0.5, 1\}$. For each p_{obf} value, we train models with multiple initial learning rates ranging from 10^{-4} to 3.10^{-4} and select the best one using the average subtoken F1 score computed on the validation dataset.

Fine-tuning details Depending on the fine-tuning tasks, we consider different model architectures: seq2seq models with encoder and decoder, architectures with two encoders or a single encoder. In all cases, we initialize the encoders of these models with the encoder of DOBF and fine-tune all parameters. For fair comparison, we rerun all baselines, and train models with the same architectures, batch sizes and optimizers as in the original papers.

4.4 Results on deobfuscation

In Table 1, we evaluate the ability of our model to recover identifier names, either when only one identifier is obfuscated ($p_{obf} = 0$) or when all identifiers are obfuscated ($p_{obf} = 1$), for models trained with $p_{obf} \in \{0, 0.5, 1\}$. Even when evaluating with $p_{obf} = 0$, training with $p_{obf} = 0$ is less efficient than $p_{obf} = 0.5$ since the model is only trained to generate a single variable for each input sequence. Training with $p_{obf} = 0.5$ is a more difficult task that requires the model to learn and understand more about code semantics. Forcing

```python
def FUNC_0(VAR_0, VAR_1):
    VAR_2 = [VAR_1]
    VAR_3 = [VAR_1]
    while VAR_3:
        VAR_4 = VAR_3.pop(0)
        for VAR_5 in VAR_0[VAR_4]:
            if (VAR_5 not in VAR_2):
                VAR_2.add(VAR_5)
                VAR_3.append(VAR_5)
    return VAR_2
```

```python
def bfs(graph, start):
    visited = [start]
    queue = [start]
    while queue:
        node = queue.pop(0)
        for neighbor in graph[node]:
            if (neighbor not in visited):
                visited.add(neighbor)
                queue.append(neighbor)
    return visited
```

Figure 2: Full deobfuscation of a breadth-first-search function by DOBF. The code on top has been fully obfuscated. The code on the bottom was recovered using DOBF by replacing the function name and every variable name using the generated dictionary. DOBF is able to suggest relevant function and variable names. It makes the code much more readable and easier to understand.

Table 1: Results on partial and full deobfuscation. Token accuracy and subtoken F1 score of DOBF evaluated with $p_{obf} = 0$ (i.e. name proposal, where a single token is obfuscated) and $p_{obf} = 1$ (i.e. full deobfuscation, where all tokens are obfuscated). We consider models trained with different obfuscation probabilities p_{obf}. $\text{DOBF}_{0.5}$ performs well for both tasks, and it even performs better than DOBF_0 for Identifier Name Proposal. DOBF_0 and DOBF_1 perform poorly when evaluated on other p_{obf} parameters. Pre-training DOBF with MLM further improves the performance.

	Eval $p_{\text{obf}} = 0$		Eval $p_{\text{obf}} = 1$	
	Acc	F1	Acc	F1
DOBF_0	56.3	68.0	0.4	0.9
$\text{DOBF}_{0.5}$	61.1	71.2	41.8	54.8
DOBF_1	18.1	27.0	45.6	58.1
$\text{MLM+DOBF}_{0.5}$	**67.6**	**76.3**	45.7	58.0
MLM+DOBF_1	20.0	28.3	**49.7**	**61.1**

the model to understand the structure of the code may be useful even when testing with $p_{obf} = 0$, as some identifier names cannot be guessed only from the names of other identifiers. When DOBF has to recover a fully obfuscated function, it obtains the best accuracy when trained with $p_{obf} = 1$. It manages to recover 45.6% of the initial identifier names. We also observe that, for every configuration, pre-training DOBF with MLM improves the performance.

Figure 2 shows an example of a fully obfuscated function recovered by our model. DOBF successfully manages to understand the purpose of the function and to predict appropriate variable names. Figure 3 shows examples of function name proposal by DOBF for functions implementing matrix operations in Python. We observe that DOBF manages to identify the key tokens and to properly infer the purpose of similar but very different functions. Figures 3, 4, 5, and 6 in the appendix show additional examples of function name proposals by DOBF in Java and Python. Figure 7 shows additional examples where we show that DOBF also leverages non-obfuscated identifier names to understand the meaning of input functions. Figures 8 and 9 in the appendix show examples of deobfuscation of fully obfuscated Python code snippets using DOBF. It is able to understand the semantics and purposes of a variety of obfuscated classes and functions, including a LSTM cell.

4.5 Results on downstream tasks

For fine-tuning, we considered models pre-trained with $p_{obf} = 0.5$ and $p_{obf} = 1$. Since they gave very similar results on downstream tasks, we only use models pre-trained with $p_{obf} = 0.5$ in the rest of the paper. As baselines, we consider a randomly initialized model and a model pre-trained with MLM only. For CodeXGLUE tasks, we also consider CodeBERT and GraphCodeBERT as baselines. We compare results for DOBF trained from scratch and DOBF initialized with MLM (MLM+DOBF), and report results in Table 2.

Table 2: Results on downstream tasks for different pre-training configurations. Models pre-trained with MLM and DOBF significantly outperform both CodeBERT and models trained with MLM only. MLM+DOBF outperforms CodeBERT by 7% on natural language code search (NLCS), and MLM by 6% in Java → Python computational accuracy. It also beats CodeBERT on every task except Clone Detection, on which CodeBERT scores much higher than our MLM. GraphCodeBERT only beats our model on python summarization and Python to Java translation by a shallow margin and is below on other tasks. The tasks where MLM provides large improvements over the transformer baseline (first row) are also the tasks where DOBF provides the largest gains (e.g. clone detection, natural language code search, and unsupervised translation).

	Clone Det (F1 score)	Sum Java (BLEU)	Sum Py (BLEU)	NLCS (MRR)	Py→Ja (CA@1)		Ja→Py (CA@1)	
					k=1	k=10	k=1	k=10
Transformer	88.14	16.58	16.43	0.025	37.6	38.9	31.8	42.1
CodeBERT	96.50	18.25	18.22	0.315	-	-	-	-
GraphCodeBERT	96.38	18.78	**18.51**	0.377	-	-	-	-
MLM	91.89	18.59	17.95	0.308	40.3	42.2	44.7	46.6
DOBF	**96.52**	18.19	17.51	0.272	38.9	**45.7**	44.7	46.4
MLM+DOBF	95.87	**19.05**	18.24	**0.383**	**43.5**	44.9	**49.2**	**52.5**

DOBF trained from scratch and DOBF pre-trained with MLM obtained state-of-the-art results on all downstream tasks, outperforming CodeBERT and MLM. The deobfuscation objective is already effective as a pre-training task: it leads to results comparable to MLM on most tasks and is much more effective on clone detection. The MLM+DOBF model outperforms MLM on all downstream tasks, the major improvement being for NL code search, which is also the task that benefited the most from MLM pretraining. For TransCoder, MLM+DOBF increases the computational accuracy of the MLM model by 2.7% when translating from Python to Java, and by 5.9% when translating from Java to Python with beam size 10. Also, MLM+DOBF beats CodeBERT by a wide margin on NL code search and code summarization, showing that programming language data aligned with natural language is not necessary to train an effective model on those tasks. MLM+DOBF yields higher scores than both DOBF and MLM on most tasks, showing that MLM and DOBF are complementary.

5 CONCLUSION

In this paper, we introduce a new deobfuscation objective and show that it can be used for three purposes: recover fully obfuscated code, suggest relevant identifier names, and pre-train transformer models for programming language related tasks. Although it does not require any parallel corpora of source code aligned to natural language, DOBF outperforms previous methods on multiple downstream tasks, including clone detection, code summarization, natural language code search, and unsupervised code translation. These results show that DOBF leverages the particular structure of source code to add noise to the input sequence in a particularly effective way. Other noise functions or surrogate objectives adapted to source code may improve the performance further.

REFERENCES

[1] Codexglue: An open challenge for code intelligence. arXiv, 2020.
[2] Ain, Q. U., Butt, W. H., Anwar, M. W., Azam, F., and Maqbool, B. A systematic review on code clone detection. IEEE Access, 7:86121–86144, 2019.
[3] Allamanis, M. The adverse effects of code duplication in machine learning models of code. In Proceedings of the 2019 ACM SIGPLAN International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software, pp. 143–153, 2019.
[4] Allamanis, M., Barr, E. T., Bird, C., and Sutton, C. Learning natural coding conventions. In Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering, pp. 281–293, 2014.
[5] Allamanis, M., Barr, E. T., Bird, C., and Sutton, C. Suggesting accurate method and class names. In Proceedings of the 2015 10th Joint Meeting on Foundations of Software Engineering, pp. 38–49, 2015.
[6] Allamanis, M., Peng, H., and Sutton, C. A convolutional attention network for extreme summarization of source code. In International conference on machine learning, pp. 2091–2100, 2016.
[7] Alon, U., Zilberstein, M., Levy, O., and Yahav, E. A general path-based representation for predicting program properties. ACM SIGPLAN Notices, 53(4):404–419, 2018.
[8] Alon, U., Brody, S., Levy, O., and Yahav, E. code2seq: Generating sequences from structured representations of code. ICLR, 2019.
[9] Alon, U., Zilberstein, M., Levy, O., and Yahav, E. code2vec: Learning distributed representations of code. Proceedings of the ACM on Programming Languages, 3(POPL):1–29, 2019.
[10] Bavishi, R., Pradel, M., and Sen, K. Context2name: A deep learning-based approach to infer natural variable names from usage contexts. arXiv preprint arXiv:1809.05193, 2018.
[11] Bichsel, B., Raychev, V., Tsankov, P., and Vechev, M. Statistical deobfuscation of android applications. In Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, pp. 343–355, 2016.
[12] Butler, S., Wermelinger, M., Yu, Y., and Sharp, H. Relating identifier naming flaws and code quality: An empirical study. In 2009 16th Working Conference on Reverse Engineering, pp. 31–35. IEEE, 2009.
[13] Cambronero, J., Li, H., Kim, S., Sen, K., and Chandra, S. When deep learning met code search. In Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, pp. 964–974, 2019.
[14] Chen, X., Liu, C., and Song, D. Tree-to-tree neural networks for program translation. In Advances in neural information processing systems, pp. 2547–2557, 2018.
[15] Chen, Z., Kommrusch, S. J., Tufano, M., Pouchet, L.-N., Poshyvanyk, D., and Monperrus, M. Sequencer: Sequence-to-sequence learning for end-to-end program repair. IEEE Transactions on Software Engineering, 2019.
[16] Clark, K., Luong, M.-T., Le, Q. V., and Manning, C. D. Electra: Pre-training text encoders as discriminators rather than generators. arXiv preprint arXiv:2003.10555, 2020.
[17] David, Y., Alon, U., and Yahav, E. Neural reverse engineering of stripped binaries using augmented control flow graphs. Proceedings of the ACM on Programming Languages, 4(OOPSLA):1–28, 2020.
[18] Devlin, J., Chang, M.-W., Lee, K., and Toutanova, K. Bert: Pre-training of deep bidirectional transformers for language understanding. CoRR, abs/1810.04805, 2018.
[19] Dong, L., Yang, N., Wang, W., Wei, F., Liu, X., Wang, Y., Gao, J., Zhou, M., and Hon, H.-W. Unified language model pre-training for natural language understanding and generation. arXiv preprint arXiv:1905.03197, 2019.
[20] Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., et al. Codebert: A pre-trained model for programming and natural languages. arXiv preprint arXiv:2002.08155, 2020.
[21] Fu, C., Chen, H., Liu, H., Chen, X., Tian, Y., Koushanfar, F., and Zhao, J. Coda: An end-to-end neural program decompiler. In Advances in Neural Information Processing Systems, pp. 3703–3714, 2019.
[22] Gellenbeck, E. M. and Cook, C. R. An investigation of procedure and variable names as beacons during program comprehension. In Empirical studies of programmers: Fourth workshop, pp. 65–81. Ablex Publishing, Norwood, NJ, 1991.
[23] Gu, X., Zhang, H., and Kim, S. Deep code search. In 2018 IEEE/ACM 40th International Conference on Software Engineering (ICSE), pp. 933–944. IEEE, 2018.
[24] Guo, D., Ren, S., Lu, S., Feng, Z., Tang, D., Liu, S., Zhou, L., Duan, N., Svyatkovskiy, A., Fu, S., et al. Graphcodebert: Pre-training code representations with data flow. arXiv preprint arXiv:2009.08366, 2020.
[25] Hu, X., Li, G., Xia, X., Lo, D., and Jin, Z. Deep code comment generation. In Proceedings of the 26th Conference on Program Comprehension, pp. 200–210, 2018.
[26] Jain, P., Jain, A., Zhang, T., Abbeel, P., Gonzalez, J. E., and Stoica, I. Contrastive code representation learning. arXiv preprint arXiv:2007.04973, 2020.
[27] Kanade, A., Maniatis, P., Balakrishnan, G., and Shi, K. Learning and evaluating contextual embedding of source code. In International Conference on Machine Learning, pp. 5110–5121. PMLR, 2020.
[28] Kim, S., Zhao, J., Tian, Y., and Chandra, S. Code prediction by feeding trees to transformers. arXiv preprint arXiv:2003.13848, 2020.
[29] Lacomis, J., Yin, P., Schwartz, E., Allamanis, M., Le Goues, C., Neubig, G., and Vasilescu, B. Dire: A neural approach to decompiled identifier naming. In 2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 628–639. IEEE, 2019.
[30] Lample, G. and Conneau, A. Cross-lingual language model pretraining. arXiv preprint arXiv:1901.07291, 2019.
[31] Lawrie, D., Morrell, C., Feild, H., and Binkley, D. What's in a name? a study of identifiers. In 14th IEEE International Conference on Program Comprehension (ICPC'06), pp. 3–12. IEEE, 2006.
[32] Lewis, M., Liu, Y., Goyal, N., Ghazvininejad, M., Mohamed, A., Levy, O., Stoyanov, V., and Zettlemoyer, L. Bart: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. arXiv preprint arXiv:1910.13461, 2019.
[33] Li, J., Wang, Y., Lyu, M. R., and King, I. Code completion with neural attention and pointer networks. IJCAI, 2018.
[34] Liblit, B., Begel, A., and Sweetser, E. Cognitive perspectives on the role of naming in computer programs. In PPIG, pp. 11, 2006.
[35] Liu, F., Li, G., Wei, B., Xia, X., Fu, Z., and Jin, Z. A self-attentional neural architecture for code completion with multi-task learning. In Proceedings of the 28th International Conference on Program Comprehension, pp. 37–47, 2020.
[36] Liu, Y., Ott, M., Goyal, N., Du, J., Joshi, M., Chen, D., Levy, O., Lewis, M., Zettlemoyer, L., and Stoyanov, V. Roberta: A robustly optimized bert pretraining approach. arXiv preprint arXiv:1907.11692, 2019.
[37] Lopes, C. V., Maj, P., Martins, P., Saini, V., Yang, D., Zitny, J., Sajnani, H., and Vitek, J. Déjàvu: a map of code duplicates on github. Proceedings of the ACM on Programming Languages, 1(OOPSLA):1–28, 2017.
[38] Murali, V., Gross, L., Qian, R., and Chandra, S. Industry-scale ir-based bug localization: A perspective from facebook. arXiv preprint arXiv:2010.09977, 2020.
[39] Papineni, K., Roukos, S., Ward, T., and Zhu, W.-J. Bleu: a method for automatic evaluation of machine translation. In Proceedings of the 40th annual meeting on association for computational linguistics, pp. 311–318. Association for Computational Linguistics, 2002.
[40] Raychev, V., Vechev, M., and Krause, A. Predicting program properties from" big code". ACM SIGPLAN Notices, 50(1):111–124, 2015.
[41] Roziere, B., Lachaux, M.-A., Chanussot, L., and Lample, G. Unsupervised translation of programming languages. Advances in Neural Information Processing Systems, 33, 2020.
[42] Svyatkovskoy, A., Lee, S., Hadjitofi, A., Riechert, M., Franco, J., and Allamanis, M. Fast and memory-efficient neural code completion. arXiv preprint arXiv:2004.13651, 2020.
[43] Takang, A. A., Grubb, P. A., and Macredie, R. D. The effects of comments and identifier names on program comprehensibility: an experimental investigation. J. Prog. Lang., 4(3):143–167, 1996.
[44] Vasilescu, B., Casalnuovo, C., and Devanbu, P. Recovering clear, natural identifiers from obfuscated js names. In Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering, pp. 683–693, 2017.
[45] Vaswani, A., Shazeer, N., Parmar, N., Uszkoreit, J., Jones, L., Gomez, A. N., Kaiser, Ł., and Polosukhin, I. Attention is all you need. In Advances in neural information processing systems, pp. 5998–6008, 2017.
[46] Wang, K., Singh, R., and Su, Z. Dynamic neural program embedding for program repair. arXiv preprint arXiv:1711.07163, 2017.
[47] Wang, W., Li, G., Ma, B., Xia, X., and Jin, Z. Detecting code clones with graph neural network and flow-augmented abstract syntax tree. In 2020 IEEE 27th International Conference on Software Analysis, Evolution and Reengineering (SANER), pp. 261–271. IEEE, 2020.
[48] Wei, H. and Li, M. Supervised deep features for software functional clone detection by exploiting lexical and syntactical information in source code. In IJCAI, pp. 3034–3040, 2017.
[49] Yang, Z., Dai, Z., Yang, Y., Carbonell, J., Salakhutdinov, R. R., and Le, Q. V. Xlnet: Generalized autoregressive pretraining for language understanding. In Advances in neural information processing systems, pp. 5753–5763, 2019.

Input Code	Function Name Proposals	
``def FUNC_0 (m1, m2):``` ```assert m1.shape == m2.shape``` ```n, m = m1.shape``` ```res = [[0 for _ in range(m)] for _ in range(n)]``` ```for i in range(n):``` ```for j in range(m):``` ```res[i][j] = m1[i][j] + m2[i][j]``` ```return res```	matrix_add matrixAdd matrixadd matrix_sum matrix_addition	25.9% 22.5% 18.8% 16.7% 16.1%
``def FUNC_0 (m1, m2):``` ```assert m1.shape == m2.shape``` ```n, m = m1.shape``` ```res = [[0 for _ in range(m)] for _ in range(n)]``` ```for i in range(n):``` ```for j in range(m):``` ```res[i][j] = m1[i][j] - m2[i][j]``` ```return res```	matrix_sub matrix_subtract matrix_subtraction sub sub_matrix	26.1% 21.5% 19.7% 17.6% 15.0%
``def FUNC_0 (matrix):``` ```n, _ = matrix.shape``` ```for i in range(n):``` ```for j in range(i,n):``` ```matrix[i][j], matrix[j][i] = \``` ```matrix[j][i], matrix[i][j]```	transpose rotate rotate_matrix symmetric rotate_matrix_by_row	36.7% 29.5% 17.1% 8.9% 7.7%
``def FUNC_0 (m1, m2):``` ```n1, m1 = m1.shape``` ```n2, m2 = m2.shape``` ```assert n2 == m1``` ```res = [[0 for _ in range(m2)] for _ in range(n1)]``` ```for i in range(n1):``` ```for j in range(m2):``` ```res[i][j] = sum([m1[i][k] * m2[k][j]``` ```for k in range(n2)])``` ```return res```	matrix_product mat_mult matmul_mat matprod matrixProduct	28.8% 23.8% 17.0% 16.0% 14.4%

Figure 3: Additional examples of function name proposals for matrix operations in Python. DOBF is able to find the right name for each matrix operation, showing that it learned to attend to the most important parts of the code. Even when the function only differs by one token (e.g. a subtraction instead of an addition operator), DOBF successfully and confidently (c.f. scores) understands the semantics of the function and its purpose.

Table 3: Dataset statistics.

	Java	Python
All - Size	26 GB	19 GB
All - Nb files	7.9M	3.6M
Av. nb of tokens / file	718	1245
Av. nb of identifiers / file	25.9	41.8

Input Code | **Proposed Function Name**

```java
public static void FUNC_0 (String path){
  try {
    Files.delete(path);
  }
  catch (Exception e) {
    System.err.println("Error deleting file " + path);
  }
}
```

Proposed Function Name	
deleteFile	48.3%
remove	16.9%
DeleteFile	13.2%
removeFile	13.1%
deleteFileQuietly	8.4%

```java
public static void FUNC_0 (String path){
  if (!Files.exists(path)) {
    Files.createDirectories(path);
  }
}
```

Proposed Function Name	
createDir	23.5%
createDirectory	20.9%
createDirIfNotExists	20.8%
ensureDirectoryExists	18.5%
createDirectoryIfNotExists	16.3%

```java
public static List<Pair<String, Double>> FUNC_0 (List<String> list1,
                                                 List<Double> list2)
{
  return IntStream.range(0, Math.min(list1.size(), list2.size()))
                .mapToObj(i -> new Pair<>(list1.get(i), list2.get(i)))
                .collect(Collectors.toList());
}
```

Proposed Function Name	
zip	28.6%
intersect	20.0%
combine	17.9%
merge	17.5%
intersection	16.0%

```java
public static int FUNC_0 (int n){
  int a = 0, b = 1;
  int tmp;
  for (int i = 0; i < n; i ++){
      tmp = a + b;
      a = b;
      b = tmp;
  }
  return a;
}
```

Proposed Function Name	
fib	41.5%
fibonacci	36.6%
fibon	9.1%
fibo	8.8%
fibonacci_series	4.0%

```java
public static float FUNC_0 (List<Float> vec1,
                            List<Float> vec2) {
  float size = vec1.size();
  assert size == vec2.size();
  float result = 0.0f;
  for (int i = 0; i < size; i++) {
    result += vec1.get(i) * vec2.get(i);
  }
  return result;
}
```

Proposed Function Name	
dotProduct	40.9%
dot	23.9%
dot_product	16.5%
dotproduct	10.5%
inner	8.3%

Figure 4: Examples of name proposal in Java. DOBF is able to suggest relevant function names for a variety of Java methods and demonstrates its ability to understand the semantics of the code. In the first two examples, the first element in the beam shows that it is able to select relevant names in the context to find a function name: it uses `Files.delete` and `Files.createDirectories` to suggest the tokens `deleteFile` and `createDir`. DOBF finds relevant names for Java methods without copying any part of the other tokens. For example for the third method combining two lists as in the python `zip` function, for the fourth method which computes the n-th element of the Fibonacci series and for the last method which computes the dot product between two vectors.

Input Code	Proposals for Highlighted Identifiers	
`def FUNC_0 (name):` `  return os.environ[name]`	get_env get_envvar env getenv get_env_variable	25.3% 19.3% 19.2% 18.5% 17.7%
`def FUNC_0 (l):` `  return list(set(l))`	unique remove_duplicates removeDuplicates uniquify unique_items	24.8% 23.8% 18.8% 18.7% 13.8%
`def FUNC_0 (path):` `  with gzip.open(path, 'rb') as f:` `     content = f.read()` `  return content`	read_gzip_file read_gzip ungzip gzip_content gzip_read	22.9% 22.1% 20.8% 18.2% 16.0%
`def FUNC_0 (n):` `  v = [True for i in range(n + 1)]` `  p = 2` `  while (p * p <= n):` `    if (v[p] == True):` `      for i in range(p * 2, n + 1, p):` `        v[i] = False` `      p += 1` `  v[0]= False` `  v[1]= False` `  return [p for p in range(n+1) if v[p]]`	sieve prime_sieve sieve_of_eratosthenes primes eratosthenes	36.1% 18.5% 15.5% 15.3% 14.5%
`def f(n):` `    VAR_0 = [True for i in range(n + 1)]` `    p = 2` `    while (p * p <= n):` `        if ( VAR_0 [p] == True):` `            for i in range(p * 2, n + 1, p):` `                VAR_0 [i] = False` `        p += 1` `    VAR_0 [0]= False` `    VAR_0 [1]= False` `    return [p for p in range(n+1) if VAR_0 [p]]`	prime l isPrime a primes	30.6% 20.5% 18.0% 16.4% 14.6%

Figure 5: Examples of name proposal in Python. Our model trained with DOBF goes well beyond copying tokens from the context. For instance, in the first example, it understands that this function is used to get environment variables. In the second example, it proposes names related to what this function actually does (removing duplicates in a list) instead of the individual operations it uses (converting to set and then to list). The last two rows show proposals for two different identifiers in a function computing the list of prime numbers below n using the sieve of Eratosthenes. The proposals for the function name are all relevant, and the third one names exactly the algorithm which is used. The variable `v` is a list of booleans. At the end of the algorithm, `v[i]` is true if and only if `i` is prime. The proposed names `prime` and `isPrime` are very relevant as they describe what the list contains. Although `l` and `a` are not very informative, they indicate that the variable is a list or an array.

Input Code | Proposed Function Name

```python
def FUNC_0 (v1, v2):
  assert len(v1) == len(v2)
  return [a * b for a, b in zip(v1, v2)]
```

multiply_lists	28.7%
multiply_list	23.5%
multiply	18.1%
multiply_vectors	14.9%
mul	14.8%

```python
def FUNC_0 (v1, v2):
  assert len(v1) == len(v2)
  return sum([a * b for a, b in zip(v1, v2)])
```

dotproduct	34.8%
dot_product	19.2%
dotProduct	18.1%
dot	15.6%
multiply_by_addition	12.3%

```python
def FUNC_0 (v1, v2):
  assert len(v1) == len(v2)
  return [a ^ b for a, b in zip(v1, v2)]
```

xor	62.9%
XOR	12.8%
vector_xor	10.8%
xors	7.4%
xor_lists	6.1%

```python
def FUNC_0 (v1, v2):
  assert len(v1) == len(v2)
  return [a ** b for a, b in zip(v1, v2)]
```

power	29.8%
list_power	20.9%
lcm	19.9%
power_list	15.1%
powersum	14.3%

```python
def FUNC_0 (v1, v2):
  assert len(v1) == len(v2)
  return [a + b for a, b in zip(v1, v2)]
```

add_lists	27.0%
add	22.9%
sum_lists	17.9%
list_concat	17.7%
list_add	14.5%

```python
def FUNC_0 (v1, v2):
  assert len(v1) == len(v2)
  return [a - b for a, b in zip(v1, v2)]
```

minus	30.4%
subtract	29.8%
difference	14.1%
subtract_lists	13.3%
substract	12.4%

Figure 6: Examples of function name proposal in Python using DOBF. DOBF is able to identify the key tokens in each function, to properly infer its purpose, and to suggest appropriate names along with a confidence score. In particular, even though the first two code snippets are very similar in terms of edit distance, they implement very different functions and DOBF is able to name them appropriately.

BFS Implementation

```python
def FUNC_0 (graph, node):
  visited = [node]
  VAR_0 = [node]
  while VAR_0 :
    s = VAR_0 .pop(0)
    for neighbour in graph[s]:
      if neighbour not in visited:
        visited.add(neighbour)
        VAR_0 .append(neighbour)
  return visited
```

FUNC_0 bfs | VAR_0 queue

DFS Implementation

```python
def FUNC_0 (graph, node):
  visited = [node]
  VAR_0 = [node]
  while VAR_0 :
    s = VAR_0 .pop()
    for neighbour in graph[s]:
      if neighbour not in visited:
        visited.add(neighbour)
        VAR_0 .append(neighbour)
  return visited
```

FUNC_0 dfs | VAR_0 stack

DFS with Erroneous Variable Name

```python
def FUNC_0 (graph, node):
  visited = [node]
  queue = [node]
  while queue:
    s = queue.pop()
    for neighbour in graph[s]:
      if neighbour not in visited:
        visited.append(neighbour)
        queue.append(neighbour)
  return visited
```

FUNC_0 bfs

Figure 7: Deobfuscation on graph traversal functions. These three functions perform graph traversals. The only difference between the first and the second function is that the first uses a queue to select the next element (`.pop(0)`) while the second uses a stack (`.pop()`). The first function implements a breadth-first search (bfs) in the graph and the second implements a depth-first search (dfs). DOBF is able to find the right function and variable names in each case. In the last function, we replaced the anonymized `VAR_0` variable with `queue` in the implementation of depth-first search. This erroneous information leads DOBF to believe that this function performs breadth-first search. It shows that, just like human programmers, DOBF uses the names of the other variables to understand programs and choose relevant identifier names. When working on code with misleading identifier names, it is often preferable to obfuscate several identifiers.

Obfuscated Code

```
class CLASS_0(nn.Module):

    def __init__(VAR_0, VAR_1, VAR_2, VAR_3):
        super(CLASS_0, VAR_0).__init__()
        VAR_0.VAR_1 = VAR_1
        VAR_0.VAR_2 = VAR_2
        VAR_0.VAR_4 = nn.Linear(VAR_1, (4 * VAR_2), bias=VAR_3)
        VAR_0.VAR_5 = nn.Linear(VAR_2, (4 * VAR_2), bias=VAR_3)
        VAR_0.FUNC_0()

    def FUNC_0(VAR_6):
        VAR_7 = (1.0 / math.sqrt(VAR_6.VAR_8))
        for VAR_9 in VAR_6.VAR_10():
            VAR_9.data.uniform_((- VAR_7), VAR_7)

    def FUNC_1(VAR_11, VAR_12, VAR_13):
        (VAR_14, VAR_15) = VAR_13
        VAR_14 = VAR_14.view(VAR_14.size(1), (- 1))
        VAR_15 = VAR_15.view(VAR_15.size(1), (- 1))
        VAR_12 = VAR_12.view(VAR_12.size(1), (- 1))
        VAR_16 = (VAR_11.VAR_4(VAR_12) + VAR_11.VAR_5(VAR_14))
        VAR_17 = VAR_16[:, :(3 * VAR_11.VAR_8)].sigmoid()
        VAR_18 = VAR_16[:, (3 * VAR_11.VAR_8):].tanh()
        VAR_19 = VAR_17[:, :VAR_11.VAR_8]
        VAR_20 = VAR_17[:, VAR_11.VAR_8:(2 * VAR_11.VAR_8)]
        VAR_21 = VAR_17[:, (- VAR_11.VAR_8):]
        VAR_22 = (th.mul(VAR_15, VAR_20) + th.mul(VAR_19, VAR_18))
        VAR_23 = th.mul(VAR_21, VAR_22.tanh())
        VAR_23 = VAR_23.view(1, VAR_23.size(0), (- 1))
        VAR_22 = VAR_22.view(1, VAR_22.size(0), (- 1))
        return (VAR_23, (VAR_23, VAR_22))
```

Code Deobfuscated using DOBF

```
class LSTM(nn.Module):

    def __init__(self, input_size, hidden_size, bias):
        super(LSTM, self).__init__()
        self.input_size = input_size
        self.hidden_size = hidden_size
        self.h1 = nn.Linear(input_size, (4 * hidden_size), bias=bias)
        self.h2 = nn.Linear(hidden_size, (4 * hidden_size), bias=bias)
        self.init_weights()

    def init_weights(self):
        stdv = (1.0 / math.sqrt(self.hidden_size))
        for m in self.modules():
            m.data.uniform_((- stdv), stdv)

    def forward(self, x, prev_state):
        (prev_h, prev_c) = prev_state
        prev_h = prev_h.view(prev_h.size(1), (- 1))
        prev_c = prev_c.view(prev_c.size(1), (- 1))
        x = x.view(x.size(1), (- 1))
        h = (self.h1(x) + self.h2(prev_h))
        s = h[:, :(3 * self.hidden_size)].sigmoid()
        c = h[:, (3 * self.hidden_size):].tanh()
        r = s[:, :self.hidden_size]
        g = s[:, self.hidden_size:(2 * self.hidden_size)]
        o = s[:, (- self.hidden_size):]
        c = (th.mul(prev_c, g) + th.mul(r, c))
        h = th.mul(o, c.tanh())
        h = h.view(1, h.size(0), (- 1))
        c = c.view(1, c.size(0), (- 1))
        return (h, (h, c))
```

ID	Ground Truth	DOBF
CLASS_0	LSTM	LSTM
FUNC_0	reset_parameters	init_weights
FUNC_1	forward	forward
VAR_0	self	self
VAR_1	input_size	input_size
VAR_2	hidden_size	hidden_size
VAR_3	bias	bias
VAR_4	i2h	h1
VAR_5	h2h	h2
VAR_6	self	self
VAR_7	std	stdv
VAR_8	hidden_size	hidden_size
VAR_9	w	m
VAR_10	parameters	modules
VAR_11	self	self
VAR_12	x	x
VAR_13	hidden	prev_state
VAR_14	h	prev_h
VAR_15	c	prev_c
VAR_16	preact	h
VAR_17	gates	s
VAR_18	g_t	c
VAR_19	i_t	r
VAR_20	f_t	g
VAR_21	o_t	o
VAR_22	c_t	c
VAR_23	h_t	h

Figure 8: Deobfuscation of an LSTM cell. DOBF is able to recover several of the original tokens, including the class name (`LSTM`) and the full signature of the `__init__` method. Even though DOBF does not always recover the original token, it generally proposes very relevant tokens which improves code readability. In particular, for some tokens the accuracy and subtoken scores would be zero but the recovered tokens are still very relevant. For instance, `reset_parameters` (`FUNC_0`) was renamed to `init_weights`, `std` (`VAR_7`) was renamed to `stdv`, and `hidden` (`VAR_13`) was renamed to `prev_state`. In those instances, the original and recovered tokens share no subtoken despite having very similar semantics.

Input Code	Deobfuscated Identifiers	
`def FUNC_0(VAR_0, VAR_1):` `    return sum(map(operator.mul, VAR_0, VAR_1))`	FUNC_0 VAR_0 VAR_1	dotProduct list1 list2
`def FUNC_0(VAR_0):` `     VAR_1 = urllib2.urlopen(VAR_0)` `     VAR_2 = VAR_1.read()` `     return VAR_2`	FUNC_0 VAR_0 VAR_1 VAR_2	get_html url response html
`def FUNC_0(VAR_0):` `    VAR_1 = set(VAR_0)` `    return (len(VAR_1) == len(VAR_0))`	FUNC_0 VAR_0 VAR_1	all_unique iterable s
`def FUNC_0(VAR_0, VAR_1):` `    return list(collections.deque(VAR_0, maxlen=VAR_1))`	FUNC_0 VAR_0 VAR_1	tail s n
`def FUNC_0(VAR_0):` `    return sum((VAR_1 for VAR_1 in VAR_0 if ((VAR_1 % 2) == 0)))`	FUNC_0 VAR_0 VAR_1	even_sum nums n

Figure 9: Examples of full deobfuscations of Python functions. Even when every identifier is obfuscated, DOBF is able to propose relevant names. The proposed function name is informative and relevant in all examples since the first function computes a dot product, the second downloads a HTML page and returns its content, the third evaluates whether the input contains only unique elements, the fourth computes the tail of an iterable, and the fifth computes the sum of the even elements of an iterable.

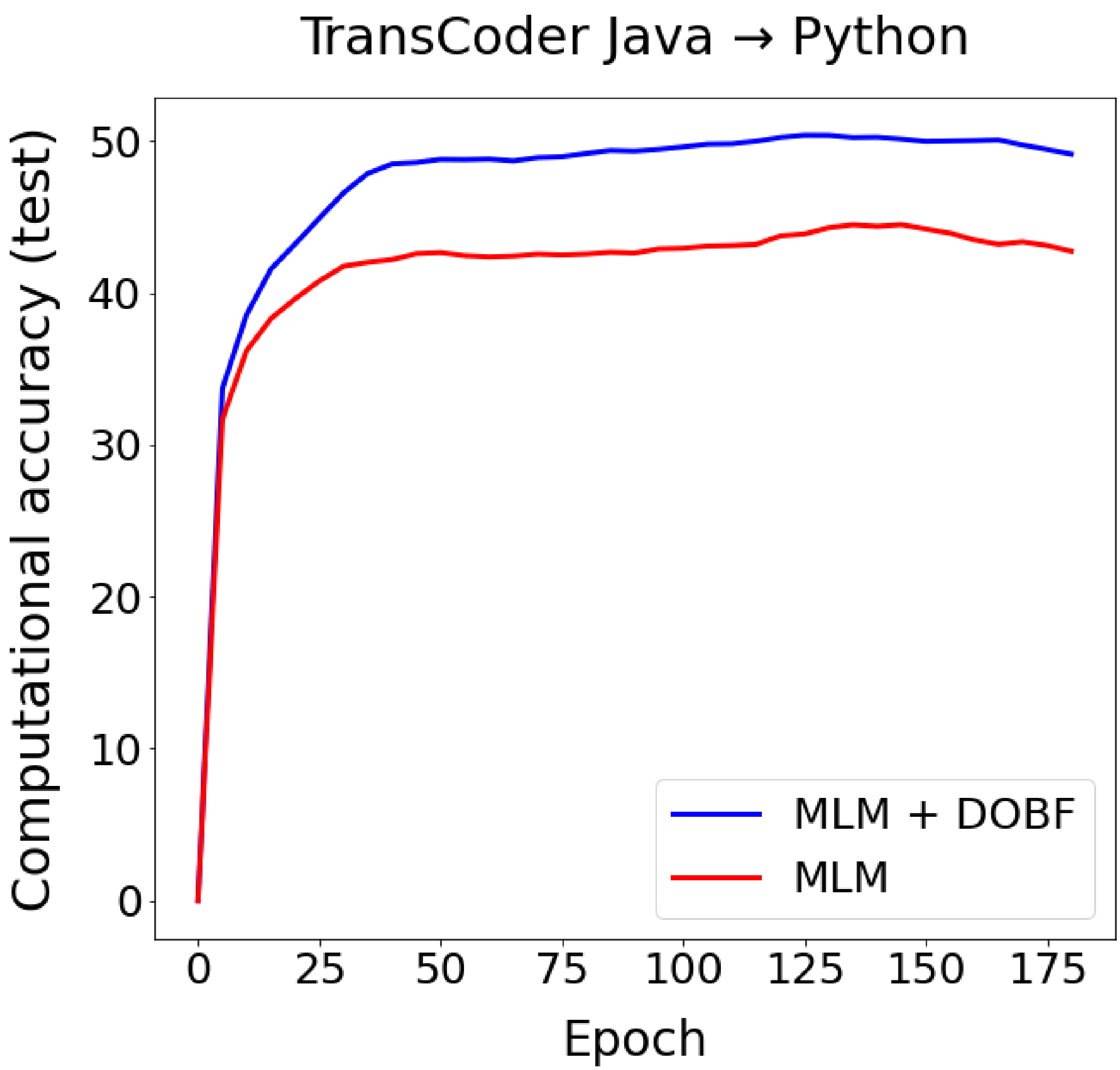

Figure 10: TransCoder results for different pre-training schemes. Pre-training our model with MLM+DOBF instead of MLM only, allows to quickly reach higher levels of computational accuracy when fine-tuning for Java → Python translation. The gap between MLM+DOBF and MLM persists until convergence.