DOBF: A Deobfuscation Pre-Training Objective for Programming Languages

Marie-Anne Lachaux*  
Facebook AI Research

Marc Szafraniec  
Facebook AI Research

Baptiste Roziere*  
Facebook AI Research and Paris-Dauphine University

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:

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 syntactic 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],
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 model for 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 bf's 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(8), 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.

\section*{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.

\subsection*{4.1 Deobfuscation}

We evaluate our model on two applications of the deobfuscation task: when \(p_{\text{obf}} = 0\) (the model has to retrieve a single identifier name), and \(p_{\text{obf}} = 1\) (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 \(p_{\text{obf}} = 1\) 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.

\subsection*{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.

\textbf{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.

\textbf{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.

\textbf{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.

\textbf{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.

\subsection*{4.3 Experimental details}

\textbf{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.

\textbf{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.

\textbf{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: \(p_{\text{obf}} \in \{0, 0.5, 1\}\). For each \(p_{\text{obf}}\) value, we train models with multiple initial learning rates ranging from \(10^{-4}\) to \(3 \times 10^{-4}\) and select the best one using the average subtoken F1 score computed on the validation dataset.

\textbf{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.

\subsection*{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 (\(p_{\text{obf}} = 0\)) or when all identifiers are obfuscated (\(p_{\text{obf}} = 1\)), for models trained with \(p_{\text{obf}} \in \{0, 0.5, 1\}\). Even when evaluating with \(p_{\text{obf}} = 1\), training with \(p_{\text{obf}} = 0\) is less efficient than \(p_{\text{obf}} = 0.5\) since the model is only trained to generate a single variable for each input sequence. Training with \(p_{\text{obf}} = 0.5\) is a more difficult task that requires the model to learn and understand more about code semantics. Forcing

\footnotesize
\begin{itemize}
  \item \(p_{\text{obf}} = 0\)
  \item \(p_{\text{obf}} = 1\)
\end{itemize}

\end{document}
We observe that DOBF manages to identify the key tokens and to recover a fully obfuscated function, it obtains the best accuracy and subtoken F1 score of DOBF evaluated with \( p_{obf} = 0 \) (i.e. full deobfuscation, where all tokens are obfuscated). We consider models trained with different obfuscation probabilities \( p_{obf} \). DOBF performs well for both tasks, and it even performs better than DOBF for Identifier Name Proposal. DOBF and DOBF perform poorly when evaluated on other \( p_{obf} \) parameters. Pre-training DOBF with MLM further improves the performance.

<table>
<thead>
<tr>
<th>Eval ( p_{obf} = 0 )</th>
<th>Eval ( p_{obf} = 1 )</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DOBF(_0)</strong></td>
<td>56.3 68.0 0.4 0.9</td>
</tr>
<tr>
<td><strong>DOBF(_{0.5})</strong></td>
<td>61.1 71.2 41.8 54.8</td>
</tr>
<tr>
<td><strong>DOBF(_1)</strong></td>
<td>18.1 27.0 45.6 58.1</td>
</tr>
<tr>
<td><strong>MLM+DOBF(_{0.5})</strong></td>
<td>67.6 76.3 45.7 58.0</td>
</tr>
<tr>
<td><strong>MLM+DOBF(_1)</strong></td>
<td>20.0 28.3 49.7 61.1</td>
</tr>
</tbody>
</table>

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 \( p_{obf} \). DOBF performs well for both tasks, and it even performs better than DOBF for Identifier Name Proposal. DOBF and DOBF perform poorly when evaluated on other \( p_{obf} \) parameters. Pre-training DOBF with MLM further improves the performance.

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

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.

<table>
<thead>
<tr>
<th></th>
<th>Java</th>
<th>Python</th>
</tr>
</thead>
<tbody>
<tr>
<td>All - Size</td>
<td>26 GB</td>
<td>19 GB</td>
</tr>
<tr>
<td>All - Nb files</td>
<td>7.9M</td>
<td>3.6M</td>
</tr>
<tr>
<td>Av. nb of tokens / file</td>
<td>718</td>
<td>1245</td>
</tr>
<tr>
<td>Av. nb of identifiers / file</td>
<td>25.9</td>
<td>41.8</td>
</tr>
<tr>
<td>Input Code</td>
<td>Proposed Function Name</td>
<td></td>
</tr>
<tr>
<td>------------</td>
<td>------------------------</td>
<td></td>
</tr>
<tr>
<td><code>public static void FUNC_0 (String path){</code></td>
<td>deleteFile 48.3%</td>
<td></td>
</tr>
<tr>
<td>try {`</td>
<td>remove 16.9%</td>
<td></td>
</tr>
<tr>
<td>Files.delete(path);`</td>
<td>DeleteFile 13.2%</td>
<td></td>
</tr>
<tr>
<td>}`</td>
<td>removeFile 13.1%</td>
<td></td>
</tr>
<tr>
<td>catch (Exception e) {`</td>
<td>deleteFileQuietly 8.4%</td>
<td></td>
</tr>
<tr>
<td>System.err.println(&quot;Error deleting file &quot; + path);`</td>
<td></td>
<td></td>
</tr>
<tr>
<td>}`</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>public static void FUNC_0 (String path){</code></td>
<td>createDir 23.5%</td>
<td></td>
</tr>
<tr>
<td>if (!Files.exists(path)) {`</td>
<td>createDirectory 20.9%</td>
<td></td>
</tr>
<tr>
<td>Files.createDirectories(path);`</td>
<td>createDirectoryNameExists 20.8%</td>
<td></td>
</tr>
<tr>
<td>}`</td>
<td>ensureDirectoryExists 18.5%</td>
<td></td>
</tr>
<tr>
<td>}`</td>
<td>createDirectoryIfNotExists 16.3%</td>
<td></td>
</tr>
<tr>
<td><code>public static void FUNC_0 (String path){</code></td>
<td>zip 28.6%</td>
<td></td>
</tr>
<tr>
<td>if (!Files.exists(path)) {`</td>
<td>intersect 20.0%</td>
<td></td>
</tr>
<tr>
<td>Files.createDirectories(path);`</td>
<td>combine 17.9%</td>
<td></td>
</tr>
<tr>
<td>}`</td>
<td>merge 17.5%</td>
<td></td>
</tr>
<tr>
<td>}`</td>
<td>intersection 16.0%</td>
<td></td>
</tr>
<tr>
<td><code>public static int FUNC_0 (int n){</code></td>
<td>fib 41.5%</td>
<td></td>
</tr>
<tr>
<td>int a = 0, b = 1;`</td>
<td>fibonacci 36.6%</td>
<td></td>
</tr>
<tr>
<td>int tmp;`</td>
<td>fibon 9.1%</td>
<td></td>
</tr>
<tr>
<td>for (int i = 0; i &lt; n; i ++){`</td>
<td>fibo 8.8%</td>
<td></td>
</tr>
<tr>
<td>tmp = a + b;`</td>
<td>fibonacci_series 4.0%</td>
<td></td>
</tr>
<tr>
<td>a = b;`</td>
<td></td>
<td></td>
</tr>
<tr>
<td>b = tmp;`</td>
<td></td>
<td></td>
</tr>
<tr>
<td>}`</td>
<td></td>
<td></td>
</tr>
<tr>
<td>return a;`</td>
<td></td>
<td></td>
</tr>
<tr>
<td>}`</td>
<td></td>
<td></td>
</tr>
<tr>
<td><code>public static float FUNC_0 (List&lt;Float&gt; vec1,</code></td>
<td>dotProduct 40.9%</td>
<td></td>
</tr>
<tr>
<td>List&lt;Float&gt; vec2) {`</td>
<td>dot 23.9%</td>
<td></td>
</tr>
<tr>
<td>float size = vec1.size();`</td>
<td>dot_product 16.5%</td>
<td></td>
</tr>
<tr>
<td>assert size == vec2.size();`</td>
<td>dotproduct 10.5%</td>
<td></td>
</tr>
<tr>
<td>float result = 0.0f;`</td>
<td>inner 8.3%</td>
<td></td>
</tr>
<tr>
<td>for (int i = 0; i &lt; size; i ++) {`</td>
<td></td>
<td></td>
</tr>
<tr>
<td>result += vec1.get(i) * vec2.get(i);`</td>
<td></td>
<td></td>
</tr>
<tr>
<td>}`</td>
<td></td>
<td></td>
</tr>
<tr>
<td>return result;`</td>
<td></td>
<td></td>
</tr>
<tr>
<td>}`</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

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.
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[1] 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.
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.

```
Input Code

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

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

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

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

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

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

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.

```
BFS Implementation

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
```

```
DFS Implementation

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
```

```
DFS with Erroneous Variable Name

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
```
### Obfuscated Code

```python
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[::,(VAR_11.VAR_8):].sigmoid()
    VAR_18 = VAR_16[::,(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 = (VAR_15 * VAR_21) + (VAR_19 * VAR_18)
    VAR_23 = VAR_22.view(1, VAR_22.size(0), (- 1))
    return (VAR_23, (VAR_23, VAR_22))
```

### Code Deobfuscated using DOBF

```python
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))
        h = (self.h1(x) + self.h2(prev_h))
        s = h[::,(VAR_11.VAR_8):].sigmoid()
        c = h[::,(VAR_11.VAR_8):].tanh()
        r = s[::,(VAR_11.VAR_8)]
        g = r[::,VAR_11.VAR_8].sigmoid()
        c_t = s[::,(VAR_11.VAR_8)]
        c = c_t[::,VAR_11.VAR_8]
        h = (r[::,VAR_11.VAR_8] * c_t) + (s[::,VAR_11.VAR_8] * prev_c)
        h = h.view(h.size(0), (- 1))
        return (h, (h, c))
```

### 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.
def FUNC_0(VAR_0, VAR_1):
    return sum(map(operator.mul, VAR_0, VAR_1))

def FUNC_0(VAR_0):
    VAR_1 = urllib2.urlopen(VAR_0)
    VAR_2 = VAR_1.read()
    return VAR_2

def FUNC_0(VAR_0):
    VAR_1 = set(VAR_0)
    return (len(VAR_1) == len(VAR_0))

def FUNC_0(VAR_0, VAR_1):
    return list(collections.deque(VAR_0, maxlen=VAR_1))

def FUNC_0(VAR_0):
    return sum((VAR_1
    for VAR_1
    in VAR_0
    if ((VAR_1 % 2) == 0)))

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.