Exploring the Potential of LLMs for Code Deobfuscation

David Beste[1][0009−0002−0597−7788]✉, Grégoire Menguy[2][0000−0002−8776−8770], Hossein Hajipour[1][0000−0001−6911−9390], Mario Fritz[1][0000−0001−8949−9896], Antonio Emanuele Cinà[3][0000−0003−3807−6417], Sébastien Bardin[2][0000−0002−6509−3506], Thorsten Holz[1][0000−0002−2783−1264], Thorsten Eisenhofer[4][0000−0002−4706−260X], and Lea Schönherr[1][0000−0003−3779−7781]

[1] CISPA Helmholtz Center for Information Security
{first.last}@cispa.de
[2] Université Paris-Saclay, CEA, List
{first.last}@cea.fr
[3] University of Genoa
{first.last}@unige.it
[4] BIFOLD & TU Berlin
{first.last}@tu-berlin.de

Abstract. Code obfuscation alters software code to conceal its logic while retaining functionality, aiding intellectual property protection but hindering security audits and malware analysis. To address this, automated deobfuscation techniques have been developed, though existing approaches remain constrained by limited scope and specificity. Motivated by these challenges, this paper explores a novel approach for code deobfuscation based on Large Language Models (LLMs). First, we investigate the general capabilities of LLMs in reducing code complexity by choosing five different source-to-source obfuscation methods. Despite challenges regarding semantical correctness, our findings indicate that LLMs can be very effective in this task. Building on this, we fine-tune two versatile models capable of simplifying code obfuscated through up to seven different chained obfuscation transformations while consistently outperforming deobfuscation based on compiler optimizations and general-purpose LLMs. Our best model demonstrates an average Halstead metric program length reduction of 89.21% for our most challenging scenario. Finally, we conduct a memorization test to assess if performance stems from memorized code rather than true deobfuscation capabilities, which our models pass.

1 Introduction

Code obfuscation [8, 9] refers to various methods to disguise a program's functionality, making its source or machine code harder for a human analyst to comprehend. Malicious actors often use obfuscation techniques to complicate malware analysis, thus impeding the development of effective detection methods

and countermeasures [21, 35]. For these reasons, deobfuscation methods have been developed to recover the original structure of the code. These methods use sophisticated approaches such as advanced static analysis [31, 35], dynamic analysis [37, 41], and, more recently, program synthesis [4, 18, 29]. Despite their potential, the effectiveness of these techniques often encounters significant limitations. Many focus on specific obfuscation strategies, such as mixed Boolean arithmetic (MBA) [31, 32] or opaque predicates [3, 30], which limits their applicability to broader obfuscation scenarios. Furthermore, although program synthesis offers a promising approach, it is so far only suitable for deobfuscating small and simple pieces of code without complex structures [4, 29]. This limitation significantly diminishes its practical utility for broad-scale deobfuscation tasks and emphasizes the need for further progress in this area.

In this paper, we explore the capabilities of LLMs in code deobfuscation tasks. LLMs have demonstrated remarkable abilities in generating code, as evidenced by applications such as writing code from natural language descriptions [33], code summarization [39], and automatic code repair [14, 22]. Based on these advances, we investigate whether LLMs provide enough inherent code comprehension to integrate specialized knowledge with broad applicability, which is essential for deobfuscating complex code. This focus allows us to address two main challenges:

First, the LLM needs to identify the transformation(s) applied to the code to remove the obfuscation structures entirely. Second, the LLM needs to develop a nuanced understanding of the context in which the obfuscated code operates so that it can reconstruct the code without breaking functionality.

To better understand the potential of LLMs, our first step involves evaluating the foundational capabilities of recent code models, including DeepSeek Coder [16], Code Llama [33], and GPT-4 [1], for this task. To this end, we consider a variety of source-to-source obfuscation techniques such as control-flow flattening [38], opaque predicates [8], and MBA encoding [4] as representative examples of different code obfuscation strategies. In total, we select five different transformation methods to construct a dataset that contains pairs of original and obfuscated codes. The data set contains 30,000 training samples and 2,400 test samples for the single transformation and multi-chain deobfuscation scenarios.

By controlling the construction of the dataset, we can generate code pairs that undergo one or more transformations. Using this constructed dataset, we conduct a series of systematic experiments to evaluate the capabilities of LLMs in deobfuscating codes obfuscated by different sets of transformations. In our study, we examine how base models and instruction-tuned code models perform in this task, both in zero-shot scenarios and through fine-tuning. Our key metrics are the complexity reduction of the code and the preservation of its functionality.

Main findings. Our experiments result in the following observations:

- While LLMs can effectively reduce the complexity of obfuscated codes, they sometimes break the functionality of the code.
- Fine-tuned models show significant improvements over general-purpose LLMs such as GPT-4 and existing compiler optimizations, which we used as a baseline to compare against.

- The models exhibited very good syntactical correctness even in our most complex scenarios.
- In fact, in the most demanding scenarios, our best model achieved an average program length reduction of 89.21% according to the Halstead metric [20].
- With increasing complexity of the obfuscations, the semantical correctness declines.
- After performing a memorization test, we find that the LLMs do not make use of memorized samples but rather truly deobfuscate the code.

Our research highlights the potential of LLMs to complement existing deobfuscation methods.

Contribution. In summary, we systematically analyze the potential of LLMs for code deobfuscation and compare general-purpose models, specialized code models, and instruction-tuned models. This analysis not only highlights the strengths of LLMs in tackling code obfuscation, but it also shows their current limitations, such as generating semantically incorrect code. We build the first scalable dataset for training and evaluating the performance of LLMs for the deobfuscation task that can be used with arbitrary C programs. Our code is available at `https://github.com/DavidBeste/llm-code-deobfuscation`.

2 Obfuscation of Code

Obfuscation refers to the process of making software code difficult to understand. To protect a program P from reverse engineering, obfuscation translates it into a program P', which is harder to analyze but semantically equal. Figure 1 shows an example where three transformations (control-flow flattening, arithmetic encoding, and argument randomization) have been applied to obfuscate the code.

Formally, we define this *obfuscation transformation* T as a function

$$T\colon P \mapsto P'\,, \tag{1}$$

which maps a program P into an obfuscated version P' subjected to semantic constraints. The *obfuscator* is assumed to be equipped with a set of such transformations $\mathcal{T}$, and obfuscation is done for a *chain* of transformations $[T_1, \ldots, T_k] \in \mathcal{T}^k$ by iteratively applying the transformations to the program, i.e.,

$$P' = (T_1 \circ ... \circ T_k)(P)\,. \tag{2}$$

The result P' then represents the obfuscated program. In this work, we focus on the *Tigress* obfuscation toolkit [7], which represents a state-of-the-art C code obfuscator and includes a wide range of configurable obfuscation schemes [9, 25, 26, 35], making it well suited for scientific investigations.

Code Deobfuscation. Similarly, efforts have been devoted to deobfuscate programs in an automated way. Approaches that rely on static analysis [31,32,35],

```
__inline static void strtoupper(char *s) {
  char *c;
  c = s;
  while (*c) {
    if ((int )*c >= 97) {
      if ((int )*c <= 122) {
        *c = (char )(((int )*c - 97) + 65);
      }
    }
    c ++;
  }
  return;
}
```

(a) Original Code

```
void _xa(char *_k0, long _k1) {
  char *_k2 ;
  unsigned long _k3 ;
  int _k4 ;
  _k3 = 1UL;
  while (1) {
    switch (_k3) {
    case 4UL: ;
    if (97 <= (int )*_k2) {
      _k3 = 0UL;
    } else {
      _k3 = 3UL;
    }
    break;
    [...]
```

(b) Obfuscated Code

Fig. 1: **Example Code.** Figure 1b presents an obfuscated version of the program shown in Figure 1a. This example is truncated for brevity; the full code consists of 55 lines.

```
void _xa(char *_k0) {
  char *_k2;
  _k2 = _k0;
  while (*_k2) {
    if ((int )*_k2 >= 97) {
      if ((int )*_k2 <= 122) {
        *_k2 = (char )(((int )*_k2 - 97) + 65);
      }
    }
    _k2 ++;
  }
  return;
}
```

Fig. 2: **Deobfuscated code.** Code recovered from the obfuscated code shown in Figure 1b, extracted with our approach

dynamic analysis [11, 37, 41] or program synthesis [4, 29] have been shown to be very efficient. These approaches aim to be *obfuscator-independent* and see each obfuscation as a general problem to solve. In exchange, we face two main challenges: (1) We must know which family of obfuscation has been used to leverage the corresponding deobfuscation method; (2) We must know on which scope the deobfuscation should be applied to get the best results.

Transformations. We consider five different transformation techniques. This selection aims to include transformations altering different aspects of the program code, such as complicating the control flow or increasing the number of operations in the program. Furthermore, the Tigress toolkit makes several recommendations for obfuscation chains [6]. From these, we additionally include all transformations from the first "recipe". To increase diversity, we vary the parameters for the chosen transformations using the recommendations from the Tigress documentation. In the following, we explain the five chosen transformations in more detail.

Encode arithmetic. Mixed-Boolean-Arithmetic (MBA) translates an easy-to-understand arithmetic expression into a more obscure equivalent expression, by manipulating both arithmetic and boolean operators [13]. The following example shows a basic MBA encoding from the Tigress [7] documentation, replacing the $+$ operator with a more complex structure:

$$x + y \longrightarrow (x \oplus y) + 2 \times (x \wedge y). \tag{3}$$

Encode branches. This transformation disguises static jumps as return instructions to fool disassembly tools into going to the next return address instead of following the jump target [25]. Again, a manual analysis of the control flow is cumbersome and requires a great deal of effort for an analyst.

Control-flow flattening. The flattening protection [9, Chap 4.3.2] breaks the control-flow graph (CFG) of the code to create a loop to execute that will be dispatched over different blocks. As a result, instead of seeing an informative CFG (with branches and loops), a reverse engineer will only see a code structure that must be simplified to understand the real behavior of the code.

Opaque predicates. The opaque predicate transformation [10] aims to break the CFG of the obfuscated code. To do so, it adds conditionals, always evaluating to true or false, to artificially increase the size of the CFG, thus obscuring which parts of the code are reachable.

Randomize arguments. This technique randomizes the order of function arguments and adds bogus (i.e., new and semantically useless) arguments [5] requiring a reverse engineer to track them and analyze their purpose.

3 LLM-supported Deobfuscation

We are now set to examine how LLMs can enhance the understanding and simplification of complex patterns in obfuscated code, potentially enhancing traditional deobfuscation methods. We focus on a scenario where, given an obfuscated program, we aim to retrieve a simpler and semantically equivalent version for further analysis.

Challenges. One of the main obstacles in using current deobfuscation methods is that many state-of-the-art techniques are tailored to specific transformations. To properly deobfuscate a program, it is necessary to first identify the specific obfuscation methods used in order to choose the right deobfuscation tool. Take, for instance, the obfuscated code shown in Figure 1b: The original 13-line code from Figure 1a has been transformed to 55 lines with a complex control flow. To deobfuscate this, we must first identify the obfuscations applied—in this case, control-flow flattening, arithmetic encoding, and argument randomization—and then apply the right tools to reverse these changes. This analysis requires expertise and can be error-prone, particularly when the code undergoes multiple chained transformations. Furthermore, deobfuscation tools often require to specify which

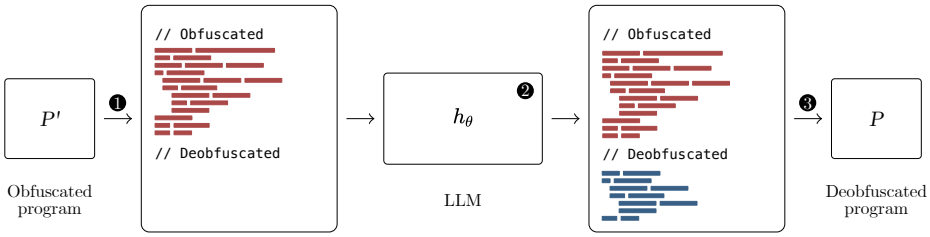

Fig. 3: **LLM-based code deobfuscation.** We consider the LLM (Step ❷) as a generic deobfuscator, receiving the obfuscated code embedding in its input (Step ❶). The model is trained to extend this input sequence with the deobfuscated code in its output. From the response, we extract the deobfuscated program (Step ❸). The actual format depends on the type of model. Shown here is a C comment style delimiter we use for pure code models.

part of the code to deobfuscate. For example, tools designed to simplify mixed boolean arithmetic expressions [26] demand the specific obfuscated expression as input. While some methods can process the entire code [34, 41], they do not scale well with larger code bases. This limitation highlights the need for more effective deobfuscation tools that can handle code efficiently.

LLMs for Deobfuscation. We hypothesize that LLMs can help address these challenges, as they have demonstrated remarkable performance in various code tasks [14, 17, 19, 22]. Rather than developing specialized tools for each transformation, we explore the use of LLMs that process the entire obfuscated code in their input and are trained to output the deobfuscated code in their response. The high-level idea is illustrated in Figure 3. When, for example, the obfuscated function from Figure 1b is fed to the LLM, the model returns a simplified version (see Figure 2) that closely resembles the original code.

Models. The effectiveness of this approach naturally relies on the selected model. On the one hand, we consider a large instruction-tuned model like GPT-4 and design a prompt to instruct the model for deobfuscation. Although the model is not a dedicated code model but rather general-purpose, it performs surprisingly well in the code domain [1]. On the other hand, previous research indicates that models fine-tuned for code-related tasks can substantially outperform generalist models while being significantly smaller [16, 33]. In light of this, we additionally explore the use of specialized coding models, namely Code Llama [33] and DeepSeek Coder [16]. We consider both the direct use of the pre-trained models and instruction-tuned versions. Instruction-tuning a code model has been shown to further enhance its coding performance across various benchmarks [16, 33]. We fine-tune the code models using the dataset depicted in Figure 4 introduced next in Section 3.1. Table 1 summarizes the selected models.

Prompt format. We format samples based on the LLMs' training: We use C-style comments for pure code models as shown in Figure 3 and conversational style for instruction-tuned models.

Table 1: LLMs considered in the study

Name	Size	Open Access	Instruction Tuned	Coding Specialist
DeepSeek Coder [16]	6.7B	✓	✓	✓
Code Llama [33]	7B	✓	✗	✓
GPT-4 [1]	n/a	✗	✓	✗

3.1 Dataset and Training

For fine-tuning and evaluation, we require a dataset that meets several criteria. First, we need a diverse and comprehensive collection of code, preferably from real-world sources. Second, we need a way to verify the semantics of the deobfuscated code to compare the functionality between the original and the deobfuscated code and to measure the models' understanding. Furthermore, to practically instantiate the dataset, we focus on a function granularity, i.e., the program P is a single source code function.

Dataset selection. Based on the aforementioned requirements, we use the ExeBench dataset [2], a comprehensive collection of real-world C code crawled from GitHub specifically designed for machine learning purposes. The dataset is representative of real-world code based on different software metrics and provides input/output (I/O) samples for each C code, facilitating the evaluation of the semantical correctness. We use the `train-real-compilable` subset for training and the `test-real` subset for evaluation since these both are the closest to real-world code and allow for compilation, which is necessary for our checks later. These subsets comprise 885,074 and 2,134 individual functions.

Pre-processing. We pre-process the dataset and exclude `main` functions and functions that contain no arithmetic operations or branches, since such samples might lead to trivial obfuscations when applying certain transformations (e.g, encoding of arithmetic and flattening of code). In addition, we filter out functions with duplicate names to minimize the inclusion of semantically similar functions, which results in a more diverse training set. We then canonicalize the original code to reduce the variance in coding style between the original and obfuscated samples, e.g., we replace ternary operators with if-else statements. Finally, we randomize all identifier names to prevent the LLM from inferring code structures from descriptive identifier names.

Dataset Generation. The generation process is illustrated in Figure 4. In the first step ❶, we create differently obfuscated versions of programs P_i with $i = 1, \ldots, N$ where N is defined by the number of samples to incorporate. Therefore, we construct M chains $\mathbf{T}_j = [T_1, \ldots, T_L]$ with length $L \leq L_{max}$ where L_{max} defines the maximum chain length. Transformations T_l' with $l = 1, \ldots, L$ are sampled uniformly from the set of transformations $\mathcal{T}$. For each program P_i and chain $\mathbf{T}_j$, we create an obfuscated program $P'_{i,j}$. To ensure a diverse set, we randomize

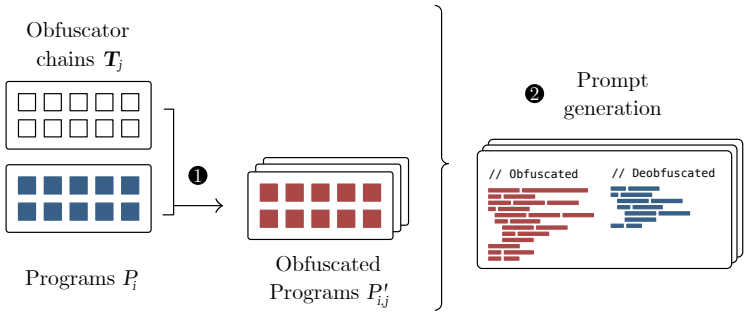

Fig. 4: **Dataset generation.** The dataset is created in two steps. The input for the first step ❶ is a set of programs, which are transformed into obfuscated programs using several obfuscator chains. In the second step ❷, we subsequently create data samples for the fine-tuning, consisting of pairs of the obfuscated programs and their unmodified counterparts.

the initialization seed for each chain and the parameters for the transformations which leads to $N \cdot M$ obfuscated programs $(P'_{1,1}, \ldots, P'_{1,M}, \ldots, P'_{N,1}, \ldots, P'_{N,M})$. Following this, in step ❷, we pair each obfuscated program with its corresponding original version, i.e., $(P_n, P'_{n,m})$. The resulting pairs serve as the unique samples.

3.2 Metrics

To assess the models' performance, we consider two aspects. We evaluate correctness and measure the models' effectiveness in recovering a code close to the original one from the obfuscated version.

Correctness. We distinguish between semantical and syntactical correctness.

Syntactical Correctness. We assess *syntactical correctness* using the executable wrappers from ExeBench and check for any errors during compilation. Since the wrappers are written in C++, we use the `g++` compiler for this.

Semantical Correctness. While checking for syntactical correctness is trivial using a compiler, semantical correctness is more challenging. We use the I/O samples from the Exebench dataset to approximate the program's functional correctness. We compare the behavior of the deobfuscated program to its obfuscated version. If the output is the same, we conclude that the program is semantically correct; otherwise, it is considered incorrect. We do not expect significant gains from approaches like differential testing and symbolic execution since the I/O samples were crafted specifically for correctness testing.

Deobfuscation Performance. To evaluate deobfuscation performance, we need a metric that effectively reflects the model's ability to reduce code complexity.

This metric should consider the complexity of the original, obfuscated, and deobfuscated code in a single value. Therefore, we propose the following metric:

$$P_{Deobf} = 1 - \frac{C_{Deobf} - C_{Orig}}{C_{Obf} - C_{Orig}} \tag{4}$$

Intuitively, the closer the deobfuscated code is to the original, the closer the score approaches 1. Conversely, the closer the deobfuscated code is to the obfuscated sample, the closer the score approaches 0. Scores larger than 1 imply that the model made the code less complex than the original version, indicating it found a more compact representation. Scores less than 0 imply that the code returned by the model is more complex than the obfuscated version, indicating a failure in deobfuscation.

To instantiate the complexity function C, we could use common code metrics to assess the complexity of program code, such as *cyclomatic complexity* [28] as well as the Halstead metrics [20]. Katzmarski and Koschke empirically evaluated that the Halstead metrics correlate with the programmer's perception of code complexity [23] and, therefore, are suitable for measuring performance in the deobfuscation task. For our evaluation, we focus on the Halstead program length since it can capture changes from all five transformations we chose.

4 Experimental Evaluation

We now present our empirical evaluation of the deobfuscation capabilities of large language models. All used LLMs models have been trained to manipulate C code efficiently. Hence, considering the C-level deobfuscation tasks enables focusing on the deobfuscation capabilities of LLMs *per se*. Moreover, in practice, an approximation of the C code can be retrieved from the binary through decompilation [27]. We split this investigation into three main parts.

- First, we analyze the code that has been obfuscated with a *single* transformation. This will help us infer the general capabilities of the considered LLMs in a very controlled scenario.
- Second, we move to a more complex setting and consider *chains* of transformations. We start with an experiment where the LLMs are trained on multiple obfuscation techniques simultaneously. Building on that, we also train and evaluate models on data where multiple transformations are applied on a single sample. Here, we target obfuscation chains of up to five transformations for training and up to seven for evaluation, which will allow us to learn about potential limits of deobfuscation with LLMs.
- Finally, we seek to determine if the observed performance of the language models might be due to the models detecting and recalling code memorized during training.

To summarize, our investigations revolve around three main research questions:

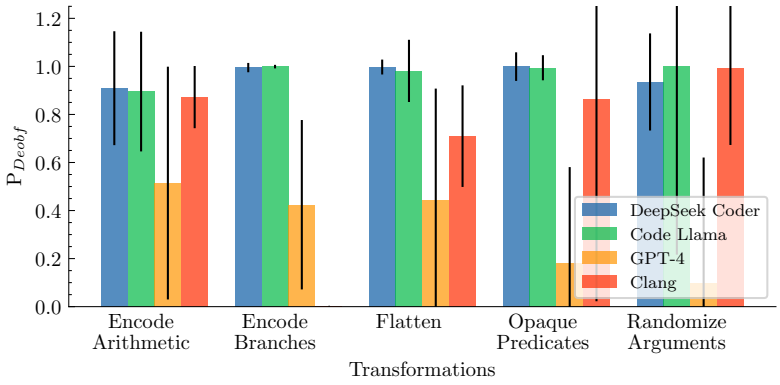

Fig. 5: **Complexity reduction.** We measure the average deobfuscation scores P_{Deobf} for the five transformations according to the formula introduced earlier for our fine-tuned models and GPT-4. For *encode branches*, Clang fails to produce semantically correct samples, which is why we only compare the three other models in this case.

RQ1 What are the general capabilities of LLMs in deobfuscating state-of-the-art code obfuscation transformations?

RQ2 How does the model performance evolve when chaining multiple transformations together?

RQ3 Do models use memorized code during obfuscation?

All experiments are performed on a server equipped with an A100 GPU with 40 GB of VRAM.

4.1 Single Transformations

For our first experiment, we examine how the language models can handle code obfuscated by each of the five transformation techniques individually. In this controlled scenario we can compare the LLMs' code deobfuscation capabilities grouped by different transformation types.

Setup. For the fine-tuning set, we sample 3,000 functions from the `train-real-compilable` split of the ExeBench dataset [2]. For each function, we create one obfuscated version for each transformation, resulting in a dataset of $5 \times 3,000 = 15,000$ samples in total. From these samples, we create pairs of obfuscated and original code as explained in Section 3 and randomize all identifier names. For the evaluation dataset, we sample 200 functions from the `test-real` split and apply each of our five transformations to each sample, resulting in a total evaluation dataset of $1,000$ samples. To evaluate the correctness of the LLMs' outputs, we use the executable wrappers from ExeBench as our test harness.

For each transformation, we fine-tune a model for both DeepSeek Coder and Code Llama. Additionally, we evaluate the performance of each transformation

using GPT-4. To provide a broader context for the performance of these models, we compare their results against compiler optimization techniques. Specifically, we consider Clang [24], which was shown to be surprisingly effective for deobfuscation [34]. We therefore convert the obfuscated code into Clang's intermediate representation. We then evaluate it across all optimization levels (`-O0`, `-O1`, `-O2`, and `-O3`), selecting the best version based on the complexity metric. This step is necessary because different optimization levels balance time and memory differently, meaning the most optimized code is not always the simplest. The complexity metric is computed directly on the intermediate representation.

Results. Figure 5 presents the average deobfuscation scores P_{Deobf} together with the standard deviation, considering only samples that were successfully deobfuscated both syntactically and semantically. We observe that both code models show strong performance, with neither consistently outperforming the other across all transformations. GPT-4, on the other hand, is significantly outperformed by the code models across all transformations, with *opaque predicates* and *randomize arguments* exhibiting the highest difference and *encode arithmetic* the lowest. Clang is consistently outperformed by the fine-tuned models except for *encode arithmetic* and *randomize arguments*, where Clang is on par with the fine-tuned LLMs. This indicates that Clang has strengths at reducing complex arithmetic expressions and removing bogus arguments from *randomize arguments*, where the latter is trivial to remove for a compiler. When comparing Clang with GPT-4, we find that Clang outperforms GPT-4 across all transformations except *encode branches*, where a direct comparison is not possible as discussed next.

Table 2 shows the syntactical and semantical correctness. Similar to deobfuscation performance, there is no clear winner. Both fine-tuned models achieve high syntactical correctness and, to a lower degree, also semantical correctness (between 50 and 94.5%). Interestingly, GPT-4 outperforms the fine-tuned models in terms of semantical correctness. This suggests that larger models such as GPT-4 may possess stronger general code reasoning capabilities. As expected, Clang generally outperforms all models, with the exception of *encode branches*, where it fails to maintain semantical correctness in nearly all cases. When manually inspecting these cases, it appears that Tigress introduces subtle undefined behaviors, which Clang exploits to perform aggressive, yet *incorrect*, optimizations.

Finally, we compare the fine-tuned code models with their unmodified base models. DeepSeek Coder struggles with correctness, with only 21.60 % being syntactically and 16.10 % syntactically correct on average. Code Llama, on the other hand, shows better correctness, with 93.50 % being syntactically and 92.20 % semantically correct on average. However, it only achieves an average complexity reduction of 0.001.

Table 2: **Correctness rates for the different models by transformation type.** We report semantical and syntactical correctness. For *encode branches*, the set of joint semantically correct samples is 0 when Clang is included, and thus we only compare the three other models in this case.

	Correctness	DeepSeek Coder	Code Llama	GPT-4	Clang
Encode Arithmetic	Syntactical	100.00 %	100.00 %	92.00 %	100.00 %
	Semantical	69.50 %	66.00 %	77.50 %	98.00 %
Encode Branches	Syntactical	97.50 %	91.50 %	97.00 %	100.00 %
	Semantical	62.50 %	57.50 %	79.00 %	0.50 %
Flatten	Syntactical	98.50 %	96.50 %	97.50 %	100.00 %
	Semantical	54.0 %	50.00 %	76.50 %	94.00 %
Opaque Predicates	Syntactical	100.00 %	99.50 %	92.00 %	100.00 %
	Semantical	94.50 %	93.50 %	89.50 %	95.00 %
Randomize Arguments	Syntactical	96.50 %	97.00 %	100.00 %	100.00 %
	Semantical	93.00 %	96.00 %	98.50 %	98.50 %

Conclusion: Models fine-tuned on specific transformations demonstrate strong deobfuscation performance. They can outperform large generalist models in reducing complexity and achieving overall correctness, but large generalist models can have some advantages in maintaining semantical correctness. LLMs outperform compiler optimizations for most obfuscation transformations. Surprisingly, considering syntactical correctness, LLMs get close to compilers — which never fail. However, their primary challenge is to ensure semantical correctness.

4.2 Multiple transformations

So far, we have been focusing on whether the models can learn individual transformations during fine-tuning. Building upon this, we want to investigate if the models are also capable of learning *multiple* transformations simultaneously.

Setup. To do this, we fine-tune the two code models on the entire dataset consisting of multiple transformations with $15,000$ samples in total. This allows us to explore if the capacity of our chosen model sizes is sufficient to exhibit enough in-depth code understanding capabilities to handle multiple transformations of different natures and various structural changes at the same time.

Results. We find that both versions of the fine-tuned models maintain strong deobfuscation performance for all the transformations, even when fine-tuned on multiple transformations at once. However, we notice a drop in performance of around 13 % for randomized arguments for both models. This indicates that the models might have a tendency to fail to recognize specific transformations

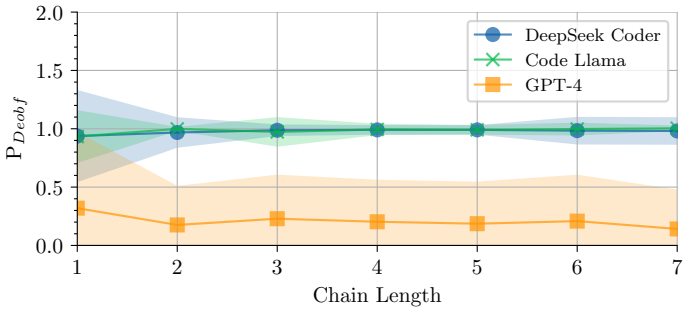

Fig. 6: Average deobfuscation scores P_{Deobf} for chained transformations of different lengths

when trained on multiple transformations at the same time. Also, we observe that for the models trained on all transformations, the correctness for *encode branches* increases for both models. We suspect that this is a result of the LLMs' improved general understanding of the deobfuscation task since they had seen more data during training. On the other hand, for Code Llama, specifically, the syntactical correctness for *opaque predicates* decreases when fine-tuned on multiple transformations. A possible explanation for this behavior is that the model is confusing different transformations. At the same time, the semantical correctness for *encode arithmetic* increases for Code Llama, indicating no clear trend toward improvement or degradation

Scaling chains of transformations. Next, we want to scale up the experiments to better understand the potential failure points of LLMs regarding code understanding and systematically measure how much the models maintain their performance with increasing complexity.

Setup. For this purpose, we build a training data set with transformations of chain lengths from one to five, i.e., $L_{max} = 5$, consisting of 3,000 samples for each chain size, resulting in a dataset of 15,000 samples in total. We allow the same transformation to be chosen multiple times in a chain, which enables a diverse data set with $5^5 = 3125$ possible transformation chains. As before, we randomize the parameters of each transformation according to the recommendations in the Tigress documentation. For testing, we consider transformations from chain lengths one to seven, i.e., $L_{max} = 7$. With chain lengths six and seven, we evaluate the performance of out-of-training samples.

Results. Figure 6 shows the code deobfuscation performance P_{Deobf}. In three cases, specifically in the chained transformation scenario, we find that Code Llama renamed the function for deobfuscation, although not trained to do so for our models, and GPT-4 explicitly instructed not to do so. We exclude these samples from the evaluation since our pipeline relies on identical function names for obfuscated and deobfuscated samples. Renamed functions could result in

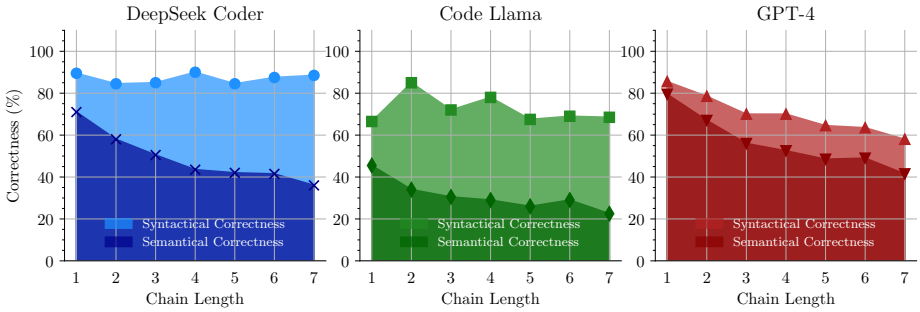

Fig. 7: Correctness rate for chained transformations

erroneously computing the metrics over an auxiliary function in the code file for the deobfuscated sample, resulting in nonsensical scores.

We find that DeepSeek Coder and Code Llama maintain stable deobfuscation performance, even for chain lengths six and seven that were not part of the fine-tuning. On the other hand, the deobfuscation performance of GPT-4 starts lower and declines slightly with increasing chain lengths, and it is significantly outperformed by both code models for all possible chain lengths.

The correctness rates are reported in Figure 7. For syntactical correctness, the rates for the code models vary with increasing chain length but still exhibit high correctness rates. On the other hand, the syntactical correctness rate for GPT-4 steadily declines. For semantical correctness, we see that GPT-4 performs the best, followed by DeepSeek Coder. GPT-4 maintains a higher semantical correctness rate for larger chain sizes. This again supports our suspicion that its larger model size can attenuate semantical correctness problems of LLMs.

Conclusion: The fine-tuned models consistently maintain stable deobfuscation performance across all evaluated chain lengths, including lengths six and seven, which were not included in the training data. In contrast, the performance of GPT-4 decreases significantly as the chain length increases.

4.3 Memorization

The LLMs we consider are trained on large amounts of text and code that is publicly available online [1, 16, 33]. This inevitably raises the question of whether the training data contained tigress-obfuscated code, which can be found on platforms such as *Stack Overflow*. As a result, the model's deobfuscation abilities could stem from either memorization of the correct outputs or a genuine understanding of the code's structure and semantics. To investigate this, we propose the following experiment: We identify and alter constant values within a program. If the model recovers the original, unmodified constant values during deobfuscation, this would strongly suggest memorization. On the other hand, correctly recovered samples would indicate more genuine understanding capabilities.

Setup. We use samples from the previous experiment that were semantically correct and contain at least one constant, excluding constants in array declarations and references from the randomization procedure, as these likely cause the program to malfunction. For the remaining samples, we randomize all constants in the program. We exclude programs that crash or time out after ten milliseconds. The latter occurs if the randomization of constants causes a slow or infinite loop. Lastly, we update the corresponding input and output samples using the new program for reference. In total, we collect 257 programs for DeepSeek Coder, 215 samples for Code Llama, and 357 for GPT-4.

Results. We observe that the semantical correctness rate ranges between 75 and 99 % across the five transformations and two models. If a deobfuscated sample is semantically correct, it is very likely to have recovered the correct constants. Therefore, we focus on the incorrect samples and manually review these.

During this analysis, we found only one sample affected by memorized constants out of the 257 for DeepSeek Coder and the 255 for Code Llama. However, we did observe that models were frequently confused by additional arithmetic complexity, as introduced by *encode arithmetic*, which was especially pronounced for DeepSeek Coder. Also, operators such as $\leq$ and $\neq$ were frequently changed. Furthermore, in several instances, the models attempted to correct nonsensical code, such as loops that are never executed or mutually exclusive logical compound conditions in if-statements, which, according to the updated I/O samples, resulted in semantically incorrect code. For GPT-4, we noticed a lower average semantical correctness over all transformations, with 83 % for GPT-4 vs. 88 % for DeepSeek Coder and 93.48 % for Code Llama. A possible explanation is that due to the lack of training in removing specific transformations, GPT-4 might remove only part of the obfuscated code and break semantics in the process as compared to the other two. Code Llama's better score might indicate a lower tendency to try to "correct" implausible statements and thus break semantical correctness.

Conclusion: We only observed minor indications of the LLMs using memorized constant code snippets for deobfuscation, which is a good indication of the inference ability of the models for deobfuscation.

5 Discussion

This paper represents an initial investigation of the capabilities of LLMs for deobfuscation. In the following, we discuss our findings and potential directions for future work in this area.

Our semantical correctness check. For checking the semantical correctness, we rely on I/O sampling and, more specifically, the *rich IO* samples from the ExeBench dataset [2], which can still, in theory, miss corner cases. We did not observe failed correctness checks during our experiments. As we have full access to both the original and obfuscated samples, this check could easily be extended

through standard techniques such as differential testing or symbolic execution. However, we do not expect a significant difference in results.

Semantical incorrectness in practice. Our study shows that, at the moment, LLMs can sometimes produce semantically incorrect results. While this is not an ideal situation and may clearly hinder some applications, these results can still be useful in some scenarios. Similarly, decompilers are known to not always be correct [27], but they are often considered useful by practitioners. We apply 8-bit quantization during fine-tuning to improve efficiency, though this can reduce output quality [12]. Furthermore, our experiments with GPT-4 suggest that larger models maintain semantical correctness better. Exploring the impact of model size and model quantization could be a promising direction for future research.

Context size. We use a 6144-token context size to balance program length with training and inference speed, limiting the size of programs we consider. We consider compressing the programs into a more compact representation as out of scope for this work to exclude potential complications that might arise from compressing larger inputs to fit the context size.

Transformations. We focus on a subset of existing obfuscation methods. Specifically, obfuscation schemes such as virtual machine-based packing or self-modifying code are hard to deobfuscate solely with static analysis [11, 35]. These types of obfuscation could be addressed using different approaches, such as dynamic techniques, to recover a dump of the code, which could then be deobfuscated using the method we presented. Additionally, our approach only includes intra-procedural obfuscation. Evaluating other obfuscation schemes, as well as inter-procedural obfuscation, is a promising direction for future work.

About deobfuscation and LLMs. We believe that deobfuscation is a great application for LLMs for several reasons: (1) Semantical correctness of the obfuscated and the LLM-simplified code can be automatically checked (at least partially), allowing for a clear evaluation of the benefit of LLMs as well as a simple safety net against hallucinations. (2) Generating datasets is easy, as obfuscators can be naturally turned into example generators. (3) Simplifying such convoluted codes requires some form of clear understanding of the code. (4) Highly obfuscated codes combining several layers of protection are less likely to be part of the initial training dataset, reducing the risk of memorization.

General Applicability. We showed that models trained on multiple transformations still show high deobfuscation performance. While our experiment showed an increase in performance for some transformations, it showed a decrease for others. We expect that with increasing training and model size, the performance gains will outweigh potential drawbacks due to LLMs confusing different transformations. This indicates the tendency of LLMs to become universal deobfuscators, possibly rendering the necessity of dedicated deobfuscation techniques obsolete.

Open questions. While we believe this work already gathered valuable insights regarding the potential of LLMs for code deobfuscation, several important questions remain, including generalization across different obfuscators and their

different versions, other programming languages—particularly machine code or bytecode—and possible countermeasures. Another direction would be extending the memorization experiment, for example, finding different representations of the program and evaluating the representation returned by the LLM.

6 Related Work

In the following, we discuss related work on applications of code LLMs and other approaches to employing learning-based methods for code deobfuscation.

Large Language Models for Codes. LLMs have advanced various fields, including natural language processing [1, 12] and programming languages [15, 33]. Feng et al. [15] propose the CodeBERT model that utilizes an encoder-only architecture with the primary focus on code classification, code retrieval, and program repair. CodeT5 [40] and CodeT5+ [39] employ an encoder-decoder architecture with various datasets and objective functions to tackle various code generation tasks. More recently, LLMs with decoder-only architecture have shown promising performance in various code generation tasks [16, 33].

Machine Learning for Code Deobfuscation. Most deobfuscation algorithms, like symbolic deobfuscation, target specific obfuscation families. It relies on static analysis to remove, for example, opaque predicates [3] or to simplify virtual machines [34]. Closer to our work, neural networks have been used to identify obfuscated code parts, and the obfuscation used [36]. Our work is more general, as it performs obfuscation identification and simplification in a row. Hence, both aspects can benefit from prior knowledge included in our fine-tuned LLM.

7 Conclusion

In this paper, we present an exploration into the use of LLMs for the task of code deobfuscation by conducting three main experiments. First, we test the LLMs in a single transformation setting in which our models show strong performance. On the downside, we find that challenges related to maintaining semantical correctness persist, indicating areas for future improvement. Second, we consider a scenario closer to real-world conditions by employing chains of transformations. As the length of these transformation chains increases, the models' ability to produce semantically correct code decreases. However, the models' deobfuscation performance remains consistently strong across all considered chain lengths. Compared to GPT-4, our models maintain higher deobfuscation performance for longer chains. Finally, we perform a memorization experiment, which all models successfully pass. As we continue to refine these models and address their shortcomings in the future, the prospect of developing more robust, scalable, and versatile deobfuscation tools based on LLMs becomes more tangible and promises to enhance security efforts in the ever-evolving arms race in software protection.

Acknowledgments. This work was supported by the German Federal Ministry of Education and Research (BMBF) under the grant AIgenCY (16KIS2012), the

Deutsche Forschungsgemeinschaft (DFG, German Research Foundation) under the project ALISON (492020528), the European Research Council (ERC) under the consolidator grant MALFOY (101043410), ANR Research under Plan France 2030 with reference ANR-22-PECY-0007 as well as BPI under Plan France 2030 with reference DOS0233319/00.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

References

1. Achiam, J., Adler, S., Agarwal, S., Ahmad, L., Akkaya, I., Aleman, F.L., Almeida, D., Altenschmidt, J., Altman, S., Anadkat, S., et al.: Gpt-4 technical report. arXiv preprint (2023). https://doi.org/10.48550/arXiv.2303.08774
2. Armengol-Estapé, J., Woodruff, J., Brauckmann, A., Magalhães, J.W.d.S., O'Boyle, M.F.: Exebench: an ml-scale dataset of executable c functions. In: Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming (MAPS). p. 50–59 (2022). https://doi.org/10.1145/3520312.3534867
3. Bardin, S., David, R., Marion, J.Y.: Backward-bounded dse: Targeting infeasibility questions on obfuscated codes. In: IEEE Symposium on Security and Privacy (S&P). pp. 633–651 (2017). https://doi.org/10.1109/SP.2017.36
4. Blazytko, T., Contag, M., Aschermann, C., Holz, T.: Syntia: Synthesizing the semantics of obfuscated code. In: USENIX Security Symposium. pp. 643–659 (2017), https://dl.acm.org/doi/10.5555/3241189.3241240
5. Collberg, C.: RandomizeArguments — tigress.wtf. https://tigress.wtf/randomizeArguments.html, Accessed: 30 April 2025
6. Collberg, C.: Recipes — tigress.wtf. https://tigress.wtf/recipes.html, Accessed: 30 April 2025
7. Collberg, C.: The Tigress C Diversifier/Obfuscator. https://tigress.wtf/index.html, Accessed: 30 April 2025
8. Collberg, C., Thomborson, C., Low, D.: A taxonomy of obfuscating transformations. Tech. rep., The University of Auckland, New Zealand (1997)
9. Collberg, C.S., Nagra, J.: Surreptitious Software - Obfuscation, Watermarking, and Tamperproofing for Software Protection. Addison-Wesley (2010), https://dl.acm.org/doi/10.5555/1594894
10. Collberg, C.S., Thomborson, C.D., Low, D.: Manufacturing cheap, resilient, and stealthy opaque constructs. In: ACM Symposium on Principles of Programming Languages (POPL). p. 184–196. https://doi.org/10.1145/268946.268962
11. Coogan, K., Lu, G., Debray, S.: Deobfuscation of virtualization-obfuscated software: A semantics-based approach. In: ACM Conference on Computer and Communications Security (CCS). p. 275–284 (2011). https://doi.org/https://doi.org/10.1145/2046707.2046739
12. Dettmers, T., Lewis, M., Belkada, Y., Zettlemoyer, L.: Llm.int8(): 8-bit matrix multiplication for transformers at scale. In: Advances in Neural Information Processing Systems (NeurIPS). pp. 30318–30332 (2022), https://dl.acm.org/doi/10.5555/3600270.3602468
13. Eyrolles, N., Goubin, L., Videau, M.: Defeating mba-based obfuscation. In: Proceedings of the 2016 ACM Workshop on Software PROtection. p. 27–38 (2016). https://doi.org/10.1145/2995306.2995308

14. Fan, Z., Gao, X., Mirchev, M., Roychoudhury, A., Tan, S.H.: Automated Repair of Programs from Large Language Models. In: International Conference on Software Engineering (ICSE). p. 1469–1481 (2023). https://doi.org/10.1109/ICSE48619.2023.00128
15. Feng, Z., Guo, D., Tang, D., Duan, N., Feng, X., Gong, M., Shou, L., Qin, B., Liu, T., Jiang, D., Zhou, M.: CodeBERT: A pre-trained model for programming and natural languages. In: Findings of the Association for Computational Linguistics (EMNLP). pp. 1536–1547 (2020). https://doi.org/10.18653/v1/2020.findings-emnlp.139
16. Guo, D., Zhu, Q., Yang, D., Xie, Z., Dong, K., Zhang, W., Chen, G., Bi, X., Wu, Y., Li, Y., et al.: Deepseek-coder: When the large language model meets programming–the rise of code intelligence. arXiv preprint (2024). https://doi.org/10.48550/arXiv.2401.14196
17. Hajipour, H., Hassler, K., Holz, T., Schönherr, L., Fritz, M.: Codelmsec benchmark: Systematically evaluating and finding security vulnerabilities in black-box code language models. In: IEEE Conference on Secure and Trustworthy Machine Learning (SaTML). pp. 684–709 (2024). https://doi.org/10.1109/SaTML59370.2024.00040
18. Hajipour, H., Malinowski, M., Fritz, M.: Ireen: Reverse-engineering of black-box functions via iterative neural program synthesis. In: Machine Learning and Principles and Practice of Knowledge Discovery in Databases (ECML PKDD). pp. 143–157 (2021). https://doi.org/10.1007/978-3-030-93733-1_10
19. Hajipour, H., Schönherr, L., Holz, T., Fritz, M.: Hexacoder: Secure code generation via oracle-guided synthetic training data. In: arXiv preprint (2024). https://doi.org/10.48550/arXiv.2409.06446
20. Halstead, M.H.: Elements of Software Science (Operating and programming systems series). Elsevier Science Inc. (1977), https://dl.acm.org/doi/10.5555/540137
21. Hammad, M., Garcia, J., Malek, S.: A large-scale empirical study on the effects of code obfuscations on android apps and anti-malware products. In: International Conference on Software Engineering (ICSE). p. 421–431 (2018). https://doi.org/10.1145/3180155.3180228
22. Jiang, N., Liu, K., Lutellier, T., Tan, L.: Impact of code language models on automated program repair. In: International Conference on Software Engineering (ICSE). pp. 10 pp.–54 (2023). https://doi.org/10.1109/ICSE48619.2023.00125
23. Katzmarski, B., Koschke, R.: Program complexity metrics and programmer opinions. In: 20th IEEE International Conference on Program Comprehension (ICPC). pp. 17–26 (2012). https://doi.org/10.1109/ICPC.2012.6240486
24. Lattner, C.: Llvm and clang: Next generation compiler technology. In: The BSD conference. pp. 1–20 (2008)
25. Linn, C., Debray, S.K.: Obfuscation of executable code to improve resistance to static disassembly. In: ACM Conference on Computer and Communications Security (CCS). p. 290–299 (2003). https://doi.org/10.1145/948109.948149
26. Liu, B., Shen, J., Ming, J., Zheng, Q., Li, J., Xu, D.: Mba-blast: Unveiling and simplifying mixed boolean-arithmetic obfuscation. In: USENIX Security Symposium. p. 2351–2365 (2021)
27. Liu, Z., Wang, S.: How far we have come: testing decompilation correctness of C decompilers. In: Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis(ISSTA). p. 475–487 (2020). https://doi.org/10.1145/3395363.3397370
28. McCabe, T.J.: A complexity measure. TSE pp. 308–320 (1976). https://doi.org/10.1109/TSE.1976.233837

29. Menguy, G., Bardin, S., Bonichon, R., Lima, C.d.S.: Search-based Local Black-box Deobfuscation: Understand, Improve and Mitigate. In: ACM Conference on Computer and Communications Security (CCS). p. 2513–2525 (2021). https://doi.org/10.1145/3460120.3485250
30. Ming, J., Xu, D., Wang, L., Wu, D.: Loop: Logic-Oriented Opaque Predicate Detection in Obfuscated Binary Code. In: ACM Conference on Computer and Communications Security (CCS). p. 757–768 (2015). https://doi.org/10.1145/2810103.2813617
31. Reichenwallner, B., Meerwald-Stadler, P.: Efficient deobfuscation of linear mixed boolean-arithmetic expressions. In: CheckMATE workshop. p. 19–28 (2022). https://doi.org/10.1145/3560831.3564256
32. Reichenwallner, B., Meerwald-Stadler, P.: Simplification of general mixed boolean-arithmetic expressions: GAMBA. In: IEEE European Symposium on Security and Privacy (EuroS&P) Workshops. pp. 427–438 (2023). https://doi.org/10.1109/EuroSPW59978.2023.00053
33. Roziere, B., Gehring, J., Gloeckle, F., Sootla, S., Gat, I., Tan, X.E., Adi, Y., Liu, J., Remez, T., Rapin, J., et al.: Code llama: Open foundation models for code. arXiv preprint (2023). https://doi.org/10.48550/arXiv.2308.12950
34. Salwan, J., Bardin, S., Potet, M.: Symbolic deobfuscation: From virtualized code back to the original. In: Detection of Intrusions and Malware, and Vulnerability Assessment (DIMVA). pp. 372–392 (2018). https://doi.org/10.1007/978-3-319-93411-2_17
35. Schrittwieser, S., Katzenbeisser, S., Kinder, J., Merzdovnik, G., Weippl, E.: Protecting software through obfuscation: Can it keep pace with progress in code analysis? ACM Computing Surveys (CSUR) (2016). https://doi.org/10.1145/2886012
36. Tofighi-Shirazi, R., Asăvoae, I.M., Elbaz-Vincent, P.: Fine-grained static detection of obfuscation transforms using ensemble-learning and semantic reasoning. In: Proceedings of the 9th Workshop on Software Security, Protection, and Reverse Engineering (SSPREW). pp. 1–12 (2019). https://doi.org/10.1145/3371307.3371313
37. Udupa, S.K., Debray, S.K., Madou, M.: Deobfuscation: Reverse Engineering Obfuscated Code. In: 12th Working Conference on Reverse Engineering (WCRE'05). pp. 10 pp.–54 (2005). https://doi.org/10.1109/WCRE.2005.13
38. Wang, C., Hill, J., Knight, J., Davidson, J.: Software tamper resistance: Obstructing static analysis of programs. Tech. rep., University of Virginia (2000), https://dl.acm.org/doi/10.5555/900898
39. Wang, Y., Le, H., Gotmare, A., Bui, N., Li, J., Hoi, S.: Codet5+: Open code large language models for code understanding and generation. In: Conference on Empirical Methods in Natural Language Processing (EMNLP). pp. 1069–1088 (2023). https://doi.org/10.18653/v1/2023.emnlp-main.68
40. Wang, Y., Wang, W., Joty, S., Hoi, S.C.: Codet5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. In: Conference on Empirical Methods in Natural Language Processing (EMNLP). pp. 8696–8708 (2021). https://doi.org/10.18653/v1/2021.emnlp-main.685
41. Yadegari, B., Johannesmeyer, B., Whitely, B., Debray, S.: A generic approach to automatic deobfuscation of executable code. In: IEEE Symposium on Security and Privacy (S&P). pp. 674–691 (2015). https://doi.org/10.1109/SP.2015.47