Quantifying and Mitigating the Impact of Obfuscations on Machine-Learning-Based Decompilation Improvement

Luke Dramko[1(✉)], Deniz Bölöni-Turgut[2], Claire Le Goues[1], and Edward Schwartz[3]

[1] Carnegie Mellon University, Pittsburgh, PA 15213, USA
{lukedram,clegoues}@cs.cmu.edu
[2] Cornell University, Ithaca, NY 14850, USA
db823@cornell.edu
[3] Carnegie Mellon University Software Engineering Institute, Pittsburgh, PA 15213, USA
eschwartz@cert.org

Abstract. Decompilers are tools that reverse the process of compilation, converting executable binaries into a high-level language like C. They are useful in situations where the original source code is unavailable, such as when analyzing malware, doing vulnerability research, and patching legacy software. Unfortunately, decompilation is necessarily incomplete, because the compiler discards many of the abstractions that make source code readable, like identifier names and types. A large body of existing work uses machine learning to predict missing names, types, and other abstractions in decompiled code. However, little of this work considers *obfuscations*: semantics-preserving transformations that obscure the functionality and design of a program. At the same time, obfuscations are common in practice, especially in malware. In this work, we perform a quantitative analysis of the impact that obfuscations have on decompiled code. Further, we investigate the degree to which training on obfuscated code mitigates the impact of obfuscations. We perform our experiments on three different models from the literature: DIRTY, HexT5, and VarBERT. We find that obfuscations do negatively impact machine learning models, but training on obfuscations can partially help recover lost accuracy.

Keywords: Decompilation · Reverse Engineering · Machine Learning

1 Introduction

A *decompiler* is a tool that reverses the process of compilation, converting executable binary programs into a high level language such as C. Decompilers are useful for a variety of security related tasks, including malware analysis, vulnerability research, and patching legacy software [37,38]. Source code is a dual-channel medium, containing a formal channel that specifies execution semantics,

M. Egele et al. (Eds.): DIMVA 2025, LNCS 15747, pp. 244–266, 2025.
https://doi.org/10.1007/978-3-031-97620-9_14

and a natural language channel that communicates information to developers who read and write the code [5].

```
1  int uv_exepath(char* buffer, size_t* size){
2    if (!buffer || !size) {
3      return -1;
4    }
5
6    *size = readlink("/proc/self/exe", buffer
           , *size - 1);
7    if (*size <= 0)
8        return -1;
9    buffer[*size] = '\0';
10   return 0;
11 }
```

(a) A function that returns the filename of the currently running process.

```
1  __int64 __fastcall <func>(char *buf,
       ssize_t *a2){
2    ssize_t v2;
3    if (!buf) return 0xFFFFFFFFLL;
4    if (!a2) return 0xFFFFFFFFLL;
5    v2 = readlink("/proc/self/exe", buf, *
         a2 - 1);
6    *a2 = v2;
7    if (!v2) return 0xFFFFFFFFLL;
8    buf[v2] = 0;
9    return 0LL;
10 }
```

(b) The same function as 1a, but after being compiled and then decompiled by IDA Pro.

Fig. 1. Decompiled code is harder to read than original source code.

Computers, however, only require the formal channel, and so compilers discard the abstractions in the natural channel during compilation. As a result, traditional decompilers struggle to recover many of the natural abstractions that make source code readable, such as variable names, types, comments, and some aspects of code structure [12]. This makes reverse engineering slow and painstaking [37,38].

To make matters worse, some targets of decompilation—especially malware—are intentionally *obfuscated*: they are transformed to obscure the functionality and design of a program (in other words, making it more difficult to comprehend) without changing the program's behavior. Figure 1a shows a simple function, and Fig. 1b shows the function after being compiled and then decompiled. Figure 2a –2d show the same function after applying an obfuscation, compilation, and decompilation. For instance, in Fig. 2a, the string literal `"/proc/self/exe"`, which provides a key clue to what the function does, has been replaced with a sequence of operations on a collection of variables that obscure the content of the string. In Fig. 2c, control flow has been completely restructured so that it is difficult to tell what statements are executed in what order. As these examples demonstrate, decompilers typically do not undo obfuscations; they simply propagate the obfuscation from the binary level to the source level.

Recently, researchers have turned to machine learning models to *probabilistically predict* missing abstractions in decompiled code [2,6,21,26,32,41,43], such as variable names and types. These techniques are based on the principle that software is *natural*, or predictable given context [17]. For example, it is possible to predict a variable's name based on how it is used. These models take as input an executable or a representation of the executable that can be deterministically derived from it—such as disassembly or the output of a deterministic decompiler—and output one or more natural-channel abstractions. We collectively refer to these as *decompilation improvement* tasks.

```
__int64 __fastcall <func>(char *buf,
      ssize_t *a2) {
    __int64 i, len, v9;
    ssize_t v3;
    char path[8], v8[8], v11[8];
    int v7, v10;
    if (!buf) return 0xFFFFFFFFLL;
    if (!a2) return 0xFFFFFFFFLL;
    len = *a2 - 1;
    v9 = 0x4958054A4755560ALL;
    v10 = 1426081857;
    strcpy(v11, "IW");
    *(_QWORD *)path = 0x4958054A4755560ALL;
    v7 = 1426081857;
    strcpy(v8, "IW");
    for (i = 0LL; i != 14; ++i)
        path[i] ^= (_BYTE)i + 37;
    v8[2] = 0;
    v3 = readlink(path, buf, len);
    *a2 = v3;
    if (!v3) return 0xFFFFFFFFLL;
    buf[v3] = 0;
    return 0LL;
}
```

(a) Figure 1a, obfuscated with ADVobfuscator [1] string obfuscation (`str`). The string literal is represented as integers that are transformed by a convoluted series of operations.

```
__int64 __fastcall <func>(char *buf,
      ssize_t *a2) {
    unsigned int v2;
    ssize_t v3;
    v2 = -1;
    if (buf != 0LL && a2 != 0LL) {
        v3 = readlink("/proc/self/exe", buf
             ,
               *a2 - 0x3EBD892878945E8LL +
                     0x3EBD892878945E7LL);
        *a2 = v3;
        if (v3) {
            buf[v3] = 0;
            return 0;
        }
    }
    return v2;
}
```

(b) Figure 1a, obfuscated with instruction substitution (`sub`) [22]. The subtraction instruction on line 5 of Figure 1b is replaced with an equivalent but more convoluted sequence of operations above.

```
__int64 __fastcall <func>(char *buf,
      ssize_t *a2) {
    unsigned int v2;
    int i, v4;
    ssize_t v6;
    for (i = -379896799;; i = 1196796914) {
        while (i <= -157289568) {
            v4 = -2108226211;
            if (a2) v4 = -1403965279;
            if (!buf) v4 = -2108226211;
            if (v4 == -1403965279) {
                v6 = readlink("/proc/self/
                     exe", buf, *a2 - 1);
                *a2 = v6;
                i = -157289567;
                if (!v6) i = 1558099169;
            } else {
            LABEL_5:
                i = 1196796914;
                v2 = -1;
            }
        }
        if (i != -157289567) break;
        buf[v6] = 0;
        v2 = 0;
    }
    if (i != 1196796914) goto LABEL_5;
    return v2;
}
```

(c) Figure 1a, obfuscated with control flow flattening (`fla`) [22]. The functions' control flow is rearranged so that basic blocks are arranged in a loop and a control variable (`i`, above) determines which blocks are executed on each iteration.

```
__int64 __fastcall <func>(char *buf,
      ssize_t *a2) {
    ssize_t v2;
    __int64 result;
    if (buf && a2) {
        v2 = readlink("/proc/self/exe", buf
             , *a2 - 1);
        *a2 = v2;
        if (v2) {
            buf[v2] = 0;
            result = 0LL;
            if (y_26 < 10) return result;
            goto LABEL_11;
        }
        result = 0xFFFFFFFFLL;
    } else {
        result = 0xFFFFFFFFLL;
        if (y_26 >= 10 && (((_BYTE)x_25 *
            ((_BYTE)x_25 - 1)) & 1) != 0)
            {
            while (1);
        }
    }
    if (y_26 < 10) return result;
LABEL_11:
    if ((((_BYTE)x_25 * ((_BYTE)x_25 - 1))
          & 1) != 0) {
        while (1);
    }
    return result;
}
```

(d) Figure 1a, obfuscated with bogus control flow (`bcf`) [22]. Extra control-flow constructs like if-statements, while loops, and `goto`s are added.

Fig. 2. Figure 1a, obfuscated in different ways, then decompiled. The obfuscated versions are more difficult to read than without obfuscations Fig. 1b. Function names are normalized to `<func>`, as is the convention in HexT5 [41].

In this paper, we analyze the impact that obfuscations have on the accuracy of decompilation-improvement machine-learning models. Further, we quantify the impact that training on obfuscated code has on a model's ability to handle obfuscations. While obfuscation is commonly employed by malware in practice, virtually no existing work in ML-based decompilation improvement considers obfuscations in training or evaluation. This is concerning because obfuscation undermines the *naturalness* assumption on which these techniques are based. By its nature, obfuscation changes the context under which variable names, types, and other abstractions occur. For instance, on Fig. 1a, line 6, the size of the buffer is decremented by 1 to accommodate the null terminator that must be present at the end of all C strings. Decrementing a string's length for the null terminator is common in C. In contrast, subtracting `0x3EBD892878945E8` and adding `0x3EBD892878945E7`, as occurs under the instruction substitution obfuscation in Fig. 2b, while semantically equivalent, is syntactically unusual; it is not what would normally be predicted given the surrounding context, and thus can be considered *unnatural*. We expect that this should undermine machine-learning-based tools whose models are only "familiar with" unobfuscated code. However, it may also be possible to learn a model that is robust to the presence of obfuscations by training on those obfuscations, making the obfuscations an expected, or at least not unexpected, part of the context.

In this paper, we answer four research questions:

1. How much does the presence of obfuscations impact the accuracy of ML-based decompilation improvement models?
2. How difficult is decompilation improvement under each type of obfuscation?
3. How well does learning transfer from one type of obfuscation to another?
4. How does varying the amount of obfuscata in training affect model performance?

In particular, we perform our experiments on three models covering three different decompilation improvement tasks: DIRTY [6], a model that predicts variable names and types, VarBERT [32], a model which only predicts variable names, and HexT5 [41], a language model which can solve a variety of tasks, but which we use to predict variable and function names. We answer our research questions by performing a series of experiments in which we train and evaluate machine learning models on unobfuscated and obfuscated code from a novel, large-scale dataset of obfuscated and unobfuscated executable binaries.

In short, we contribute:

- Four experiments answering our research questions involving 30 trained machine learning models which quantify the impact that obfuscations have on machine-learning-based decompilation improvement.
- A tool for building datasets including obfuscations at scale.
- A novel dataset consisting of unobfuscated and obfuscated binaries, with up to four obfuscations per binary.

We make available the code and data used in our experiments.

2 Research Questions and Findings

Our high level research goal is to understand and mitigate the effect of intentional software obfuscation on decompilation improvement models. In machine learning parlance, unobfuscated code and obfuscated code represent different distributions. Using a neural model trained on unobfuscated code to decompile obfuscated code represents a *covariate shift* [33]. Note that naturalness [17] is defined with respect to a distribution, because what is "natural" is context dependent: a natural-sounding sentence in American English may sound unnatural in British English. Meanwhile, some distributions may be closer to each other than others, i.e., two English dialects may be closer to one another than either is to Spanish. Here, different obfuscations may transform code in similar ways, and distributions for code obfuscated by these obfuscations might be closer to one another than they are to that of a dissimilar obfuscation. While it is difficult to measure the distances between distributions directly, we can instead indirectly measure it by quantifying the performance of a model trained on one distribution when evaluated on another. This idea underlies our high-level approach: we train and evaluate models on different code distributions, represented by different obfuscations, to understand their impact on decompilation improvement models.

In RQ1, (Sect. 5.1), we ask "*How much does the presence of obfuscations impact the accuracy of decompilation improvement models?*" To answer this question, we measure the performance of models of unobfuscated and obfuscated code both with and without covariate shift. We do find empirical evidence of a covariate shift that harms the performance of the three models. In particular, training solely on unobfuscated code, as virtually all decompilation improvement models do today, leads to poorer performance on obfuscated code, which is found in many real-world reverse-engineering scenarios. Fortunately, training on obfuscated code alleviates most or all of the impact by eliminating the covariate shift.

Code obfuscated in different ways may represent different distributions. In RQ1, we trained the obfuscated-code model to learn a combined distribution of all obfuscations. But some obfuscations' individual distributions may be farther away from the distribution of unobfuscated code than others, leading to poorer performance. Driven by this intuition, in RQ2 (Sect. 5.2), we ask "*How difficult is decompilation improvement under each type of obfuscation?*" We find that control flow flattening is the most difficult obfuscation for a model trained on unobfuscated code.

In RQ3 (Sect. 5.3), we further quantify the differences between individual obfuscation's distributions. We ask "*How well does learning transfer from one type of obfuscation to another?*" It is possible that the distributions for obfuscations that are similar are close to one another, and that training on one obfuscation means the performance on a related one may be relatively good. In other words, the learning transfers well between the two obfuscations. Unfortunately, we find that learning usually transfers poorly amongst the different obfuscations in our dataset. These results imply that models may perform poorly in practice when they encounter a new obfuscation.

If in-distribution data is required to learn obfuscations, then a natural question is *how much data is required?* In RQ4 (Sect. 5.4), we ask "*How does varying the amount of obfuscated data in training affect model performance?*" Neural networks require a large amount of data to be fit well; for novel obfuscations, there will likely be little data available. However, all decompiled C code—obfuscated or not—is similar in many ways: it obeys the same syntactic rules, and the semantics assigned to that syntax is the same. This suggests that there are at least some common parts of code, regardless of obfuscation, that can be used to at least partially inform predictions on unseen obfuscations, though our results from RQ3 suggest that at least some data of a particular obfuscation is required for good performance on that obfuscation. This is reminiscent of the popular pretrain/finetune paradigm in machine learning, where a model is (pre)trained on one task for which there is much data, and then trained a little more (fine-tuned) on data for a related task. To answer RQ4, then, we start by building a dataset of unobfuscated code, the purpose of which is to provide the model with information about the syntax and semantics of C code. Then we vary the amount of obfuscated code, measuring model performance for several different base-2 orders of magnitude sizes. Model performance gains increase rapidly after the first obfuscated data are added, but drop off rapidly as more are added. This suggests that large performance gains on obfuscated code requires a substantial amount of obfuscated data.

3 Datasets

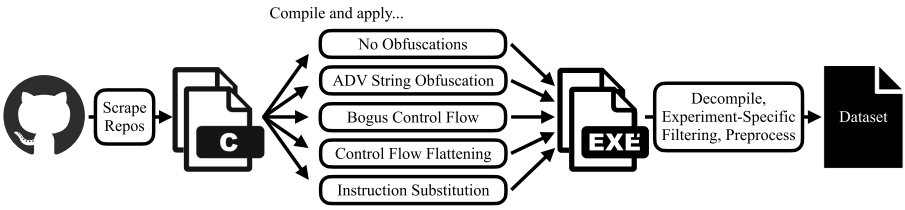

Fig. 3. An overview of our approach. We download open-source software from GitHub, then compile each project five times: once without obfuscations, once with each of our four obfuscations. In doing so, we ensure that each executable is compiled using debug information. Next, we use IDA Pro's decompiler through DIRTY's [6] dataset generation scripts to produce labeled training data. Finally, we select and preprocess data, building datasets to answer each research question.

To answer our research questions, we needed a large collection of obfuscated binaries compiled with debug information. An overview of our approach for constructing datasets with respect to the desired experiments is shown in Fig. 3. We generated training data by cloning and compiling a large collection of open-source software, in line with prior work [2,6,9,18,24,26,31,32,41]. In particular,

we targeted majority C-language repositories, though these may occasionally contain a minority of C++. We collected data from 19,552 such repositories. Unlike in prior work, however, we compiled each project five times: once with no obfuscations applied, then once with each of four obfuscations. To do this, we adapted the GitHub Cloner and Compiler (GHCC) tool [20]. GHCC automatically clones and compiles a given list of GitHub repositories, first by executing configuration scripts if they exist, then executing each Makefile found in the project. Our adapted tool, GHCC-Obfuscator, first clones, then compiles the repository without any obfuscations using the repository's original Makefile(s). It then repeats the process, applying one obfuscation at a time, resetting the repository to a freshly-cloned state in between. In performing each compilation, GHCC-Obfuscator intercepted the calls to the compiler and added the `-g` flag for debug information to each compilation command. This feature was also used to add obfuscation-specific flags as necessary. This process produces 8,081,059 unique binaries, from which we sampled to build the datasets in the experiments.

With the binaries compiled, we used DIRTY [6]'s dataset generation scripts to extract labeled data from the binaries compiled with debug information. Each binary is decompiled using IDA Pro in batch mode. Because the binaries contains debug information, its developer-provided identifier names and types are present in the decompiled code. Next, the binaries are stripped of debug symbols. The binaries are decompiled again, and this time they are missing developer-provided names and types. The two decompilations form input-output pairs for supervised training: the second decompilation is the input, and the first, the output. Offsets within the binary are used to map the variables in the decompiled code and original code together. The obfuscations we use preserve debug information, making it possible to establish ground truth in this way for them as well. Obfuscations make the functions larger on average, 297 as opposed for 249 tokens for unobfuscated code.

3.1 Obfuscations

We chose a diverse collection of obfuscations that modify the code in a variety of ways. We describe each below.

String Obfuscation with ADVobfuscator. ADVobfuscator [1] is a library of C++ header files with macros that are applied by modifying the source code. In particular, used it to obfuscate string literals found in the source code. Obfuscated strings are either encoded as an integer or as another string which is transformed by a series of operations into the original string. See Fig. 2a for an example of ADVobfuscator applied to a function. Since ADVobfuscator depends on C++ macros, we compiled any programs obfuscated with ADVobfuscator using the C++ compiler g++ (as opposed to gcc). ADVobfuscator only produced obfuscated binaries when the C files were compiled with at least an O1 optimization level. (For consistency, we also compiled all other binaries in the experiments at O1 as well). Prior work has shown that optimization levels have

a small-to-negligible impact on model performance of at most a few percentage points [6,32], likely because the deterministic decompiler undoes the optimizations when generating decompiled code. We abbreviate string obfuscation as `str` when presenting results elsewhere in the remainder of this paper.

Prior work has shown that string literals are a helpful feature of code for DIRE [13], the predecessor of DIRTY, which predicts variable names in decompiled code (but not types). It seems likely that this is a consequence of how language models (such as transformers) represent text. To input code to a language model, the input is split into a sequence of discrete tokens, each of which maps to a learned vector that encodes the semantics of that token. In general, there are more tokens allocated to natural language words and subwords then there are to C-language syntatic symbols like `*`, `(`, and `{` . As a result, natural language words carry an inflated importance in helping models reason about code; misleading natural language can substantially confuse even powerful models, even when the syntax is otherwise identical, as Miceli-Barone et al. show with identifier names [30]. Because identifier names are discarded during compilation, string literals are one of the sole sources of natural language in non-obfuscated decompiled code.

Obfuscator-LLVM Compile-Time Obfuscations. Obfuscator-LLVM [22] is a compile-time obfuscation tool based on LLVM that includes three different forms of obfuscation: instruction substitution (abbreviated `sub`), control flow flattening (`fla`), and bogus control flow (`bcf`).

Instruction substitution (`sub`) replaces arithmetic and binary operations on integers with a more complicated—but equivalent—series of operations. For example, this process may introduce random numbers into computations which cancel out because of mathematical identities. Figure 2b shows an example of a function obfuscated with instruction substitution.

Control flow flattening (`fla`) [27,39] is a form of obfuscation that implements control flow without using the traditional control structures of a programming language. Specifically, it creates a new control variable that represents which block should be executed next, and transforms each function's control flow into a loop that uses conditional statements to dispatch to the code that corresponds to the current value of the control variable. Each conditional statement body represents a basic block from the original, unobfuscated version of the code. The value of the variable determines which conditional statement body is executed; at the end of each statement body, the control variable is reset such that control flow mimics that of the original function. Figure 2c shows an example a function obfuscated with control flow flattening.

The bogus control flow (`bcf`) obfuscation inserts additional conditional statements with complex, opaque predicates that ultimately end up having no impact on the dynamic control flow of a program. In doing so, `bcf` introduces many irrelevant lines of code into the function. Figure 2d shows an example of a function obfuscated with bogus control flow.

To apply Obfuscator-LLVM to our dataset, we built the code using its original Makefiles, but forced the Obfuscator-LLVM version of the `clang` compiler to be used and specified the obfuscation's compiler flag.

3.2 Dataset Preprocessing

A particularly difficult problem in dataset preparation for decompilation-based models is data leakage. Machine learning models have the tendency to "memorize" their training data; they perform unrealistically well when evaluated on their training data. Data leakage occurs when a model is evaluated on data on which it was trained [23]. Duplicate copies of software projects can often result in data leakage when one is added to the training set and others are added to the evaluation (test) set. Because duplicate copies (e.g. forks) of software projects are extremely common on open-source hosting services like GitHub [34], careful attention must be paid to data leakage. Identifying duplicate repositories is a difficult problem, which in turn makes data leakage hard to prevent. We use the following measures to prevent data leakage:

- *By-project splitting*: All three models operate at the function level. Putting some functions from a given project in the training set and others in the test set allows for the leakage of project-specific details. As a result, data from each project are placed exclusively in either the training set or test set. Xiong et al. [41] evaluate both with and without by-project splitting; by-project splitting causes the accuracy to decrease by more than two thirds, highlighting the importance of this data leakage prevention strategy.
- *MinHashing* [4]: This is a technique for efficiently approximating the Jaccard similarity between words or sequences of words in documents. Here, we treat all of the C code in a software project as a"document." MinHashing is often used with locality-sensitive hashing (LSH) to group similar documents into "buckets"; we consider software projects that end up in the same "buckets" to be duplicates. We ensure that each project on which we evaluate is not a duplicate of any project in the training set. The Stack [25], a popular dataset of source code for training machine learning models, also uses minhashing with LSH for deduplication.
- *Binary Hashing*: Following Chen et al. [6], we hash each executable file produced by the model and ensure that models with the same binary hash do not end up in both the train and the test sets.

We control for dataset composition for each model across each experiment; that is, for each trial in each experiment, all three models are trained on the same datasets. To ensure that this is the case, we perform dataset preprocessing and splitting once (using a modified version of the DIRTY dataset preprocessor) then convert the prepared train and test sets into formats suitable for VarBERT and HexT5.

4 Models

Here, we describe the three different model types on which we perform experiments and reproduce the relevant experiments from the original papers, though on our dataset, to establish a baseline.

4.1 Architecture and Training Practices

We perform all of our experiments on three different models: DIRTY [6], VarBERT [32], and HexT5 [41].

DIRTY [6] is a decompilation improvement model with a transformer-based encoder-decoder architecture [36] that predicts variable names and types in decompiled C code. DIRTY models are trained from scratch (that is, from randomly initialized parameters).

VarBERT [32] is a decompilation improvement model based on a transformer encoder that predicts variable names in decompiled C code. VarBERT is pretrained on both a masked language modeling objective [10] and a constrained masked-language-modeling objective, before being trained on variable-name-prediction data, in line with Gu et al. [15]. We make use of the pretrained checkpoints provided by the authors, but fine-tune on our own variable-prediction data.

HexT5 [41] is emblematic of the modern trend of representation learning in natural language processing, whereby large neural networks, often transformer-architecture sequence-to-sequence models, are trained to predict parts of a sequence that are artificially hidden from the model. HexT5 is pretrained to learn representations of source code, decompiled code, assembly code, and intermediate representations. The authors evaluate it on four different tasks: summarization, function name prediction, variable name prediction, and code similarity. We evaluate only on the function name and variable name prediction tasks because our dataset does not have an oracle for textual summaries or code similarity.

Note that the choice of model is orthogonal to our research questions. While it is possible that obfuscations may affect different models in different ways, in general, we have reason to believe our findings are likely to generalize beyond the three models we evaluate here. First, like DIRTY, VarBERT, and HexT5, virtually all modern decompilation improvement models are transformer-architecture models [2,6,18,19,24,31,32,41]. Further, they are almost invariably trained on large corpora of open source code downloaded from internet repositories [2,6,9,18,24,26,31,32,41]. Finally, theory predicts that a covariate shift is expected to decrease the performance of any machine learning model, independent of architecture, so the trends in performance degredation (if not their magnitude) should generalize widely.

4.2 Baseline

(a) DIRTY

	Retyping		Renaming
	overall	structs	variables
DIRT [6]	56.4	54.6	36.9
Ours	54.5	48.8	27.7

(b) VarBERT

	Renaming
	variables
VarCorpus [32]	42.6
Ours	28.7

(c) HexT5

	Renaming	
	variable	function
NSP [41][1]	25.7	–
Ours	25.2	30.6

Fig. 4. Baseline performance numbers taken from the original papers for the three models compared with performance when trained and evaluated on our dataset. DIRT was the dataset originally used to train DIRTY; VarCorpus was originally used to train VarBERT, and NSP was the dataset used to train HexT5. All values are in percent accuracy.

To set a baseline, we reproduce results from the original DIRTY [6], VarBERT [32] and HexT5 [41] papers, but using our dataset. That is, we train and evaluate them on a subset of the full dataset consisting only of unobfuscated code.

In all cases, the scores we obtain in our reproduction are lower than the original works, sometimes significantly. Because machine learning is a "black box" method, it is difficult to conclusively determine the cause for the difference. We suspect it is due to our data-leakage-prevention measures outlined in Sect. 3.2; with less data leakage; there are fewer examples in the test set which the trained models have memorized.

DIRTY is a variable name and type prediction model. We report three metrics, as shown in Fig. 4a: the percentage of correctly predicted variable names, the percentage of correctly predicted variable types, and the percentage of correctly predicted types that are `struct`s in the original code. Structures make up a minority of variables' types in source code yet are often more important for understanding the functionality of code than primitive types.

VarBERT is a variable name prediction model. We use their IDA-O1 accuracy number because we use the IDA Pro decompiler at optimization level O1, as discussed in Sect. 3. The results are shown in Fig. 4b.

HexT5 is a language model fine-tuned on several tasks; we use it for variable name and function name prediction here. The results are shown in Fig. 4c[1].

[1] The HexT5 results reported here are imprecise estimates based on information in a bar graph in the paper; exact numbers are not provided. The original HexT5 paper reports function name prediction efficacy in terms of precision and recall. (We reached out to the authors for exact numbers but did not hear back).

5 Experiments

We evaluated the effect of introducing obfuscations to the decompilation problem over four separate experiments. For each of these experiments, we constructed several different datasets which we used to train DIRTY [6], VarBERT [32], and HexT5 [41] models. We performed model training and evaluation on NVIDIA A100 and Titan X (Pascal) GPUs and other tasks on 64-bit Linux with Intel Xenon CPUs. All three models are implemented as python programs using the `pytorch` library. For DIRTY and VarBERT, we used the dataset preprocessing, training, and evaluation scripts and environment files provided by the authors. For HexT5, these were not released, so we wrote our own, available in the replication package, using the `transformers` API [40]. For each model in each experiment, we selected training data (in compiled binary form) according to the experiment's aims, then ran DIRTY's decompilation and preprocessing scripts. We converted this data into the formats required by HexT5 and VarBERT, then ran the corresponding model-specific preprocessing scripts. With the datasets prepared, we train each model. We dedicated the bulk of the data to training but ensured that each test set, derived using the same process, contained at least 5000 examples that are drawn from repositories that do not overlap the training set. We evaluate each model with each test set as dictated by the experiments' goals.

5.1 RQ1: How much does the Presence of Obfuscations Impact the Accuracy of Decompilation Improvement Models?

Table 1. RQ1: Impact of Obfuscations on Accuracy. Results displayed in terms of percent accuracy, along with a relative percent change compared with the baseline (first row). Higher is better.

Train	Test	DIRTY				VarBERT		HexT5	
		Retyping		Renaming					
		overall		structs		variables		variables	functions
Unobf	Unobf	54.5	–	48.8	–	27.7	–	28.7	–	25.2	–	30.6	–
Unobf	Obf	51.7	(-5.1%)	44.1	(-9.6%)	21.0	(-24.0%)	20.6	(-28.4%)	23.2	(-8.0%)	26.8	(-12.3%)
Obf	Obf	56.2	(+3.3%)	46.3	(-5.2%)	27.8	(+0.3%)	25.8	(-10.1%)	27.0	(+6.9%)	29.0	(-5.3%)
Obf	Unobf	50.1	(-8.1%)	44.6	(-8.5%)	24.9	(-10.1%)	23.8	(-17.1%)	22.3	(-11.8%)	28.5	(-6.9%)

In our first experiment, we investigate the impact that obfuscations have on the performance of the three model types, and to what degree training on obfuscated data can mitigate the impact of obfuscations.

Methodology. We use our reproductions of the three models from Sect. 4.2 as a baseline against which we evaluate the impact of obfuscations. We reuse the dataset from the reproduction, but also build a dataset of obfuscated code of the same size. To provide an additional control, we attempt to use data from the same projects as in the reproduction where possible. (This may not be possible if the project failed to build with obfuscations applied.) Because we compile each repository with four different obfuscations, including all obfuscated data would result in a training set that is several times larger than the unobfuscated training set. To control for training set size, instead, we select an obfuscation uniformly at random and use data from that obfuscation, up to the amount used in the unobfuscated training set. If there is insufficient data (perhaps due to an early compilation failure on that obfuscation), we continue sampling from other obfuscations. The final sizes of the unobfuscated and obfuscated training sets are 1,627,991 and 1,578,083 functions, respectively. We train each type of model using the obfuscated dataset. Finally, we evaluate each model trained on the unobfuscated and obfuscated training sets against both of the unobfuscated and obfuscated test sets, leading to a total of four different combinations.

Results. The results are shown in Table 1. The first row contains our baseline reproductions. The second row represents what happens when these same models—trained only on unobfuscated code—are exposed to obfuscations in evaluation. This simulates what would happen if these models, as released, were applied to obfuscated code, as is commonly found in practice. In all cases, the accuracy drops, but the magnitude of the drop varies considerably. The relative drop in accuracy varies between 5.1% for overall retyping with DIRTY and 28.4% for variable renaming with VarBERT. However, training on obfuscations can help mitigate the loss in accuracy. The third row of Table 1 illustrates this. Training on obfuscated code substantially improves prediction accuracy on obfuscated code. However, there is no free lunch: a model trained solely on obfuscated code performs worse evaluated on *un*obfuscated code (row 4) than a model trained on unobfuscated code. This is perhaps because unobfuscated code is not "natural" from the perspective of an obfuscated-code-only model; unobfuscated code represents a covariate shift with respect to a model trained on obfuscated code.

Answer to RQ1: Training a model on unobfuscated code leads to poor performance on obfuscated code, but training on obfuscated code can mitigate the performance loss, at the cost of poorer performance on unobfscated code.

5.2 RQ2: How Difficult is Decompilation Improvement Under Each Type of Obfuscation?

Some obfuscations may create a more challenging context than others for name and type prediction. In this experiment, we benchmark the difficulty of each obfuscation.

Methodology. For this experiment, we used the baseline model trained on unobfuscated code from Sect. 4.2. The model was then tested on 4 disjoint subsets of the obfuscated test set from RQ1 (Sect. 5.1), where each subset contained binaries with a particular obfuscation applied. In partitioning the test set, we excluded examples that appeared under multiple obfuscations. This happens for simple functions where the obfuscations do not apply or where obfuscations fail (the latter primarily in the dataset's C++ minority). Excluding simple functions has the effect of making the task slightly more challenging; filtering C++ has the effect of increasing the difficulty of the `struct`-prediction task in particular because `struct`s are more common in C++ code, including easier-to-predict standard-library features like iterators and strings.

We use the same model's performance on the unobfuscated test set from Sect. 4.2 as a baseline.

Table 2. RQ2: Accuracy on Individual Obfuscations. Results displayed in terms of percent accuracy, along with a relative percent change compared with the baseline (first row). Higher is better.

Obfuscation	DIRTY				VarBERT	HexT5		
	Retyping		Renaming					
	overall		structs		variables	variables	variables	functions
`none`	54.5	–	48.8	–	27.7	– 28.7	– 25.2	– 30.6 –
`fla`	51.6	(-5.2%)	15.5	(-68.2%)	8.9	(-67.9%)	9.1 (-68.5%)	12.5 (-50.5%) 13.2 (-56.7%)
`sub`	50.9	(-6.5%)	37.0	(-24.2%)	17.1	(-38.3%)	24.8 (-13.5%)	19.0 (-24.9%) 19.2 (-37.3%)
`bcf`	45.6	(-16.2%)	21.7	(-55.5%)	10.6	(-61.8%)	18.6 (-35.4%)	13.5 (-46.7%) 14.8 (-51.7%)
`str`	52.8	(-3.1%)	25.6	(-47.4%)	28.3	(+2.4%)	28.4 (-0.1%)	24.2 (-4.2%) 21.5 (-29.6%)

Results. The results are summarized in Table 2. Control flow flattening generally provides the model with the most difficulty across all three tasks and models. Instruction substitution and ADVobfuscator's string obfuscation provide the least difficulty.

These results are not surprising. Control flow flattening provides the largest textual and syntactic differences from unobfuscated code, and is thus most likely

to be the least natural. Conversely, instruction substitution only affects operations on integers, leaving parts of the code that don't deal with integer arithmetic unaffected. ADVobfuscator only affects string literals. While string literals are important, they exist in a minority of functions; functions with no string literals are unaffected.

Answer to RQ2: Different obfuscations may vary widely in difficulty. Difficulty is correlated with the amount of textual changes made to the code.

5.3 RQ3: How well does Learning Transfer from One Type of Obfuscation to Another?

In this experiment, we measure how well learning models trained on one obfuscation perform on binaries compiled with a different obfuscation. Since there are an infinite number of possible obfuscations, with malware authors often inventing new obfuscations as well, it is impossible for a model to be trained on every possible obfuscation. Therefore, it is important to measure a model's ability to generalize to new obfuscations unseen during training.

Methodology. We trained models on two obfuscations—control flow flattening (`fla`) and instruction substitution (`sub`)—and evaluated their performance on all other obfuscations in our dataset individually. These two obfuscations complement each other: one affects the control flow while largely leaving basic blocks intact, while instruction substitution involves modifying computations within basic blocks while leaving control flow unchanged. We choose these obfuscations because they are substantially different so we can better make general claims about the transferability of learning on obfuscations. Training a model and evaluating a model on the same obfuscation serves as a baseline (`fla` on `fla` and `sub` on `sub`). We trained the `fla` models on 4,394,527 functions and the `sub` models on 5,171,901 functions.

Results. The results are shown in Table 3. We note that in both cases, the baseline trial performance, where the train and test sets are drawn from the same obfuscations, generally perform better than other trials. Both bogus control flow and control flow flattening are control flow related obfuscations, but training on one and evaluating on the other (Table 3 line 2) does not produce results that are consistently or meaningfully better than other obfuscations. Similarly, the model trained on instruction-substitution for the most part does not generalize well to other obfuscations, though it does for retyping with DIRTY on string obfuscation, the other non-control flow obfuscation. These results illustrate the magnitude of the shift between even superficially similar obfuscations. Further, they suggest that training on one obfuscation will not necessarily transfer to other obfuscations.

Table 3. RQ3: Transferability of Learning. Results displayed in terms of percent accuracy, along with a relative percent change compared with the baselines (first row of each table section). Higher is better.

Train	Test	DIRTY				VarBERT	HexT5		
		Retyping		Renaming					
		overall		structs		variables	variables	variables	functions
fla	fla	59.3	–	30.5	–	19.8	– 22.5	– 20.1	– 18.6 –
fla	bcf	41.0	(-30.9%)	15.7	(-48.3%)	10.6	(-46.1%)	20.1	(-10.6%) 14.1 (-29.8%) 13.5 (-27.3%)
fla	sub	48.6	(-18.1%)	24.4	(-19.9%)	12.3	(-38.0%)	21.1	(-6.2%) 14.9 (-26.2%) 15.3 (-17.7%)
fla	str	56.5	(-4.8%)	28.0	(-8.0%)	10.1	(-48.9%)	10.1	(-55.0%) 12.6 (-37.4%) 11.9 (-36.2%)
sub	sub	50.5	–	25.1	–	14.6	– 30.1	– 20.3	– 19.0 –
sub	str	56.0	(+11.0%)	28.1	(+11.8%)	7.6	(-47.8%)	12.1	(-59.6%) 14.0 (-31.1%) 11.9 (-37.6%)
sub	bcf	41.1	(-18.6%)	11.4	(-54.6%)	13.4	(-8.4%)	22.3	(-25.9%) 17.4 (-14.4%) 14.3 (-24.8%)
sub	fla	51.7	(+2.4%)	22.1	(-12.0%)	8.0	(-45.0%)	10.3	(-65.6%) 12.6 (-38.0%) 16.1 (-15.4%)

Answer to RQ3: Learning is not easily transferred between obfuscations, even those which are similar.

5.4 RQ4: How Does Varying the Amount of Obfuscated Data in Training Affect Model Performance?

Table 4. RQ4: Effect of the Quantity of Obfuscated Training Data on Accuracy

Train Set	DIRTY				VarBERT	HexT5		
	Retyping		Renaming					
	overall		structs		variables	variables	variables	functions
128-none	47.1	–	18.9	–	12.9	– 14.9	– 14.4	– 12.1 –
1024-none	44.5	(-5.5%)	19.6	(+4.0%)	13.4	(+3.7%)	15.1	(+2.0%) 14.8 (+3.0%) 12.2 (+1.1%)
8192-none	45.2	(-3.9%)	18.3	(-3.2%)	13.0	(+0.8%)	15.1	(+1.7%) 15.4 (+7.0%) 12.7 (+5.1%)
128-obf	49.8	(+5.7%)	18.2	(-3.6%)	16.2	(+25.4%)	15.7	(+5.4%) 14.9 (+3.1%) 14.4 (+19.4%)
1024-obf	50.6	(+7.6%)	17.4	(-7.9%)	14.3	(+10.7%)	15.7	(+6.0%) 15.1 (+4.4%) 14.8 (+22.4%)
8192-obf	50.5	(+7.2%)	17.5	(-7.2%)	16.8	(+30.1%)	17.4	(+16.9%) 16.6 (+14.9%) 14.9 (+23.1%)

Adding small amounts of obfuscated data improves performance on obfuscated data, while adding small amounts of unobfuscated data does not. VarBERT's 1024-none accuracy is slightly greater than its 8192-none accuracy, though they round to the same value.

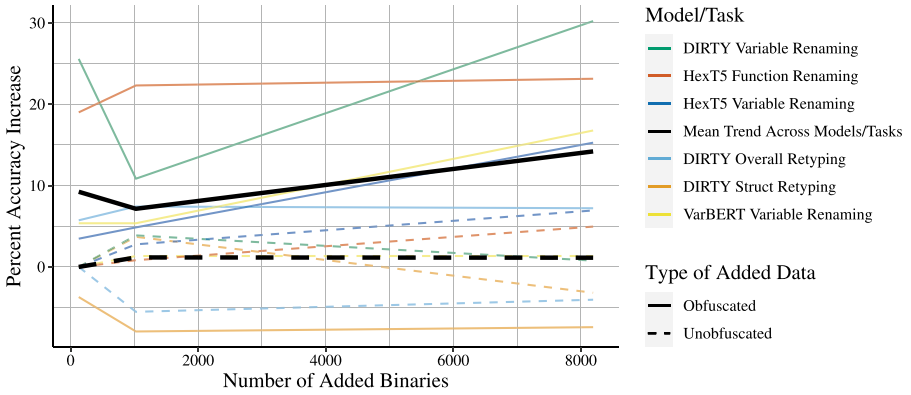

Fig. 5. Accuracy gains relative to the 128-none trials for each model/task combination (background colored lines) and the mean across all models and tasks (the black lines). Adding even a small amount of obfuscated data leads to accuracy gains, though the gains are very noisy. The results suggest steeply diminishing returns for additional data, possibly following a power law error curve, as predicted by theory [16].

Generating obfuscated training data at scale is a nontrivial task. Further, one may encounter new obfuscations for which no commercial generation tools are available. In these cases, there may be only a limited amount of labeled training data available (perhaps produced by malware analysts as they encounter new examples). This is especially important, because, as discussed in Sect. 5.3, learning does not transfer well between obfuscations.

Methodology. We designed an experiment to quantify the relationship between model performance and the number of obfuscated examples in its training set. Neural networks, including almost all state-of-the-art techniques, require large datasets to be fit well. If the amount of obfuscated training data is limited, as is likely the case for newly discovered obfuscations in practice, then training only on obfuscated training data would be insufficient. On the other hand, generating unobfuscated training samples is straightforward using the process described by Chen et al. [6] and Pal et al. [32]. Therefore, we build datasets consisting primarily of unobfuscated data with a limited amount of obfuscated data. In principle, this is very similar to the pretrain/finetune paradigm. The two steps are typically separated so that a pretrained model can be copied and used for multiple different finetuning tasks, but for our experiments, that is not relevant, so we don't separate the steps to simplify the training process.

Our training sets were created by combining a fixed set of unobfuscated data with varying numbers of obfuscated binaries. We created 3 training sets each with 128, 1024, and 8192 obfuscated binaries respectively. For each obfuscated training set, there are an equal number of binaries of each of the four obfuscation types. These had 868,968, 883,793, and 1,013,980 functions, respectively.

Our goal is to measure the relationship between amount of obfuscated data used in training and model accuracy on obfuscated code. However, training set size is a factor in model accuracy; typically, the greater the size of the training set, the higher the accuracy. To account for this, however, we also control for training set size by running separate trials with the same amount of purely unobfuscated training data. We created training sets consisting purely of unobfuscated binaries which had the same total number of decompiled executable files as the 128, 1024, and 8192 obfuscated training sets. That is, the training set of equal size to the 8192 obfuscated executable training set, contained the same unobfuscated executables as the 1024 unobfuscated train set along with 7168 more unobfuscated binaries. The sizes of these control training sets are 870,089, 885,795, and 1,030,167 functions, respectively.

We also created shared validation and test sets, with contents evenly split between no obfuscations, `fla`, `bcf`, `sub`, and `str`. We then evaluated all six models on our single test set.

Results. The results are shown in Table 4. As expected, adding a small amount of unobfuscated data (Table 4, rows 1–3) has very little impact on performance. On the other hand, adding obfuscated training data does increase performance. Most of the performance gain happens with the first 128 binaries worth of functions. Accuracy gains after this point are more muted and are very noisy. The results suggest sharply diminishing returns after adding a relatively small amount of data, as illustrated in Fig. 5. It is possible that the results follow a power law curve as well; there are diminishing returns as the amount of obfuscated data added increases. Unfortunately, the scale of the curve is such that only a few examples is insufficient to achieve a meaningful improvement in performance.

> **Answer to RQ4**: Model performance gains increase rapidly after the first obfuscated data are added, but drop off rapidly as more obfuscated data are added.

6 Threats to Validity

Internal validity is the degree to which an experiment establishes an causal relationship. A source of internal validity was that during data generation, different obfuscators can cause build failures at different stages. The automated compilation tool we used, GHCC, collects all products of compilation, even if there are failures on subsequent steps. Thus, in some cases, compiling the same project with different obfuscations may yield different collections of binaries if an obfuscation causes a compilation error. In Sect. 5.1, we account for this by sampling from other obfuscations from the same project to ensure that the training set size is consistent among the trials. Another option would have been to include only

repositories which built completely, but this would have substantially decreased the available pool of training data and biased the dataset against complex software projects that are difficult to build.

External Validity is the degree to which an experiment's results generalize beyond the population in the experiment. We expect our results to generalize well across other real-world software projects because we train on a variety of different real world software and evaluate on a randomly selected subset of them. However, our dataset is necessarily biased towards some projects which we could build automatically; it is possible that binaries from unbuilt projects might systematically present some unique challenges that may impact the results. Similarly, our dataset is biased towards open-source software, which may not be representative of all software. Malware may include other techniques to defend against reverse engineering, such as packing and encryption; however, there exist a wide variety of unpacking and decryption techniques and commercial services. From the perspective of neural variable name and type prediction, these are preprocessing steps that are orthogonal to our problem. Finally, it is possible that there are other obfuscations to which our findings do not generalize.

Construct Validity is the degree to which the experiment design supports the claims. We only claim to measure the accuracy of the models with respect to the original names and types in the source code, a common strategy in virtually all existing work in ML-based decompilation improvement. Notably, however, we cannot directly make claims that the names and types predicted are *helpful* to reverse engineers. However, we expect that the names and types in the original source code are typically more helpful than those in the decompiled code, and thus predicting more variable names and type names correctly is, in general, a positive sign.

Following Chen et al. [6], we filter away variables for which the decompiled variable name is the same as what the dataset-generation technique identifies as an original variable name. This usually happens when a variable is decompiler-generated, but also for loop indices and a few other types of variables that have predictable names. There are also other variables, about 10% of the total, that look decompiler-generated but have original variable names different from the original (e.g. `v2` vs `v5`). For consistency with prior work [6], our results include predictions on these variables. Excluding these causes numbers to shift (in either direction) a small amount, though the trends reported in each experiment remain the same.

7 Related Work

ML-Based Decompilation Improvement There is a significant body of existing work in leveraging machine learning to improve and complement decompilation. Much of the existing work focuses on predicting one particular type of information lost during compilation: variable names [26,31,32], variable types [28,43,44], function names [9,21,24], or the original syntactic structure of the original source code [2,14,18,19].

Handling Obfuscations Despite significant work in learning for decompilation, very little existing work considers the impact that obfuscations have on these techniques. The authors of SymLM [21], which predicts function names, do evaluate their work on binaries with four obfuscations (bogus control flow, control-flow-flattening, instruction substitution, and basic block splitting), similar to RQ2 in our work (Sect. 5.2). We use three of the same obfuscations, but instead of basic block splitting we use string literal obfuscation. They find that obfuscations decrease the accuracy of their tool by 2–10%, depending on the obfuscation.

Deobfuscation There is also work on undoing obfuscations. Instruction substitution can be undone using common compiler optimization passes. Other work focuses on eliminating bogus control flow constraints[42] and eliminating dead or bogus code [7]. There is also work on undoing control-flow flattening obfuscations [11]. Symbolic execution can be used in conjunction with techniques similar to those in compiler optimization to simplify functions [29,35], sometimes in conjunction with program synthesis [8]. Deobfuscation techniques generally produce code that is typically equivalent to the original but may be syntactically very different. We leave a study on how deterministic deobfuscation techniques affect neural decompilers to future work.

Neural decompilers may be used on obfuscated code in scenarios where deterministic deobfuscation is not integrated with the security researchers' toolchains or when deterministic deobfuscation fails. In addition, new obfuscations may be created at any time; while neural models only need examples, deterministic deobfuscation may require the design and implementation of new algorithms to handle them if they exploit the limitations in existing techniques.

8 Conclusion

Neural decompilation improvement models predict missing abstractions, like variable names and types, in decompiled code. Little existing work in neural decompilation improvement considers obfuscated code, despite obfuscations being widespread in practice. In this work, we quantified the impacts that four obfuscations have on three prominent decompilation improvement models.

We find that obfuscations do negatively impact the performance these models, though training on obfuscated code largely mitigates the impact of obfuscations. Unfortunately, as we show in Sects. 5.1, 5.2 and 5.3, each obfuscation we tested produced its own substantially different distribution of decompiled code. Practically, this means that if an attacker creates malware using their own secret obfuscation, a decompilation improvement model will likely perform poorly. However, there is a silver lining: as we show in Sect. 5.4, a model can see gains in performance on obfuscations when trained on only a few hundred examples, which means that models can be adapted for *known* obfuscations.

In this work, we focus on predictions at the function level, following DIRTY [6], VarBERT [32], and HexT5 [41]; it may also be interesting to examine prediction at different levels of granularity (e.g. partial function or full-program level), though function-level remains the most common approach.

Acknowledgments. This material is based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. Additionally, this material is based upon work supported by the National Science Foundation Graduate Research Fellowship Program under Grant No DGE2140739. This work used GPU nodes at Purdue Anvil through allocation CIS240492 from the Advanced Cyberinfrastructure Coordination Ecosystem: Services & Support (ACCESS) program [3], which is supported by U.S. National Science Foundation grants #2138259, #2138286, #2138307, #2137603, and #2138296. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

1. Andrivet, S.: ADVobfuscator (2020). https://github.com/andrivet/ADVobfuscator
2. Armengol-Estapé, J., Woodruff, J., Cummins, C., O'Boyle, M.F.: Slade: a portable small language model decompiler for optimized assembly. In: CGO, pp. 67–80. IEEE (2024)
3. Boerner, T.J., Deems, S., Furlani, T.R., Knuth, S.L., Towns, J.: Access: advancing innovation: NSF's advanced cyberinfrastructure coordination ecosystem: services & support. In: Practice and Experience in Advanced Research Computing 2023: Computing for the Common Good, pp. 173–176. PEARC '23, Association for Computing Machinery, New York, NY, USA (2023). https://doi.org/10.1145/3569951.3597559
4. Broder, A.Z.: On the resemblance and containment of documents. In: Proceedings. Compression and Complexity of SEQUENCES 1997 (Cat. No. 97TB100171), pp. 21–29. IEEE (1997)
5. Casalnuovo, C., Barr, E.T., Dash, S.K., Devanbu, P., Morgan, E.: A theory of dual channel constraints. In: ICSE-NIER, pp. 25–28 (2020)
6. Chen, Q., Lacomis, J., Schwartz, E.J., Le Goues, C., Neubig, G., Vasilescu, B.: Augmenting decompiler output with learned variable names and types. In: 31st USENIX Security Symposium, pp. 4327–4343 (2022)
7. Coogan, K., Lu, G., Debray, S.: Deobfuscation of virtualization-obfuscated software: a semantics-based approach. In: CCS, pp. 275–284 (2011)
8. David, R., Coniglio, L., Ceccato, M., et al.: QSynth-a program synthesis based approach for binary code deobfuscation. In: BAR 2020 Workshop (2020)
9. David, Y., Alon, U., Yahav, E.: Neural reverse engineering of stripped binaries using augmented control flow graphs **4**(OOPSLA), 1–28 (2020)
10. Devlin, J., Chang, M.W., Lee, K., Toutanova, K.: BERT: Pre-training of deep bidirectional transformers for language understanding. In: Burstein, J., Doran, C., Solorio, T. (eds.) NAACL, pp. 4171–4186. Association for Computational Linguistics, Minneapolis, Minnesota (2019)
11. Dong, W., Lin, J., Chang, R., Wang, R.: CaDeCFF: compiler-agnostic deobfuscator of control flow flattening. In: Proceedings of the 13th Asia-Pacific Symposium on Internetware (2022)
12. Dramko, L., Lacomis, J., Schwartz, E.J., Vasilescu, B., Le Goues, C.: A taxonomy of C decompiler fidelity issues. In: 33rd USENIX Security Symposium (2024)

13. Dramko, L., et al.: Dire and its data: neural decompiled variable renamings with respect to software class. ACM Trans. Softw. Eng. Methodol. **32**(2), 1–34 (2023)
14. Fu, C., et al.: Coda: an end-to-end neural program decompiler. NeurIPS **32** (2019)
15. Gu, Y., Zhang, Z., Wang, X., Liu, Z., Sun, M.: Train no evil: selective masking for task-guided pre-training. In: Webber, B., Cohn, T., He, Y., Liu, Y. (eds.) EMNLP (2020)
16. Hestness, J., et al.: Deep learning scaling is predictable, empirically. arXiv preprint arXiv:1712.00409 (2017)
17. Hindle, A., Barr, E.T., Gabel, M., Su, Z., Devanbu, P.: On the naturalness of software. Commun. ACM **59**(5), 122–131 (2016)
18. Hosseini, I., Dolan-Gavitt, B.: Beyond the c: retargetable decompilation using neural machine translation. In: NDSS (2022)
19. Hu, P., Liang, R., Chen, K.: DeGPT: optimizing decompiler output with LLM. In: NDSS (2024)
20. Hu, Z.: GHCC (2021). https://github.com/huzecong/ghcc
21. Jin, X., Pei, K., Won, J.Y., Lin, Z.: SymLM: predicting function names in stripped binaries via context-sensitive execution-aware code embeddings. In: CCS, pp. 1631–1645 (2022)
22. Junod, P., Rinaldini, J., Wehrli, J., Michielin, J.: Obfuscator-LLVM–software protection for the masses. In: SPRO, pp. 3–9. IEEE (2015)
23. Kapoor, S., Narayanan, A.: Leakage and the reproducibility crisis in machine-learning-based science. Patterns **4**(9), 100804 (2023)
24. Kim, H., Bak, J., Cho, K., Koo, H.: A transformer-based function symbol name inference model from an assembly language for binary reversing. In: Asia-CCS, pp. 951–965 (2023)
25. Kocetkov, D., et al.: The stack: 3 TB of permissively licensed source code. arXiv preprint (2022)
26. Lacomis, J., et al.: DIRE: a neural approach to decompiled identifier naming. In: ASE, pp. 628–639. IEEE (2019)
27. László, T., Kiss, Á.: Obfuscating C++ programs via control flow flattening (2009)
28. Lehmann, D., Pradel, M.: Finding the Dwarf: recovering precise types from WebAssembly binaries. In: PLDI, pp. 410–425 (2022)
29. Liang, M., Li, Z., Zeng, Q., Fang, Z.: Deobfuscation of virtualization-obfuscated code through symbolic execution and compilation optimization. In: ICICS 2017. Springer (2018)
30. Miceli-Barone, A.V., Barez, F., Cohen, S.B., Konstas, I.: The larger they are, the harder they fail: language models do not recognize identifier swaps in Python. In: ACL 2023 (2023)
31. Nitin, V., Saieva, A., Ray, B., Kaiser, G.: DIRECT: a transformer-based model for decompiled variable name recovery. In: NLP4Prog 2021, p. 48 (2021)
32. Pal, K.K., et al.: Len or index or count, anything but v1: predicting variable names in decompilation output with transfer learning. In: 2024 IEEE Symposium on Security and Privacy (SP), p. 152 (2024)
33. Quionero-Candela, J., Sugiyama, M., Schwaighofer, A., Lawrence, N.D.: Dataset Shift in Machine Learning. The MIT Press (2009)
34. Spinellis, D., Kotti, Z., Mockus, A.: A dataset for GitHub repository deduplication. In: International Conference Mining Software Repositories, pp. 523–527 (2020)
35. Tofighi-Shirazi, R., Christofi, M., Elbaz-Vincent, P., Le, T.H.: DoSE: deobfuscation based on semantic equivalence. In: SSPREW, pp. 1–12 (2018)
36. Vaswani, A., et al.: Attention is all you need. NeurIPS **30** (2017)

37. Votipka, D., Rabin, S., Micinski, K., Foster, J.S., Mazurek, M.L.: An observational investigation of reverse engineers' process and mental models. In: Extended Abstracts of the 2019 CHI Conference on Human Factors in Computing Systems, pp. 1–6 (2019)
38. Votipka, D., Rabin, S., Micinski, K., Foster, J.S., Mazurek, M.L.: An observational investigation of reverse engineers' processes. In: 29th USENIX Security Symposium, pp. 1875–1892 (2020)
39. Wang, C., Davidson, J., Hill, J., Knight, J.: Protection of software-based survivability mechanisms. In: DSN, pp. 193–202. IEEE (2001)
40. Wolf, T., et al.: Transformers: state-of-the-art natural language processing (2020).https://doi.org/10.5281/zenodo.7391177
41. Xiong, J., Chen, G., Chen, K., Gao, H., Cheng, S., Zhang, W.: Hext5: unified pre-training for stripped binary code information inference. In: ASE, pp. 774–786. IEEE (2023)
42. You, G., Kim, G., Han, S., Park, M., Cho, S.J.: Deoptfuscator: defeating advanced control-flow obfuscation using android runtime (ART). IEEE Access **10**, 61426–61440 (2022)
43. Zhang, Z., et al.: OSPREY: recovery of variable and data structure via probabilistic analysis for stripped binary. In: 2021 IEEE Symposium on Security and Privacy (SP), pp. 813–832. IEEE (2021)
44. Zhu, C., et al.: TYGR: type inference on stripped binaries using graph neural networks. In: 33rd USENIX Security Symposium, pp. 4283–4300 (2024)