Automatically Annotating Decompiled Code with Meaningful Names and Types

Jeremy Lacomis

CMU-S3D-23-103

June 2023

Software and Societal Systems Department
School of Computer Science
Carnegie Mellon University
Pittsburgh, PA 15213

Submitted in partial fulfillment of the requirements for the degree of Doctor of Philosophy in Software Engineering.

This material is based upon work supported in part by the Software Engineering Institute (LINE project 6-18-001), and National Science Foundation (awards 1815278 and 1910067).

Keywords: software reverse engineering; decompilation; decompiled code readability; decompiled code names; decompiled code types;

To Jessica, thank you for the endless support, for uprooting your entire life, and for threatening to kill me if I didn't finish my Ph.D.

Abstract

Software reverse engineering is the problem of understanding the behavior of a program without access to its source code. Since there is no source code, analysts must use the binary directly. A primary tool used by reverse engineers is the decompiler, which attempts to reverse the process of compilation. Although decompilers generate abstractions that improve code readability, the act of compilation irreversibly destroys information contained in the source code including comments, control flow abstractions, user-defined types, and identifier names, all of which are provably impossible to reconstruct.

However, software is natural: programmers tend to write the same code to perform the same tasks. While it is technically impossible to generate the original code, it is possible to train a model to automatically generate more meaningful identifier names and types. Treating code augmentation as an instance of translation allows the application of tools and techniques originally developed for natural language translation to the problem of identifier renaming and retyping.

The goal of the work presented in this thesis is to automatically augment the output of decompilers with more meaningful names and user-defined types under the hypothesis that this will decrease the cognitive burden of reasoning about their generated code. We hope that this will have several advantages: first, we believe that this will save reverse engineers valuable time that could be spent reasoning about the higher-level functionality of the code, second, we believe it will flatten the learning curve, allowing more novices to enter the field.

My core thesis statement is: Exploiting structure inherent in code, together with its naturalness, enables the application of machine translation techniques to useful transformations of decompiled code. These techniques can be used to meaningfully rename and retype variables in decompiled code.

To support this thesis I present two automated techniques for automatically renaming and retyping decompiled code. I demonstrate how these techniques are effective at making decompiled code more approachable through metrics developed as a proxy for human understanding and through a user study designed to measure the performance of the techniques in real-world applications.

Acknowledgments

This dissertation would not have been possible without the support and guidance of my wife, my advisors, my collaborators, and my friends. First, I would like to thank my friend Zak for recognizing that my talents were being wasted working at a restaurant and introducing me to the world of academia. I'd also like to thank my sister, Kristen, and my good friends Kevin, Lee, and Seth for keeping me sane while I was in graduate school. Special thanks to Seth and Zak for reading over this document and providing valuable feedback.

I would like to thank my advisors, Claire Le Goues and Bogdan Vasilescu for their financial and mental support throughout my Ph.D. I literally would not have been here at CMU if Claire did not personally come to my desk at the University of Virginia and order me to apply to the program. I would also like to thank my undergraduate advisor Westley Weimer who taught me many lessons about succeeding in graduate school that I still use to this day, and (mis)reading my many, many, *many* drafts of my statement of purpose out loud to my face.

I'd like to thank the other members of my committee, Graham Neubig and Edward Schwartz, for their feedback on this dissertation and their time. I'd also like to thank my other collaborators: Jonathan Dorn, Stephanie Forrest, Afsoon Afzal, Christopher Timperley, Alan Jaffe, Pengcheng Yin, Miltiadis Allamanis, Luke Dramko, Alex Shypula, and Qibin Chen, it has been an extreme pleasure working with all of you. I'd like to thank Chris in particular for literally saving my life. I apologize for any lasting trauma that my untimely seizure may have caused you.

Huge thanks to everyone who has ever sat through my practice talks, including past and present members of squaresLab and STRUDEL: Rijnard van Tonder, Zack Coker, Deborah Katz, Mauricio Soto, Cody Kineer, Leo Chen, Trenton Tabor, Milda Zizyte, Sophia Kolak, Daniel Ramos, Kush Jain, Aidan Yang, Tobias Dürschmid, Yiwei Lyu, Zhen Yu Ding, Courtney Miller, Hongbo Fang, David Widder, Marat Valiev, Daye Nam, Huilian Qiu, Alex Nolte, Fabio Calefato, Hemank Lamba, and anyone else I might have missed. The fact that the list is so long demonstrates just how supportive the community at Carnegie Mellon are.

Last, but by no means least, I would like to thank my wife, Jessica, for encouraging me to go back to school and threatening me with literal death if I did not finish my Ph.D. I don't think it would have been possible to complete not just this, but *any* program without your support and encouragement.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

Software reverse engineering is the problem of understanding the behavior of a program without having access to its source code. Reverse engineering is often used to predict the behavior of malware [48, 177, 178], discover vulnerabilities [146, 159, 177], and patch bugs in legacy software [146, 159]. For malware and malicious botnets, reverse engineering enables understanding and response, and helps identify and patch infection vectors. For example, by reverse engineering the Torbig botnet (which caused 180K infections and collected 70GB of credit card/bank account information), responders were able to predict future domain names that bots would contact, and redirect the bots to servers under the responders' control [155]. Reverse engineering can also help identify who created a piece of malware, as was done for the Uroburos rootkit [1] (which captured files and network traffic while propagating over networks of companies and public authorities), and estimate the extent of infection [140].

Since there is no source code, analysis is often performed at the binary level. This can be challenging: compilers (and instruction sets in general) optimize for execution speed or binary size, not readability. One of the main tools reverse engineers use to inspect programs is the *disassembler*—a tool that translates a binary to low-level assembly code. Disassemblers range from simple tools like GNU Binutils' objdump [18], to more advanced tools like IDA [80], which can be used interactively and have more sophisticated features. However, reasoning at the assembly level requires considerable cognitive effort even with these advanced features [146, 159, 178]. More recently, reverse engineers are employing *decompilers* such as Hex-Rays [71] and Ghidra [61], which reverse compilation by translating the output of disassemblers into code that resembles high-level languages such as C. These state-of-the-art tools are able to use program analysis and heuristics to reconstruct structural information about a program's variables, types, functions, and control flow structure.

The Problem. Although decompilers generate abstractions that improve code readability and are widely used by reverse engineers in practice, they rarely fully reconstruct the original developer-written code [145], since the process of compilation irrevocably destroys some information. This means that useful pieces of information, such as comments, identifier names, and user-defined types, all of which are known to meaningfully contribute to program comprehension [59, 92], are typically absent from decompiler output.

Using a decompiler still has a high cognitive burden and a steep learning curve. A real-world

```c
 1  data_unset *array_extract_element_klen(array * const a, const char *key, const uint32_t klen
        ) {
 2      const int32_t ipos = array_get_index(a, key, klen);
 3      if (ipos < 0) return NULL;
 4
 5      /* remove entry from a->sorted: move everything after pos one step left */
 6      data_unset * const entry = a->sorted[ipos];
 7      const uint32_t last_ndx = --a->used;
 8      if (last_ndx != (uint32_t)ipos) {
 9          data_unset ** const d = a->sorted + ipos;
10          memmove(d, d+1, (last_ndx - (uint32_t)ipos) * sizeof(*d));
11      }
12
13      if (entry != a->data[last_ndx]) {
14          /* walk a->data[] to find data ptr */
15          /* (not checking (ndx <= last_ndx) since entry must be in a->data[]) */
16          uint32_t ndx = 0;
17          while (entry != a->data[ndx]) ++ndx;
18          a->data[ndx] = a->data[last_ndx]; /* swap with last element */
19      }
20      a->data[last_ndx] = NULL;
21      return entry;
22  }
```

(a) Original source code.

```c
 1  __int64 __fastcall array_extract_element_klen(__int64 a1, __int64 a2, unsigned int a3)
 2  {
 3    unsigned int i; // [rsp+24h] [rbp-1Ch]
 4    int index; // [rsp+28h] [rbp-18h]
 5    unsigned int v6; // [rsp+2Ch] [rbp-14h]
 6    __int64 v7; // [rsp+30h] [rbp-10h]
 7
 8    index = array_get_index(a1, a2, a3);
 9    if ( index < 0 )
10      return 0LL;
11    v7 = *(_QWORD *)(8LL * index + *(_QWORD *)(a1 + 8));
12    v6 = --*(_DWORD *)(a1 + 16);
13    if ( v6 != index )
14      memmove(
15        (void *)(8LL * index + *(_QWORD *)(a1 + 8)),
16        (const void *)(8LL * index + *(_QWORD *)(a1 + 8) + 8),
17        8LL * (v6 - index));
18    if ( v7 != *(_QWORD *)(8LL * v6 + *(_QWORD *)a1) )
19    {
20      for ( i = 0; v7 != *(_QWORD *)(8LL * i + *(_QWORD *)a1); ++i )
21        ;
22      *(_QWORD *)(*(_QWORD *)a1 + 8LL * i) = *(_QWORD *)(*(_QWORD *)a1 + 8LL * v6);
23    }
24    *(_QWORD *)(8LL * v6 + *(_QWORD *)a1) = 0LL;
25    return v7;
26  }
```

(b) Hex-Rays decompilation.

Figure 1.1: Typical output of the Hex-Rays decompiler. Line 7 in (a) corresponds to line 12 in (b). Notice how the meaningful `--a->used` has been turned into the semantically equivalent but less idiomatic `--*(_DWORD *)(a1 + 16)`.

example of typical output of the Hex-Rays decompiler is shown in Figure 1.1. Even a seasoned programmer would have difficulty understanding the meaning of `v6 = --*(_DWORD *)(a1 + 16);` on line 12 in Figure 1.1b. It takes practice to understand that `a1` is a pointer to a structure passed into a function, and this operation decrements one of its fields. In the original code, shown in Figure 1.1a, this line has the much more natural form `const uint32_t last_ndx = --a->used;`. Recognizing patterns like these is large part of the job: reverse engineers spend much of their time manually changing the output of decompilers to add more meaningful names and types allowing them to reason about the code [163]. It would be useful if the decompiler itself could recognize these cases, but in his Ph.D. thesis, Van Emmerik observed: "Certainly, the original names are not recoverable, assuming the debug symbols are not present. [...] Unless some advanced artificial intelligence techniques become feasible, these aspects will never be satisfactorily generated without manual intervention" [159].

Goal of this Work. The goal of the work presented in this thesis is to automatically augment the output of decompilers with more meaningful variable names and types under the hypothesis that this will decrease the cognitive burden of reasoning about code. I hope that this will have several advantages: first, I believe that it will save experienced reverse engineers valuable time that could be spent reasoning about the higher-level functionality of the code, second, I believe it will flatten the learning curve, lowering the barrier of entry so that the required knowledge is closer to what is needed for software development.

Overall, this thesis identifies difficulties experienced by users of decompilers and proposes approaches to automatically augment their output. It also identifies the challenges of automatically evaluating the effectiveness of these models and motivates future research. I hope that the techniques proposed here can lower the difficulty of interacting with decompilers to allow current reverse engineers to use their time more productively and also enable more computer scientists to become reverse engineers.

Specific Challenges. While it is technically impossible to exactly generate the same code originally written by a developer, it *is* possible to generate useful information about the original code lost during compilation. Intuitively, humans tend to write the same code to perform the same tasks (e.g., there are very few ways to compute the distance between two points in 2D space). This *naturalness* property is well-studied [8, 43, 72], and can be exploited to generate names and types that humans used in similar code, *even when the information is not included in the binary*.

The task of transforming code without meaningful variable names and types to code with meaningful names and types can be viewed as an instance of *translation* between code without meaningful names and types to code with them. Treating the problem as an instance of translation allows us to adapt machine learning algorithms from the field of natural language processing (NLP). NLP is an extremely mature field, and the idea of automatic natural language translation is older than the computer itself [79]. Supervised machine-learning translation systems need input/output pairs of aligned sentences in two languages. For natural language these pairs can be extracted from sources such as bilingual newspapers, for programming languages these can be automatically generated directly from open-source code on GitHub. I discuss the challenges associated with generating the training corpus and creating the system more in Chapter 3.

Renaming variables is a conceptually simple task: excluding edge cases such as keywords or aliasing, developers are free to choose any name they want for a variable. While being able to choose an arbitrary name for a variable makes the task much more tractable, the open vocabulary makes it much more difficult to predict the "correct" one. Comparatively, types should be much easier to predict correctly since they come with natural constraints on their uses. For example, a `char` type can almost never be replaced with a `float` type.[1] By conditioning predictions on the size of the data represented by a specific variable, the accuracy should increase. While this is not always the case, memory layout does improve the performance of retyping. I discuss the implementation of this system and challenges encountered further in Chapter 4.

Effectiveness. I measured the effectiveness of these techniques in two different ways. First, under the hypothesis that the original programmer chose meaningful names and types, I measured what percentage of the outputs generated the techniques that exactly match what the original developer used. Second, I performed a human study designed to measure the difficulty of reasoning about code treated with our techniques. I discuss specifics of both further in Section 1.2.

1.1 Thesis Statement

Exploiting structure inherent in code, together with its naturalness, enables the application of machine translation techniques to useful transformations of decompiled code. These techniques can be used to meaningfully rename and retype variables in decompiled code.

1.2 Scope and Evaluation Metrics

The techniques in this dissertation are intended to be applied to the output of real-world decompilers with the ultimate goal of easing the cognitive burden on end-users. Specifically, I address the inability of existing decompilers to algorithmically choose variable names and types that communicate their purpose to a human. At least one recent study observing reverse engineers has called for reverse engineering tools that minimize the difficulty of renaming variables [163]. While the problem of assigning meaningful names to identifiers is not unique to decompiled code, our approaches target it specifically. The primary reasons are that (a) it is particularly important that decompiled code is understandable since reverse engineering is primarily a *reading* task rather than a *writing* task and (b) decompiled code is automatically generated from a set of rules, which put it in a canonical form that should be easier to reason about.

We use two techniques to measure how meaningful the generated types and names are. First, since we are trying to train a supervised machine learning model, we need a fitness function that can be automatically run very quickly. For this, I directly compared the output of the model to the name or type chosen by the original programmer and considered a prediction to be correct if they exactly matched. I view this metric as reasonable because when a developer writes code they

[1]The use of *almost never* here is deliberate. According to the C standard this is implementation-defined, but for the purposes of this dissertation I will be assume type sizes for a standard Linux kernel targeting x86-64.

tend to use types and names that convey the intended use of variables. This has the advantage of being very easy to implement and run.

However, this metric does have some shortcomings. First, it misses cases where there are many semantically equivalent names, e.g., `len` and `length`. Second, it cannot measure the consequences of a model predicting a name or type of a variable is exactly the *most misleading*. For example, if a variable is supposed to be named `speed`, but is instead named `size`, this might lead a reverse engineer in the wrong direction.

Alternative metrics originally designed for benchmarking natural language translation techniques, such as BLEU [133], make assumptions about the structure of the natural language that are inapplicable to code. For example, BLEU is designed to work with natural language, which has a relatively high tolerance for ambiguity. Comparatively code is specifically designed to have minimal ambiguity, and changing the order of tokens can have a massive effect on its meaning. Other techniques for automatically evaluating the output of code synthesis models have been developed [31, 139], but the actual, real-world relationship between a high score on these metrics and users' ability to use the generated code is still unknown.

To evaluate the real-world performance, I performed a human study based on prior works that measured the impact of automatically-generated code on human understanding [55], and measured the output of usability-optimized decompilers [178]. The study presented professional and amateur reverse engineers with snippets of code from the Hex-Rays decompiler with and without access to our techniques and compared their performance on tasks that required reasoning about the code. This study collected both quantitative (i.e., correctness and timing) and qualitative (i.e., subjective opinions about the examples) data. In this study I observed that while these augmentations are useful for solving many problems, there are others where engineers' performance decreases when they are inaccurate. I discuss this study further in Chapter 5.

1.3 Contributions

This thesis contains both qualitative and quantitative studies, where I conceptualized and constructed actual tools for annotating decompiled code with meaningful variable names and types. In addition, I conducted a human study asking students, professors, and real-world professional reverse engineers to reason about the modified code.

This thesis contributes in the following ways:

1. It demonstrates that viewing the problem of postprocessing decompiler output as an instance of translation allows the effective application of techniques from the domain of natural language translation.
2. It introduces a novel technique for generating input/output examples suitable for training natural language translation models on decompiled code.
3. It presents a technique that automatically renames variables in decompiled code by leveraging the naturalness of code.
4. It presents a black-box technique for postprocessing decompiled code with a Transformer-based neural network to recommend user-created variable types.
5. It produces two datasets suitable for training and evaluating models of decompiled code.

6. It identifies specific challenges with automated fitness functions for measuring the effectiveness of these techniques.
7. It provides a human study protocol for measuring the impact of decompiler augmentations on user performance.
8. It describes the results of a human study testing the effectiveness of both techniques.

Parts of this thesis have been published in peer reviewed venues:

- **DIRE: A Neural Approach to Decompiled Identifier Naming**, Jeremy Lacomis, Pengcheng Yin, Edward J. Schwartz, Miltiadis Allamanis, Claire Le Goues, Graham Neubig, and Bogdan Vasilescu, in International Conference on Automated Software Engineering (ASE), 2019 [88].
- **DIRE and its Data: Neural Decompiled Variable Renamings with respect to Software Class**, Luke Dramko, Jeremy Lacomis, Pengcheng Yin, Edward J. Schwartz, Miltiadis Allamanis, Graham Neubig, Bogdan Vasilescu, and Claire Le Goues, in Transactions on Software Engineering and Methodology, 2022 [47].
- **Augmenting Decompiler Output with Learned Variable Names and Types**, Qibin Chen, Jeremy Lacomis, Edward J. Schwartz, Claire Le Goues, Graham Neubig, and Bogdan Vasilescu, in USENIX Security Symposium, 2022 [30].

1.4 Outline

The structure of this document is as follows. Chapter 2 presents an overview of the literature and a the background knowledge needed to follow the rest of the document. In Chapter 3 I describe the design and implementation of an automatic technique for renaming variables in decompiled code. In Chapter 4 I present a similar automated technique for annotating decompiled code with meaningful user-generated types. In Chapter 5 I discuss an empirical human study on the effectiveness of renamings and retypings. Finally, in Chapter 6 I conclude the thesis.

Chapter 2

Background and Review of Literature

This chapter will summarize the background information and related work for this thesis.

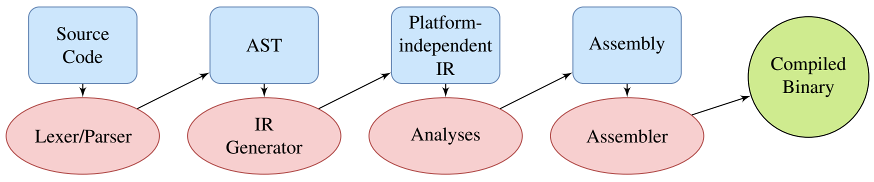
2.1 Background

2.1.1 Software Reverse Engineering

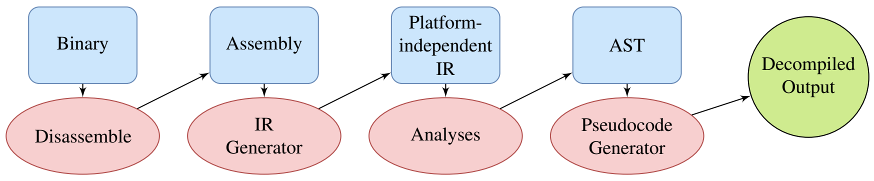
Software reverse engineering is the task of understanding the behavior of a program without use of its source code. Reverse engineering is commonly used to understand malware [48, 177, 178], and discover vulnerabilities in existing software [146, 159, 177], and legacy software [146, 159]. Reverse engineering malware can be used to prevent damage, for example, by reverse-engineering the Torbig botnet, which stole 70GB of credit card/bank account information, responders were able to predict future domain names that bots would contact, and redirect the bots to controlled servers [155]. Reverse engineering can also identify the provenance of a piece of malware [1], or measure the extent of infection [140].

One of the most important tools used to investigate binary programs is the *disassembler*. Disassemblers translate binary code to a listing of assembly instructions run on a CPU. Disassemblers themselves have to deal with many challenges. One of the biggest is identifying which sections of a binary are code and which sections are data. Unlike Java bytecode, x86 allows code and data to be mixed and uses variable length instructions that do not need to be aligned. In practice, disassembly is usually reliant on patterns commonly used by compilers, but in general the distinction between data and code on x86 is undecidable. That is, bytes are code only if they are reachable when the program is run. The next issue in disassembly is identifying boundaries between functions. Typically these are identified with explicit `call` and `ret` instructions, but advanced compiler optimizations are allowed to ignore Application Binary Interface (ABI) conventions and generate code that performs the same function as the input.

Disassemblers range from the relatively simple (e.g., `objdump` [18], which prints the instructions to the standard output), to advanced (e.g., IDA [80] and Ghidra [61], which can be used interactively). IDA and Ghidra offer quite advanced features, including control-flow graphs, call graphs, displaying entropy to identify encrypted sections of binaries, integration with debuggers, scripting, and more. However, even with these advanced features it is still very difficult to reason



(a) The stages of a compiler.



(b) The stages of a decompiler.

Figure 2.1: Stages of a typical compiler (a) and decompiler (b). The ovals represent stages, the rectangles represent the input to each stage, and the circles represent output.

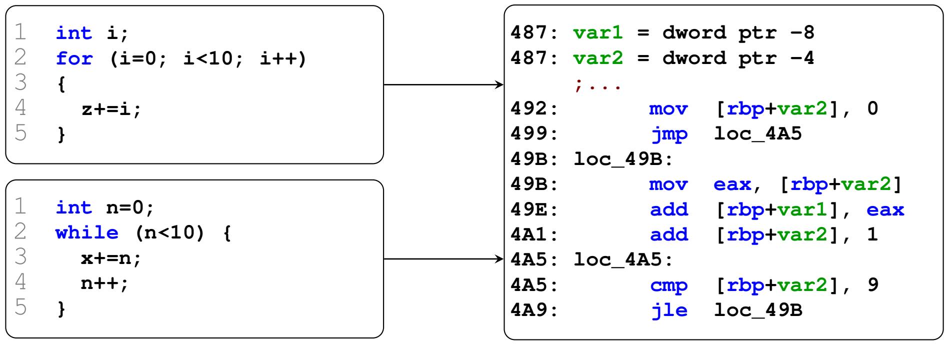
about disassembled code directly [146, 159, 178].

2.1.2 Decompilation

Decompilers are designed to alleviate the problem of reasoning about assembly code. A decompiler attempts to fully reverse the compilation process; taking the output of a disassembler and generating code in a high-level language. The decompiler is not much younger than the compiler: while the first compilers appeared in the 1950s, the first decompilers were used in the 1960s to assist developers when porting programs between machines [67].

At a high level, a compiler generates binaries from source using a pipeline of processing stages, and decompilers try to reverse this pipeline using various techniques [38, 83]. A comparison of the two techniques is shown in Figure 2.1. Typically, a binary is first passed through a platform-specific *disassembler*. Next, assembly code is typically *lifted* to a platform-independent intermediate representation (IR) using a binary-to-IR lifter. The next stage is the heart of the decompiler, and is where a number of program analyses are used to recover variables, types, functions and control flow abstractions, which are ultimately combined to reconstruct an abstract syntax tree (AST) corresponding to an idiomatic program. Finally, a code generator converts the AST to the decompiled output. The separation of the compiler and decompiler into stages allows them to be developed independently and the use of an IR allows cross-platform compatibility. Hex-Rays [71] and Ghidra [61] are two examples whose output resembles ANSI C. These decompilers use advanced program analyses to reconstruct the variables, types, functions, and control flow structure used by the original programmer.

Although this output is more understandable than assembly, decompilation is imperfect. The act of compilation almost always discards source-level information that cannot be recomputed such as comments, variable names, types, and structure to minimize binary size and execution time. Although existing studies have shown that the quality of this type of information is essential for program comprehension [59, 92], decompilers are rarely able to do more than generate



```c
int i;
for (i=0; i<10; i++)
{
  z+=i;
}
```

```c
int n=0;
while (n<10) {
  x+=n;
  n++;
}
```

```
487: var1 = dword ptr -8
487: var2 = dword ptr -4
     ;...
492:       mov  [rbp+var2], 0
499:       jmp  loc_4A5
49B: loc_49B:
49B:       mov  eax, [rbp+var2]
49E:       add  [rbp+var1], eax
4A1:       add  [rbp+var2], 1
4A5: loc_4A5:
4A5:       cmp  [rbp+var2], 9
4A9:       jle  loc_49B
```

Figure 2.2: Two different C loops that compile to the same assembly code.

placeholders (e.g., v1 and v2) for variable names.

For example, the lexing/parsing stage of the compiler does not propagate code comments to the AST. Similarly, converting from the AST to IR can lose additional information. This loss of information allows multiple distinct source code programs to compile to the same assembly code. For example, the two loops in Figure 2.2 are reduced to the same assembly instructions. The decompiler cannot know which source code was the original, but it does try to generate code that is *idiomatic*, using heuristics to increase code readability. For example, high-level control flow structures such as `while` loops are preferred over `goto` statements.

The choice of which code to generate is largely heuristic, but can be informed by the inclusion of DWARF debugging information [49]. This debugging information, which can optionally be generated at compile-time, greatly assists the decompiler by identifying function offsets, types of variables, identifier names, and user-defined structures and unions.

Academic decompiler research has a long history. The earliest decompilers in the 1960s and 1970s were primarily focused on the translation of code between different machines [13, 15, 67, 143] and documentation [74]. Cifuentes gives a comprehensive history of the early history of decompilers between 1960 and 1994 in her Ph.D. dissertation [38]. The decompiler she presents as a large portion of her dissertation, `dcc`, is considered to be the first to represent decompiled code using an Intermediate Representation (IR), which allows for portability (additional architectures can be supported with new machine code to IR lifters), and allows the simple addition of new analyses to output better high-level code.

Identifier Names

Identifier naming is well-known to contribute to understanding, but decompilers typically only make a cursory attempt to assign meaningful names to identifiers. This task is quite difficult since, with very few exceptions, developers are free to assign any name they like to an identifier. There are some exceptions: IDA, for example, uses a technique called Fast Library Identification and Recognition Technology (FLIRT)[1] to identify standard library functions generated by supported compilers and assign names (and types) based on their known signatures. However in other cases it falls back to a standard technique of generating placeholder dummy names such

[1]`https://hex-rays.com/products/ida/tech/flirt/`

as `v1` and `a4` variables and arguments, and some simple heuristics such as using `i` and `j` for loop guards. Ghidra[2] has a slightly more advanced heuristic: instead of `v` and `a`, it also assigns arguments and variables a prefix that depends on their computed type. Beyond this, assigning identifiers meaningful names typically requires the use of a machine learning technique, I discuss these further in Section 2.2.1.

Types

Much more research effort has been focused on the recovery of types in decompilers. This is understandable: even more than names, the type of a variable provides information about the semantics of a program. For example, it is more important to know if a variable represents a floating-point number or a boolean than a reasonable name for it. Caballero et al. compiled a survey of work on type recovery that is quite thorough [22].

It is important to note that there are different kinds of type recovery. The first, which I will refer to as *syntactic* recovery, attempts to recover the memory layout of a variable, such as `struct {float; float}`, but not the name of a structure or its fields. The other, which I will call *semantic* recovery, also attempts to recover these names. The majority of research focuses on syntactic recovery, which can be done statically [29, 80, 124], dynamically [152, 154, 183], or both [23, 96]. The techniques used in these analyses vary greatly. Value-based inference techniques directly examine the contents of memory [40, 154], while flow-based type inference constrain types based on the operations performed on variables (e.g., values returned from calls to functions in known libraries can be assigned a type) [64, 137].

Semantic recovery is a more recent development, with REWARDS [109] being one of the first in 2010. REWARDS dynamically runs a binary, tagging memory locations accessed by a program and propagating information until it hits a "type sink" (e.g., a call to a standard library with known types). While this is useful for reconstructing structures composed of primitive types, it is limited in its generation of types to those that reach predefined type sinks. Other semantic recovery typically relies on machine learning, which I discuss further in Section 2.2.2.

2.1.3 Machine Learning

Neural Networks. A Neural Network (NN) is a collection of nodes called *neurons* that is used to perform some computation. An example of a feed-forward neural network is shown in Figure 2.3. The first layer is called the *input layer*, which takes an input $\mathbf{x}_i \in \mathbb{R}^n$; in Figure 2.3, $\mathbf{x}_i \in \mathbb{R}^3$, so there are three nodes in the input layer. Similarly, the *output layer* is a vector of real numbers $\mathbf{y}_j \in \mathbb{R}^n$. In between the input and output layers are *hidden layers*, where each has some fixed number of nodes. Each neuron is of the form:

$$f(\mathbf{W}\mathbf{x} + \mathbf{b}) \tag{2.1}$$

Where $\mathbf{x}$ is the vector of inputs from the previous layer, $\mathbf{W}$ is a vector of *weights* for each of the inputs, $\mathbf{b}$ is a *bias* to apply to each neuron, and f is a nonlinear and differentiable *activation function*. The values of $\mathbf{W}$ and $\mathbf{b}$ are adjusted in a process called *training*. Supervised training

[2]`https://ghidra-sre.org`

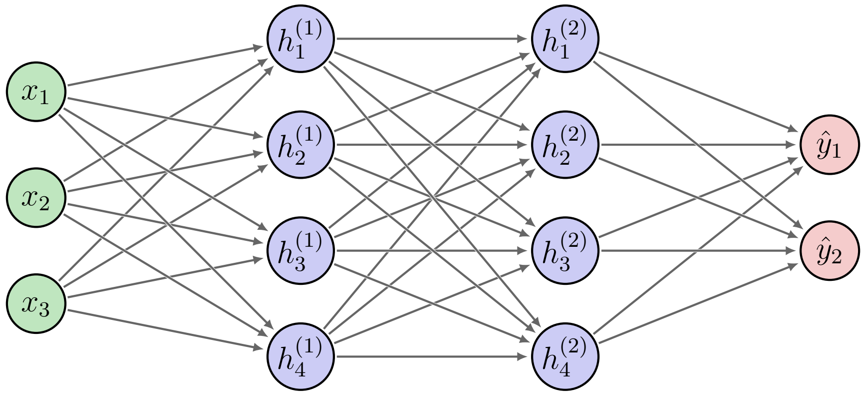


Figure 2.3: An illustration of a 3-4-4-2 feed-forward neural network architecture. The inputs are on the left represented by x_n, hidden layers are in the middle represented as $h_n^{(l)}$, and the outputs are on the right, represented as $\hat{y}_n$.

uses input/output examples and adjusts the weights to minimize the difference between the actual output and the expected output.

Encoder-Decoders. Our task consists of generating variable types and names as output given decompiled code as input. Unlike a traditional classification problem with a fixed number of classes, both our input and output are sequences of variable length: input code can have arbitrarily many variables, and each requires a type and name prediction. This prohibits the direct use of traditional feed-forward neural network architectures, which have a fixed size input and output.

Encoder-Decoder architectures [36], commonly used for sequence-to-sequence transformations, were designed to address the problem of variable-sized inputs and/or outputs. More specifically, the *encoder* takes the variable-length input and encodes it in an abstract numerical representation. This encoding is then passed to the *decoder*, which converts it into a variable-length output sequence. This architecture, further enhanced through the *attention* mechanism [12], has been shown to be effective in many tasks such as machine translation, text summarization [126], and image captioning [176]. There are several ways to implement an encoder-decoder. The approaches I use rely on three advances in statistical models for representing source code: recurrent neural networks (RNNs), gated-graph neural networks (GGNNs), and Transformers.

Recurrent Neural Networks. RNNs are networks where connections between nodes form a sequence [142], one of them is shown in Figure 2.4. In this diagram, inputs are at the bottom, outputs are at the top, and the network is in the middle. Note that the output of the network is connected back to itself. At each timestep t, a token is vectorized and input to the network, together with a previous state. RNNs are typically used to process sequences of inputs by reading in one element at a time, making them well-suited to sequences, such as source code tokens. In this thesis, I use long short-term memory (LSTM) models [73], a variant of RNNs widely used in text processing. An LSTM has a specific structure that enables it to maintain both long-term and short-term information. I will discuss the details of LSTMs more in Chapter 3.

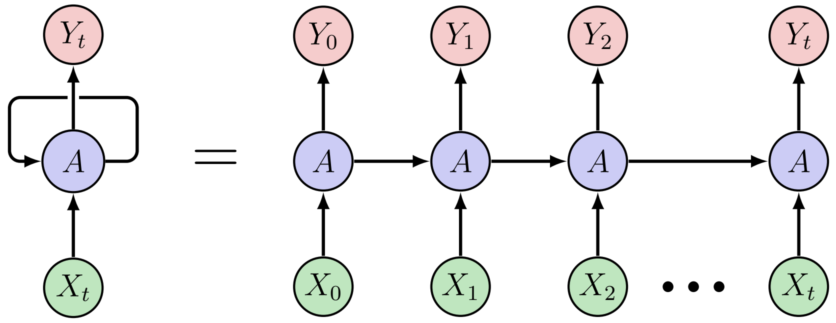


Figure 2.4: An illustration of a Recurrent Neural Network (RNN). The left-hand side of the diagram is an overview of the network, where the inputs and outputs (bottom and top, respectively) are represented as a vector. The blue circle in the is an arbitrary network, but with a connection back to its input. The unrolled diagram on the right represents a sequence of inputs at distinct timesteps t.

Gated-Graph Neural Networks. While LSTMs are useful for modeling sequences, they do not capture additional structural information. Within the decompilation task, structured information provided by the AST is a natural information source about choice of variable names. For this purpose, I also employ structural encoding of the code using GGNNs, a class of neural models that map *graphs* to outputs [106, 144]. At a high level, GGNNs are neural networks over directed graphs. Initially, each vertex is associated with a learned or computed hidden state containing information about the vertex. GGNNs compute representations for each node based on the initial node information and the graph structure. These will also be discussed further in Chapter 3.

Transformers. Transformer-based models [19, 54, 136, 180], build on the original Transformer architecture [161], and have been shown to outperform LSTMs. They are considered to be the state-of-the-art for a wide range of natural language processing tasks, including machine translation [19], question answering and abstractive summarization [44, 97], and dialog systems [2]. Transformer-based models have also been shown to outperform convolutional neural networks (CNNs) such as ResNet [69] on image recognition tasks [46]. Transformers will be discussed more in Chapter 4.

2.2 Related Work

2.2.1 Variable Names

There is existing work on predicting variable names directly from source code [10]. The most closely related examples of this type of prediction to this thesis target minimized or obfuscated JavaScript [17, 138, 160]. Although this problem is similar to the problem of choosing variable names in decompiled code, it is much easier to generate training data for obfuscated JavaScript than for decompiler output.

Other work targets renaming variables and generating debug information directly from executables [41, 68]. The generated debug information can be passed to the decompiler, which can use it to fill in information such as variable names. However, these approaches make predictions based only on the contents of the binary. In contrast, this thesis integrates with the decompiler, leveraging its advanced analyses to make more accurate predictions.

2.2.2 Type Recovery

Program analysis-based work on type recovery for decompilation such as REWARDS [109], TIE [96], and CATI [29] compute constraints to generate types. These do not use types mined from the real world, and are are either limited to only predicting the syntactic type (i.e., the base types of fields of a `struct`, but not their names), or predicting one of a small set of hand-written types. Additionally these techniques predict information directly from stripped binaries without the use of a decompiler. Other approaches work directly on assembly [56, 83, 84], and learn code structure generation instead of aiming to recover developer-specified variable types or names.

There are projects that use machine learning to predict types, but do not target decompilation. For example, DeepTyper [70] learns type inference for JavaScript while OptTyper [132], LambdaNet [172], and R-GNN$_{\text{NS-CTX}}$ [179] target TypeScript. Note that these techniques target dynamically-typed languages where type information is optional. This makes the generation of training data simple, since a parallel corpus can be generated by simply deleting type annotations from typed code. Generating a training dataset for decompiled code is much more challenging; this is described more in Chapter 3.

TypeMiner [119] and Escalada et al. [52], which recover types from binaries compiled from C, are the most directly related to the work in Chapter 4. However, both of these approaches are limited to the prediction of a small number of types (17 and 10, respectively). Both methods use relatively simple machine learning classifiers trained on small datasets. Other projects related to type recovery for decompilation are REWARDS [109], TIE [96], Retypd [130], and Osprey [185]. Unlike my approaches, they use program analyses to compute constraints on types. Additionally, they are either limited to only predicting the syntactic type (TIE, Retypd, Osprey), or only predicting one of a small set of hand-written types (150 for REWARDS).

2.2.3 Statistical Modeling for Source Code

A wide variety of statistical models for representing source code have been proposed based on the *naturalness* of software [43, 72]. This key property states that source code is highly repetitive given context, and is therefore predictable. Statistical models capture the implicit knowledge hidden within code, and apply it to build new software development tools and program analyses, e.g., for code completion, documentation generation, and automated type annotation [8].

Predicting variable names is no exception. Work has shown that statistical models trained on source code corpora can predict descriptive names for variables in a previously-unseen program, given the contextual features of the code the variable is used in. These naming models can help to distill coding conventions [5] or analyze obfuscated code [138, 160]. Several classes of statistical models have been used for renaming, including n-grams [5, 160], conditional random fields (CRFs) [138], and deep learning models [6, 9, 11].

Software Development. Statistical modeling techniques have been used for many aspects of software development. Machine learning has not only been used to automatically generate code comments [39, 50, 63, 75, 108, 116, 171, 194], and code completion [57, 99, 110, 128], but to generate source code itself [14, 20, 32, 33, 65, 131, 150, 156, 169, 170, 181, 184]. Code generation has spanned from using generative Transformers to create syntactically correct code in multiple programming languages [156], to referencing documentation for generation [193], to code optimization [118]. Code summarization [7, 94, 95, 99, 165] and method name generation [58, 129] have also been targets of techniques originally used to generate natural language.

Software Design. Statistical techniques can also aid software designers. A key issue when adding a design to an existing project is the detection of existing design patterns, which can be aided with machine learning [28, 157]. One technique works by extracting subtrees from an abstract semantic graph to generate a feature map that a trained model uses to identify design patterns. Other techniques have been used for GUI modeling [27], including an ML-based search engine to detect Android UI designs [123].

Testing. Testing is a key problem in software engineering, and machine learning has been used extensively for tasks such as bug detection [16, 42, 45, 98, 104, 105, 111, 115, 164, 168, 174, 191, 192], and localization [26, 34, 53, 66, 76, 77, 78, 89, 175, 182, 186, 188, 190]. These techniques can be general, for example using ML to predict aging-related cross-project bugs [164], or quite specific, for example Textout [168] which was developed specifically to detect bugs related to text layout in mobile applications. Bug localization is another problem that is particularly amenable to ML: CNN-FL [186] localizes suspicious statements in source code by training with test cases and outputting suspiciousness scores. Test case generation is another frequent target of ML [86, 93, 107, 112, 113, 114, 141, 189]. DeepSQLi [112], for example, produces test cases for detecting SQL injection attacks using sequence to sequence models that capture the semantic knowledge of attacks and transforms user inputs into new test cases.

Requirements Engineering. Machine learning can be used to assist requirements engineers. ML-based natural language processing has been used for extracting requirements directly from natural-language documentation [3, 100, 101, 102, 103, 135, 149]. Convolutional Neural Networks (CNNs) have been used to predict which verification techniques are best to use for existing requirements [173]. Machine learning has also been used to resolve ambiguity in natural language to improve the performance of information retrieval methods used to automatically generate requirements traces [167].

Chapter 3

A Neural Approach to Decompiled Identifier Naming

In this chapter, I will present the Decompiled Identifier Renaming Engine (DIRE), a novel neural network approach for assigning meaningful names to variables in decompiled code. To build DIRE, I relied on two key insights. First, software is *natural*, i.e., programmers tend to write similar code and use the same variable names in similar contexts [43, 72]. Therefore, because of this repetitiveness, if given a large enough training corpus one can *learn* appropriate variable names for a particular context.

Prior approaches exist to predict natural variable names from both source code [10, 17, 138, 160] and compiled executables [68, 81]. However, approaches to predict variable names either operate directly on the binary semantics [41, 68], or on the lexical output of the decompiler [81]. The former ignores the rich abstractions that modern decompilers are able to recover. The latter is an improvement, but a lexical program representation is by its very nature sequential, and lacks rich structural information that could be used to improve predictions. In contrast, DIRE uses the extended context provided by the decompiler's internal abstract syntax tree (AST) representation of the decompiled binary, which encodes additional *structural* information.

To train such models, one needs training data that specifies what names are natural in what contexts. The second key insight is that unlike other domains, where creating training data often requires manual curation (e.g., machine translation [85]), it is possible to *automatically* generate large amounts of training data for identifier name prediction, To that end, we mine open-source C code from GITHUB, compile it *with debugging information* such that the binaries preserve the original names, and decompile those binaries so that the output contains the original names. We then strip the debug symbols, decompile the binary again, and identify the alignment between the identifiers in the two versions of the decompiler outputs. While this is conceptually straightforward, the two outputs are not simply α-renamings (i.e., identical except for the choice of variable names), making the process of calculating these alignments far from trivial. In prior work, we identified alignments based entirely on heuristics [81]. In contrast, here we observe that the set of instruction addresses that access each variable uniquely identifies that variable, and this can be used to generate accurate alignments, discussed further in Section 3.2.

With these insights we train and evaluate Decompiled Identifier Renaming Engine on a large dataset of C code mined from GITHUB, showing that we can predict variable names identical to

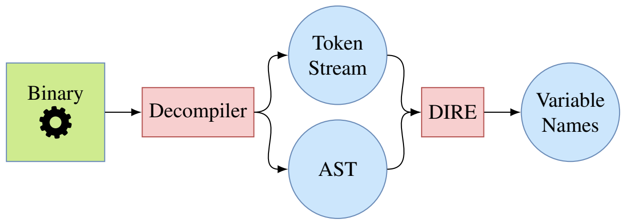


Figure 3.1: High-level overview of DIRE.

those in the original code up to 74.3% of the time. In short, the contributions of this chapter are:

- The Decompiled Identifier Renaming Engine (DIRE), a technique for assigning meaningful names to decompiled variables that outperforms previous approaches.
- A novel technique for generating corpora suitable for training both lexical and graph-based probabilistic models of variable names in decompiled code.
- A dataset of 3,195,962 decompiled x86-64 functions and parse trees annotated with gold-standard variable names.

3.1 The DIRE Architecture

I will start with a high-level overview of DIRE, then examine the details of each component.

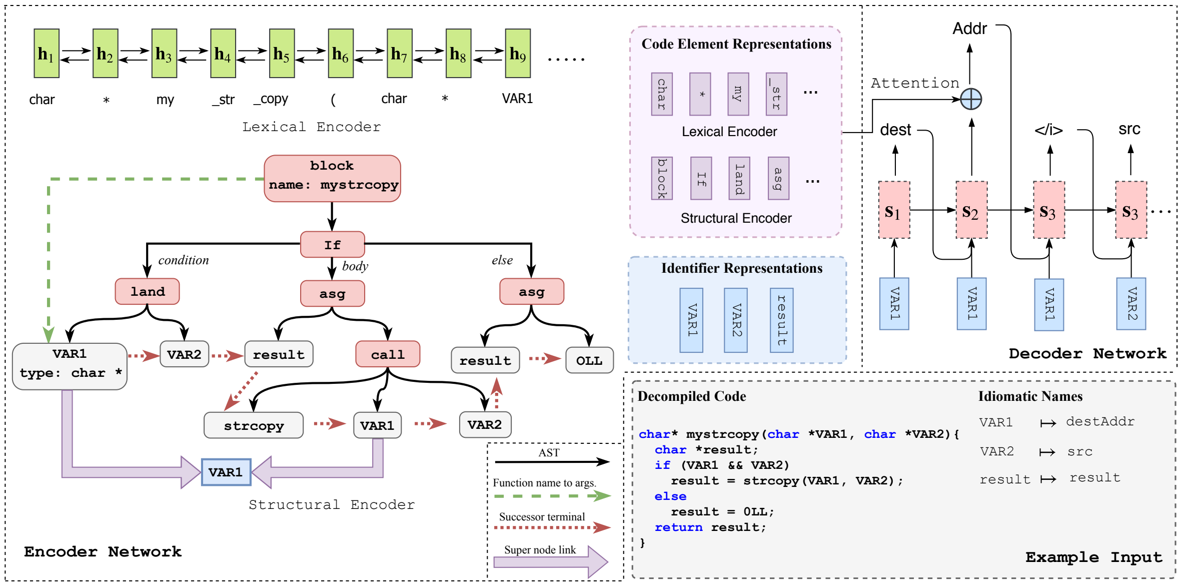
3.1.1 Overview

DIRE is designed to work on top of a decompiler as a plugin that can automatically suggest more informative variable names. We use Hex-Rays, a state-of-the-art industry decompiler, though our approach is not fundamentally coupled to Hex-Rays and can be adapted to other decompilers.

Figure 3.1 gives a high-level overview of our workflow. First, a binary is passed to a decompiler, which decompiles each function in the binary. For each function, our plugin traverses the AST, inserting placeholders at variable nodes. This produces two outputs: the AST and the tokenized code. These outputs are provided as input to our neural network model, DIRE, which generates unique variable names for each each placeholder in the input. The decompiler output can then be rewritten to include the suggested variable names.

Figure 3.2 gives an overview of the neural architecture. DIRE follows an encoder-decoder architecture: An *encoder* neural network (Section 3.1.2) first encodes the decompiler's output—both the sequence of decompiled code tokens and its internal AST—and computes distributed representations (i.e., real-valued vectors, or *embeddings*) for each identifier and code element. These encoded representations are then consumed by a *decoder* neural network (Section 3.1.3) that predicts meaningful names for each identifier based on the contexts in which it is used.

The key takeaway is that DIRE uses both lexical information obtained from the tokenized code as well as structural information obtained from the corresponding ASTs. This is achieved by using two encoders—a *lexical encoder* (Section 3.1.2) and a *structural encoder* (Section 3.1.2)—to separately capture the lexical and structural signals in the decompiled code. As I will show,



Decompiled Code

```
char* mystrcopy(char *VAR1, char *VAR2){
  char *result;
  if (VAR1 && VAR2)
    result = strcopy(VAR1, VAR2);
  else
    result = 0LL;
  return result;
}
```

Idiomatic Names

```
VAR1   ↦ destAddr
VAR2   ↦ src
result ↦ result
```

Figure 3.2: Overview of DIRE's neural architecture. For clarity, we omit the data-flow links in the AST in the structural encoder.

this combination of lexical and structural information allows DIRE to outperform techniques that rely on lexical information alone [81].

3.1.2 The Encoder Network

Each encoder network in DIRE outputs two sets of representations:

- A *code element representation* for each element in the decompiler's output. Depending on the type of the encoder, a code element will either be a token in the surface code (for the lexical encoder), or a node in the decompiler's internal AST (for the structural encoder).
- An *identifier representation* for each unique identifier defined in the input binary, which is a real-valued vector that represents the identifier in the neural network.

The lexical and structural representations are then merged to generate a unified encoding of the input binary (dashed boxes in Figure 3.2). By computing separate representations for code elements and identifiers, the DIRE decoder can better incorporate the contextual information in the encodings of individual code elements to improve name predictions for the different identifiers; I will discuss this further in Section 3.1.3.

Lexical Code Encoder

The lexical encoder sequentially encodes the tokenized decompiled code, projecting each token x_i into a fixed-length vector encoding $\boldsymbol{x}_i$. Specifically, the lexical encoder uses the sub-tokenized code as the input, where a complex code token (e.g., the function name **mystrcopy**) is automatically broken down into sub-pieces (e.g., **my**, **str**, and **copy**) using SentencePiece [87], based on sub-token frequency statistics. Sub-tokenization reduces the size of the encoder's vocabulary (and thus its training time), while also mitigating the problem of rare or unknown tokens by decomposing them into more common subtokens. We treat the placeholder and reserved variable names (e.g., `VAR1`, `VAR2`, and the decompiler-inferred name `result`) in the decompiler's output as special tokens that should not be sub-tokenized.

DIRE implements the lexical encoder using LSTMs. Formally, an LSTM has the following structure: given a sequence of tokens $\{x_i\}_{i=1}^n$, an LSTM $\overrightarrow{f}_{\text{LSTM}}$ processes them in order, maintaining a hidden state $\overrightarrow{\boldsymbol{h_i}}$ for each subsequence up to token x_i using the recurrent function $\overrightarrow{\boldsymbol{h_i}} = \overrightarrow{f}_{\text{LSTM}}(\text{emb}(x_i), \overrightarrow{\boldsymbol{h_{i-1}}})$, where $\text{emb}(\cdot)$ is an embedding function mapping x_i into a learnable vector of real numbers.

We use a bidirectional LSTM: The forward network $\overrightarrow{f}_{\text{LSTM}}$ processes the tokenized code $\{x_i\}_{i=1}^n$ sequentially. The backward LSTM processes the input tokenized code in backward order, producing a hidden state $\overleftarrow{\boldsymbol{h}_i}$ for each token x_i. Intuitively, a bidirectional LSTM captures informative context around a particular variable both before and after its sequential location.

Element Representations. We encode a token x_i by concatenating its asssociated state vectors, i.e., $\boldsymbol{x}_i = [\overrightarrow{\boldsymbol{h}_i} : \overleftarrow{\boldsymbol{h}_i}]$, a common strategy in source code representations using LSTMs [8]. For a particular token x_i we compute the forward (resp. backward) representation using both its embedding and the hidden states of its preceding (resp. succeeding) tokens. This is important because the resulting encoding $\boldsymbol{x}_i$ captures both the local and contextual information of the current token and its surrounding code.

To compute the *identifier* representation $\boldsymbol{v}$ for each unique identifier v, we collect the set of subtoken representations $\mathcal{H}_v$ of v, and perform an element-wise mean over $\mathcal{H}_v$ to get a fixed-length representation: $\boldsymbol{v} = \mathsf{MeanPool}(\mathcal{H}_v)$.

Structural Code Encoder

The lexical encoder only captures sequential information in code tokens. To also learn from the rich structural information available in the decompiler AST, DIRE employs a gated-graph neural network (GGNN) structural encoder over the AST. As described in Chapter 2, LSTMs are useful for modeling sequences, they do not capture additional sequential information.

A GGNN is defined as follows. Formally, let $\mathcal{G} = \langle V, E \rangle$ be a directed graph describing our problem, where $V = \{v_i\}$ is the set of vertices and $E = \{(v_i \mapsto v_j, \mathcal{T})\}$ is the set of typed edges. Let $\mathcal{N}_{\mathcal{T}}(v_i)$ denote the set of vertices adjacent to v_i with edge type $\mathcal{T}$. In a GGNN, each vertex v_i is associated with a state $\boldsymbol{h}_{i,t}^g$ indexed by a time step t. At each time step t, the GGNN updates the state of all nodes in V via neural message passing (NMP) [62]. Concurrently for each node v_i at time t, NMP is performed as follows: First, for each $v_j \in \mathcal{N}_{\mathcal{T}}(v_i)$ we compute a message vector $\boldsymbol{m}_{\mathcal{T}}^{v_j \mapsto v_i} = \boldsymbol{W}_{\mathcal{T}} \cdot \boldsymbol{h}_{j,t-1}^g$, where $\boldsymbol{W}_{\mathcal{T}}$ is a type-specific weight matrix. Then, all $\boldsymbol{m}_*^{v_* \mapsto v_i}$ are aggregated, and summarized into a single vector $\boldsymbol{x}_i^g$ via element-wise mean (pooling):

$$\boldsymbol{x}_i^g = \mathsf{MeanPool}(\{\boldsymbol{m}_{\mathcal{T}}^{v_j \mapsto v_i} : v_j \in \mathcal{N}_{\mathcal{T}}(v_i), \forall \mathcal{T}\}). \tag{3.1}$$

Finally, the state of every node v_i is updated using a nonlinear activation function f: $\boldsymbol{h}_{i,t}^g = f(\boldsymbol{x}_i^g, \boldsymbol{h}_{i,t-1}^g)$. GGNNs use a Gated Recurrent Unit (GRU) update function, $f_{\text{GRU}}(\cdot)$, introduced by Cho et al. [36]. By repeatedly applying NMP for T steps, each node's state gradually represents information about that node and its *context* within the graph. The computed states can then be used by a decoder, similarly to the LSTM-based decoder architectures. As in LSTMs, all GGNN parameters (parameters of $f_{\text{GRU}}(\cdot)$ and the $\boldsymbol{W}_{\mathcal{T}}$s) are optimized together simultaneously.

A GGNN requires a mechanism to compute initial node states, as well as design choices deciding which AST edges should be considered in the node encodings:

Initial Node States. The initial state of a node v_i, $\boldsymbol{h}^g_{i,t=0}$ is computed from three separate embedding vectors, each capturing different types of information of v_i:

1. An embedding of the node's syntactic type (e.g., the root note in the AST in Figure 3.2 has the syntactic type **`block`**).
2. For a node that represents data (e.g., variables, constants) or an operation on data (e.g., mathematical operators, type casts, function calls), an embedding of its data type, computed by averaging the embeddings of its subtokenized type. For instance, the variable node **`VAR1`** in Figure 3.2 has the data type **`char *`**; its embedding is computed by averaging the embeddings of the type subtokens **`char`** and **`*`**.
3. For named nodes, an embedding of the node's name (e.g., the root node in Figure 3.2 has a name **`mystrcopy`**), computed by averaging the embeddings of its content subtokens. The initial state $\boldsymbol{h}^g_{v,t=0}$ is then derived from a linear projection of the concatenation of the three separate embedding vectors. For nodes without a data type or name, we use a zero-valued vector as the respective embedding.

Graph Edges. The structural encoder uses different types of edges to capture different types of information in the AST. Besides the simple *parent-child* edges (solid arrows in the AST in Figure 3.2) in the original AST, we also augment it with additional edges [9]:

- We add an edge from the root **`block`** node containing the function name to each identifier node. The function name can inform names of identifiers in its body. In our running example the two arguments **`VAR1`** and **`VAR2`** defined in the **`mystrcopy`** function might indicate the source and destination of the copy. This type of link ("Function name to args" in Figure 3.2) captures these naming dependencies.
- To capture the dependency between neighboring code, we add an edge from each terminal node to its lexical successor ("Successor terminal").
- To propagate information among all mentions of an identifier, we add a virtual "supernode" (rectangular node labeled **`VAR1`**) for each unique identifier v_i, and edges from mentions of v_i to the supernode ("Super node link") [62].
- Finally, we add a reverse edge for all edge types defined above, modeling bidirectional information flow.

Representations. For the *element* representation, we use the final state of the GGNN for node n_i, $\boldsymbol{h}^g_{i,T}$, as its representation: $\boldsymbol{n}_i = \boldsymbol{h}^g_{i,T}$ (the recurrent process unrolls T times; $T = 8$ for all our experiments). For the *identifier* representation for each unique identifier v_i, its representation $\boldsymbol{v}_i$ is defined as the final state of its supernode as the encoding of v_i. Since the supernode has bidirectional connections to all the mentions of v_i, its state is computed using the states of all its mentions. Therefore, $\boldsymbol{v}_i$ captures information about the usage of v_i in different occurrences.

Combining Outputs of Lexical and Structural Encoders

The lexical and the structural encoders output a set of representations for each identifier and code element. In the final phase of encoding, we combine the two sets of outputs. Code elements are combined by unioning the lexical set (of code tokens) and structural set (of AST nodes) of element representations as the final encoding of each input code element; identifiers are combined by merging the lexical and structural representations of each identifier v using a linear transformation as its representation.

3.1.3 The Decoder Network

The decoder network predicts names for identifiers using the representations given by the encoder. As shown in Figure 3.2, the decoder predicts names based on both the representations of identifiers, and contextual information in the encodings of code elements. Specifically, as with the encoder, we assume an identifier name is composed of a sequence of sub-tokens (e.g., **`destAddr`** $\mapsto$ **`dest`**, **`Addr`**; see Section 3.1.2).

The decoder factorizes the task of predicting idiomatic names to a sequence of time-indexed decisions, where at each time step, it predicts a sub-token in the idiomatic name of an identifier. For instance, the idiomatic name for **`VAR1`**, **`destAddr`**, is predicted in three time steps (s_1 through s_3) using sub-tokens **`dest`**, **`Addr`**, and **`</i>`**, (the special token **`</i>`** denoting the end of the token prediction process). Once a full identifier name is generated, the decoder continues to predict other names following a pre-order traversal of the AST. As we will elaborate in Section 3.2, not all identifiers in the decompiled code will be labeled with corresponding "ground-truth" idiomatic names, since the decompiler often generates variables not present in the original code. DIRE therefore allows an identifier's decompiler-assigned name to be preserved by predicting a special **`</identity>`** token.

The probability of generating a name is therefore factorized as the product of probabilities of each local decision while generating a sub-token y_t:

$$p(Y|X) = \prod_{t=1}^{T} p(y_t|y_{<t}, X), \tag{3.2}$$

where X denotes the input code, and Y is the full sequence of sub-tokens for all identifiers, and $y_{<t}$ denotes the sequence of sub-tokens before time step t.

We model $p(y_t|y_{<t}, X)$ using an LSTM decoder, following the parameterization established in previous work [117]. Specifically, to predict each sub-token y_t, at each time step t, the decoder LSTM maintains an internal state $\boldsymbol{s}_t$ defined by

$$\boldsymbol{s}_t = f_{\text{LSTM}}([\boldsymbol{y}_{t-1} : \boldsymbol{v}_t : \boldsymbol{c}_t], \boldsymbol{s}_{t-1}), \tag{3.3}$$

where [:] denotes vector concatenation. The input to the decoder consists of two representations: the embedding vector of the previously predicted name, $\boldsymbol{y}_{t-1}$; and the encoder's representation of the current identifier to be predicted, $\boldsymbol{v}_t$.

Our decoder also uses *attention* [37] to compute a context vector $\boldsymbol{c}_t$, generated by aggregating contextual information from representations of relevant code elements. $\boldsymbol{c}_t$ is computed by taking the weighted average over encodings of AST nodes and surface code tokens, for each current sub-tokenized name y_t. The decoder's hidden state is then updated using the context vector, incorporating the contextual information into the decoder's state $\tilde{\boldsymbol{s}}_t = \boldsymbol{W} \cdot [\boldsymbol{s}_t : \boldsymbol{c}_t]$, where $\boldsymbol{W}$ is a weight matrix. Then, the probability of generating a sub-token (y_t) is:

$$p(y_t|\cdot) = \frac{\exp\left(\boldsymbol{y}_t^{\intercal}\tilde{\boldsymbol{s}}_t\right)}{\sum_{y'} \exp\left(\boldsymbol{y'}^{\intercal}\tilde{\boldsymbol{s}}_t\right)} \tag{3.4}$$

3.1.4 Training the Neural Network

Since DIRE is constructed from neural networks, training data is required to learn the weights for each neural component. Our training corpus is a set $\mathcal{D} = \{\langle X_i, Y_i \rangle\}$, consisting of pairs of code X and sub-token sequences Y, denoting the decoder-predicted sequence of identifier names. DIRE is optimized by maximizing the log-likelihood of predicting the gold sub-token sequence Y_i for each training example X_i:

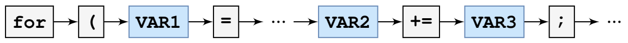
$$\sum_{\langle X_i, Y_i \rangle} \log p(Y_i|X_i) = \sum_{\langle X_i, Y_i \rangle} \sum_{t=1}^{|Y_i|} w_t \cdot \log p(y_{i,t}|X_i), \tag{3.5}$$

where $Y_{i,t}$ denotes the t-th sub-token in the decoder's prediction sequence Y_i. As discussed in Section 3.1.3, there are intermediate variables in the decompiled code. To ensure the decoder network will not be biased towards predicting **`</identity>`** for other identifiers, we use a tuning weight w_i set to 0.1 for sub-tokens that correspond to intermediate variables (and 1.0 otherwise).

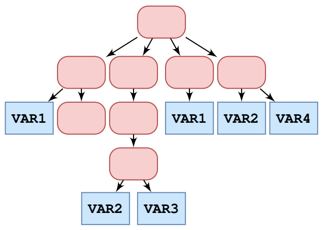
3.2 Generation of Training Data

Training DIRE requires a large corpus of annotated data. Fortunately, it is possible to create this corpus automatically, starting from a large repository of existing C source code. At a high level, each entry in our corpus corresponds to a source code function, and consists of the information necessary to train our model. An entry in the training corpus is illustrated in Figure 3.3. Each entry contains three elements: (a) the tokenized code, with variables replaced by an ID that uniquely identifies the variable in the function; (b) the decompiler's AST modified to contain the same unique variable IDs; and (c) a lookup table mapping variable IDs to both the decompiler- and developer-assigned names. It is important to assign a unique variable name to each variable to disambiguate any shadowed variable definitions. The tokenized code and AST representations are used in both the model's input and output. The input representation uses the decompiler-assigned names, while the output uses the developer-assigned names.

Generating the placeholders and decompiler-chosen names is relatively straightforward. First, a binary is compiled normally and passed to the decompiler. Next, for each function, we traverse its AST and replace each variable reference with a unique placeholder token. Finally, we instruct the decompiler to generate decompiled C code from the modified AST, tokenizing the output. Thus, we have tokenized code, an AST, and a table mapping IDs to decompiler-chosen names.



(a) Tokenized decompiled code with variable placeholders.



(b) AST with placeholders.

ID	Decompiler	Developer
1	`v1`	`ans`
2	`v2`	`size`
3	`i`	`i`
4	`ptr`	`head`

(c) Variable lookup table.

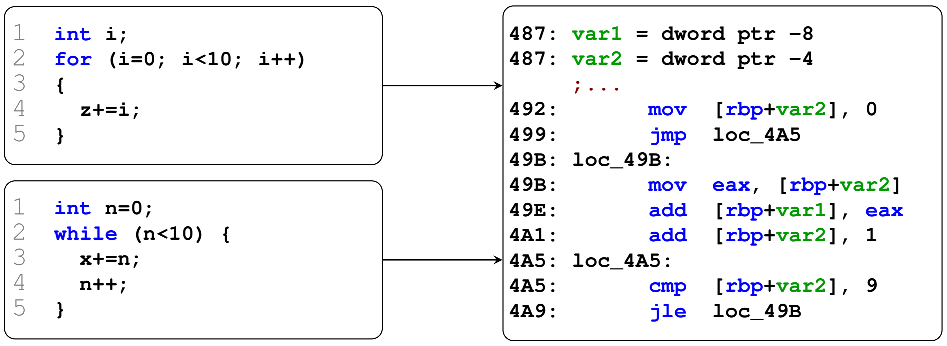
Figure 3.3: Entry in the training corpus. Each corresponds to a function and contains (a) tokenized code (b) the AST, both with variables replaced with unique IDs, and (c) a lookup table containing decompiler- and developer-assigned names.

The remaining step, mapping developer-chosen names to variable IDs, is the core challenge in automatic corpus generation. Following our previous approach [81], we leverage the decompiler's ability to incorporate developer-chosen identifier names into decompiled code when DWARF debugging symbols [49] are present. However, this alone is not sufficient to identify which developer-chosen name maps to a particular variable ID generated in the first step.

Specifically, challenges arise because decompilers use debugging information to enrich the decompiler output in a variety of ways, such as improving type information. Recall from Section 2.1.2 that decompilers often make choices between semantically-identical structures: the addition of debugging information can change which structure is used. Unfortunately, this means that the difference between code generated with and without debugging symbols is not always an α-renaming. In practice, the format and structure of the code can greatly differ between the two cases. An example is illustrated in Figure 3.4. In this example, the first pass of the decompiler is run without debugging information, and the decompiler generates an AST for a **`while`** loop with two automatically-generated variables named **`v1`** and **`v2`**. Next, the decompiler is passed DWARF debugging symbols and run a second time, generating the AST on the right. While the decompiler is able to use the developer-selected variable names **`i`** and **`z`**, it generates a very different AST corresponding to a **`for`** loop.

An additional challenge is that there is not always a complete mapping between the variables in code generated with and without debugging information. Decompilers often generate more variables than were used in the original code. For example, **`return (x + 5);`** is commonly decompiled to **`int v1; v1 = x + 5; return v1;`**. The decompiled code introduces a temporary variable **`v1`** that does not correspond to any variable in the original source code. In this case, there is no developer-assigned name for **`v1`**, since it does not exist in the original code. The use of debugging information can change how many of these additional variables are generated.

One solution to these problems proposed by prior work is to post-process the decompiler output using heuristics to *align* decompiler-assigned and developer-assigned names [81]. However, this technique can only correctly align 72.8% of variable names, therefore limiting the overall accuracy of any subsequent model trained on this data. Instead, we developed a technique that di-

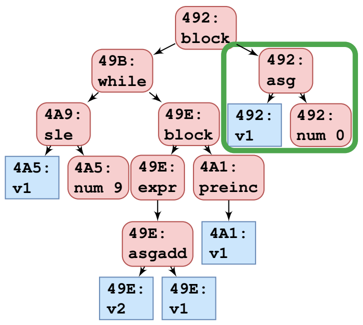


```c
int i;
for (i=0; i<10; i++)
{
  z+=i;
}
```

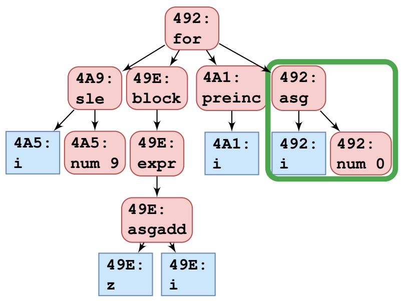
```c
int n=0;
while (n<10) {
  x+=n;
  n++;
}
```

```
487: var1 = dword ptr -8
487: var2 = dword ptr -4
     ;...
492:        mov   [rbp+var2], 0
499:        jmp   loc_4A5
49B: loc_49B:
49B:        mov   eax, [rbp+var2]
49E:        add   [rbp+var1], eax
4A1:        add   [rbp+var2], 1
4A5: loc_4A5:
4A5:        cmp   [rbp+var2], 9
4A9:        jle   loc_49B
```

(a) Two different C loops that compile to the same assembly code.



(b) AST without DWARF.



(c) AST with DWARF.

Figure 3.4: Decompiler ASTs for the code shown in (a). Hexadecimal numbers indicate the location of the disassembled instruction used to generate the node. While the ASTs are different, operations on variables and their offsets are the same, enabling mapping between variables, i.e., **v1**$\mapsto$**i** (green box) and **v2**$\mapsto$**z**.

rectly integrates with the decompiler to generate an accurate alignment *without using heuristics*. Our key insight is that while the AST and code generated by the decompiler may change when debugging information is used, *instruction offsets and operations on variables do not change*. As a result, each variable can be uniquely identified by the set of instruction offsets that access that variable.

For example, in Figure 3.4, although there is not an obvious mapping between the nodes in the trees, the addresses of the variable nodes in the trees have not changed. This enables us to uniquely identify each variable by creating a signature consisting of the set of all offsets where it occurs. The variables **`v1`** and **`i`** have the signature `{492,49E,4A1,4A5}`, while **`v2`** and **`z`** have the signature `{49E}`. Note that some uses of variables overlap, e.g., **`v1`** (**`i`**) is summed with **`v2`** (**`z`**) in the instruction at offset `49E`. This necessitates collecting the full set of variable uses to disambiguate these instances.[1]

In summary, to generate our corpus we:

1. Decompile binaries containing debugging information.
2. Collect signatures and developer-assigned names for each variable in each function.
3. Strip debugging information and decompile the stripped binaries.
4. Identify variables by their signature, and rename them in the AST, encoding both the decompiler- and developer-assigned names.
5. Generate decompiled code from the updated AST.
6. Post-process the updated AST and generated code to create a corpus entry.

The final output is a per-binary file containing each function's AST and decompiled code with corresponding variable renamings.

3.3 Evaluation

We ask the following research questions:

- **RQ1**: How effective is DIRE at assigning names to variables in decompiled code?
- **RQ2**: How does each component of DIRE contribute to its efficacy?
- **RQ3**: How does provenance and quantity of data influence the efficacy of DIRE?
- **RQ4**: Is DIRE more effective than prior approaches?

Data Preprocessing. To answer our first two research questions, we trained DIRE on 3,195,962 decompiled functions extracted from 164,632 binaries mined from GITHUB. First, we automatically scraped GITHUB for projects written in C. Next, we modified project build scripts to include debug information when compiling the project, and collected all successfully generated 64-bit x86 binary files. We then hashed each binary to remove any duplicates. We then passed these binaries through our automated corpus generation system.

[1]While it is possible for two variable signatures to be identical, we found these collisions to occur very rarely in practice. In these cases we do not attempt to assign names to variables.

Finally, we filtered out any functions that did not have any renamed variables and, for practical reasons, any functions with more than 300 AST nodes. After filtering, 1,259,935 functions with an average AST size of 77 nodes remained. These functions were randomly split per-binary into training, development and testing sets with a ratio of 80:10:10. Splitting the sets per-binary ensures that binary-specific identifiers are not included in both the training and test sets.

Evaluation Methodology. After training, we ran DIRE to generate name suggestions on the test data. We evaluate the accuracy of these predictions, comparing the predicted variable names to names used in the original code (i.e., names contained in the debugging information) counting a successful prediction as one exactly matching the original name. However, there can be multiple, equally acceptable names (e.g., `file_name`, `fname`, `filename`) for a given identifier. An accuracy metric based on exact match cannot detect these cases. We therefore use character error rate (CER), a metric that calculates the edit distance between the original and predicted names, then normalizes by the length of the original name [166], assigning partial credit to near misses.

Recall from Section 3.2 that there are often many more variables in the decompiled code than in the original source; these variables will not have a corresponding original name. Although DIRE generates predictions for these variables, we do not evaluate them. We do this because it is not necessarily incorrect for a renaming system to assign names to variables not present in the original source code. Recall the example where `return (x + 5);` is decompiled to `int v1 ; v1 = x + 5; return v1;`. The name `sum` is likely more informative than `v1`, and it would be unhelpful to penalize a system that suggests this renaming. However, although renaming in these cases could be helpful, we do not want to overapproximate the effectiveness of our system by claiming any renaming of these variables as correct: it is also possible to assign variables a misleading name that *decreases* the readability of code by obfuscating the purpose of a variable (something that will be demonstrated in Chapter 5). For example, suggesting the name `filename` to replace `v1` in the above code would likely be misleading.

Neural Network Configuration. For our experiments we replicate the neural network configuration of Allamanis et al. [9]. We set the size of word embedding layers to be 128. The dimensionality of the hidden states for the recurrent neural networks used in the encoders is 128, while the hidden size for the decoder LSTM is 256. For both the sequential and structural encoders, we use two layers of recurrent computation, adding another identical recurrent network to process the decompiled code using the output hidden states of the first layer. For both DIRE and the baseline neural systems, we train each model for 60 epochs. At testing time, we use beam search to predict the sequence of sub-tokenized names for each identifier (Section 3.1.3), with a beam size of 5.

3.3.1 RQ1: Overall Effectiveness

The experimental results are summarized in Table 3.1. The "Overall" row shows the performance of our technique on the full test set and the leftmost column shows the accuracy of DIRE. From this, we can see that DIRE can recover 74.3% of the original variable names in decompiled code,

Table 3.1: Evaluation of DIRE. Values are percentages, higher accuracy and lower character error rate (CER) are better.

	DIRE		Lexical Encoder		Structural Encoder	
	Accuracy	CER	Accuracy	CER	Accuracy	CER
Overall	**74.3**	**28.3**	72.9	28.5	64.6	37.5
Body in Train	**85.5**	**16.1**	84.3	16.3	75.7	25.5
Body not in Train	**35.3**	**67.2**	33.5	67.7	26.3	76.1

```
void *file_mmap(int V1, int V2)
{
  void *V3;
  V3 = mmap(0, V2, 1, 2, V1, 0);
  if (V3 == (void *) -1) {
    perror("mmap");
    exit(1);
  }
  return V3;
}
```

	DIRE	Developer
V1	fd	fd
V2	size	size
V3	buf	ret

Figure 3.5: Decompiled function (simplified for presentation), DIRE variable names, and developer-assigned names.

demonstrating that it is effective in assigning contextually meaningful names to identifiers in decompiled code.

Figure 3.5 shows an example renaming generated by DIRE. Here, DIRE generates the variable names shown in the "DIRE" column of the table. The developer-chosen names are shown in the "Developer" column. Two of three names suggested by DIRE exactly match those chosen by the developer. Though DIRE suggests `buf` instead of `ret` as the replacement for `V3`, the name is not entirely misleading: `mmap` returns a pointer to a mapped area of memory that can be written to or read from.

Work has shown that large code corpora may contain near-duplicate code across training and testing sets, which can cause evaluation metrics to be artificially inflated [4]. Though our corpus contains no duplicate binaries, splitting test and training sets per-binary still results in functions appearing in both. A common cause of duplicate functions in different binaries is the use of libraries. We argue that it is reasonable to allow such duplication since reverse-engineering binaries that link against known (e.g., open source) libraries is a realistic use case.

Nevertheless, to better understand the performance of our system, we partition the test examples into two sub-categories: ***Body in Train*** and ***Body not in Train***. The *Body in Train* partition includes all functions whose entire body matches at least one function in the training set; similarly, the *Body not in Train* set includes only functions whose body does not appear in the training set. The last two rows in Table 3.1 show the performance on these partitions. DIRE performs well on the *Body in Train* test partition (85.5%). This indicates that DIRE is particularly accurate at name prediction when code has appeared in its training set (e.g., libraries, or code copied from another project). DIRE is still able to exactly match 35.3% of variable names in the *Body not in Train* set, indicating that it still generalizes to unseen functions.

Table 3.2: Example identifiers from the *Body not in Train* testing partition and DIRE's top-5 most frequent predictions.

`len`	`value`	`new_node`	`bytes_read`
`len` (60%)	`value` (28%)	`node` (48%)	`size` (38%)
`n` (6%)	`data` (7%)	`child` (31%)	`bytes_read` (13%)
`size` (5%)	`val` (3%)	`treea` (0.3%)	`len` (13%)
`length` (1%)	`name` (3%)	`tree` (0.3%)	`cmd_code` (13%)
`l` (1%)	`key` (2%)	`root` (0.3%)	`read` (13%)

```
file *f_open(char **V1, char *V2, int V3) {
  int fd;
  if (!V3)
    return fopen(*V1, V2);
  if (*V2 != 119)
    assert_fail("fopen");
  fd = open(*V1, 577, 384);
  if (fd >= 0)
    return fdopen(fd, V2);
  else
    return 0;
}
```

	Lex.	Struct.	DIRE	Developer
V1	`file`	`fname`	`filename`	`filename`
V2	`name`	`oname`	`mode`	`mode`
V3	`mode`	`flags`	`create`	`is_private`

Figure 3.6: Decompiled function (simplified for presentation), suggested names, and developer-assigned names. The lexical and structural models are unable to correctly predict the name **`mode`** for **`V2`**, but DIRE can by combining them.

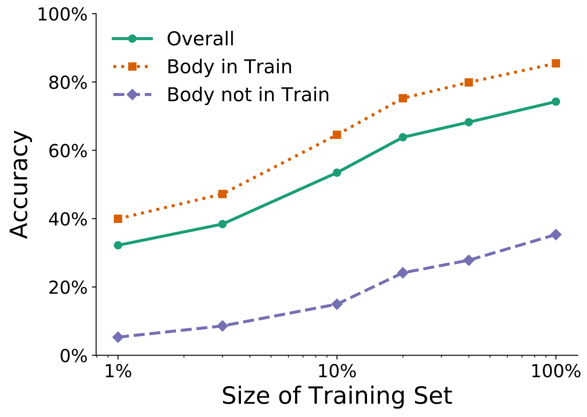
Table 3.2 contains example identifiers from the *Body not in Train* test set, along with DIRE's most frequent predictions. We observe that inexact suggested names are often semantically similar to the original names. DIRE also performs best on simple identifiers such as **`len`** and **`value`**. This is because it is difficult to predict the exact name for complex identifiers with compositional names. However, DIRE is still often outputs semantically relevant names (e.g., **`node`**, **`child`**).

RQ1 Answer: We find that DIRE is able to suggest variable names identical to those chosen by the original developer 74.3% of the time.

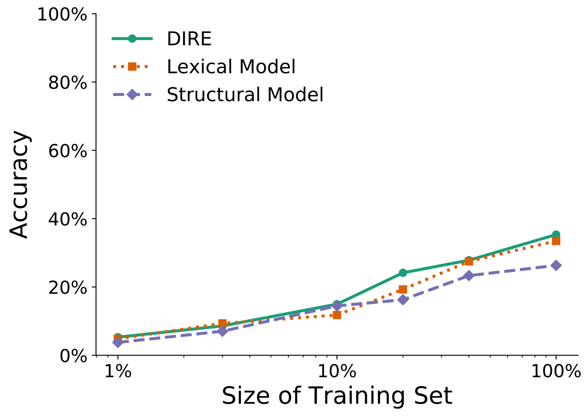
3.3.2 RQ2: Component Contributions

Table 3.1 also shows the results for models that only used our lexical or structural encoders. We find that the lexical encoder is able to correctly predict 72.9% of the original variable names, while a model using the structural encoder is able to correctly predict 64.6% of the original variable names. These simpler models still perform well, but by combining them in DIRE we are able to achieve even better performance.

Figure 3.6 illustrates how DIRE can effectively combine these models to improve suggestions. Here, the placeholders **`V1`**, **`V2`**, and **`V3`** are variables which should be assigned names. The "Lex.", "Struct.", and "DIRE" columns show the predictions from each model, and the "Developer" column shows the name originally assigned by the developer. In this example, the lexical and the structural models are unable to predict any of the original variable names, while DIRE is



(a) Accuracy of DIRE (higher is better).



(b) Accuracy of each neural model on the *Body not in Train* partition.

Figure 3.7: The impact of training corpus size on the performance of DIRE. Figure (a) shows how increasing the amount of training data improves the performance of DIRE; (b) shows the performance of each of the submodel as training size changes.

able to correctly predict two of the three names.

This example also shows the contributions from each of the submodels. For example, for `V1`, the lexical model predicts **`file`** while the structural model predicts **`fname`**. Combining the predicted subtokens generates **`filename`**, the same name chosen by the developer. For `V2`, the lexical and structural models both fail to predict **`mode`**, but note that the lexical model *does* predict **`mode`** for `V3`. By combining the models, DIRE instead correctly predicts **`mode`** for `V2`.

RQ2 Answer: Each component of DIRE contributes uniquely to its overall accuracy.

3.3.3 RQ3: Effect of Data

To answer RQ3, we varied the size of the training data and measured the change in performance of our models. Training data was subsampled at rates of 1%, 3%, 10%, 20%, and 40%. The results of these experiments are shown in Figure 3.7.

Figure 3.7a shows the change in accuracy of DIRE. The size of the training data is plotted on the x-axis, while accuracy is plotted on the y-axis. While DIRE has low accuracy on the *Body not in Train* set at the lowest sampling rates, at a 1% sampling rate it is still able to correctly select names over 40% of the time for the *Body in Train* test set, suggesting that it is possible to use much less data to train a model if the target application is reverse engineering of libraries rather than binaries in general. At a sampling rate of 40%, DIRE comes quite close to the performance of the model trained on the full training set, with an overall accuracy of 68.2% (vs. 74.2%).

Figure 3.7b shows the effect of training set size on the performance of DIRE and its component neural models on the *Body not in Train* test set. Note how at sampling rates at or below 10% the models have similar performance. In cases where there is little training data, training time

```
long gray(unsigned a1, int a2) {
  unsigned v3, v4;
  int v5;
  if (a2 >= 0)
    return a1 ^ (a1 >> 1);
  v5 = 1;
  v4 = a1;
  while (1) {
    v3 = v4 >> v5;
    v4 ^= v4 >> v5;
    if (v3 <= 1 ||
        v5 == 16)
      break;
    v5 *= 2;
  }
  return v4;
}
```

(a) Hex-Rays.

```
void gray() {
  unsigned v0;
  int v1;
  unsigned i, v3;
  int x;
  if (v1 < 0) {
    x = 1;
    v3 = v0;
    while (1) {
      i = v3 >> x;
      v3 ^= v3 >> x;
      if (i <= 1 ||
          x == 16)
        break;
      x *= 2;
    }
  }
}
```

(b) Hex-Rays w/ DEBIN.

Figure 3.8: Effects of incorrect debugging information on decompiler output. The **`gray`** function computes the Gray code of **`a1`** in **`a2`** bytes [134]. On the left, (a) is the output of Hex-Rays without debugging symbols; it is able to correctly identify the arguments and return type. On the right, (b) is the output with incorrect DWARF information generated by DEBIN: note missing arguments, **`return`** statements, and incorrect type.

can be further reduced by using only one of the two submodels.

RQ3 Answer: DIRE is data-efficient, performing competitively using only 40% of the training data. DIRE is also robust, outperforming the lexical and structural models in most sub-sampling cases.

3.3.4 RQ4: Comparison to Prior Work

To answer RQ4, we compare to our prior work [81] and to DEBIN [68], the state-of-the-art technique for predicting debug information directly from binaries.

In our earlier work, which used a purely-lexical model based on statistical machine translation (SMT), we were able to exactly recover 12.7% of the original variable names chosen by developers. In contrast, DIRE is able to suggest identical variable names 74.3% of the time. We attribute this improvement to two factors: 1) the improved accuracy of our corpus generation technique, and 2) the use of a model that incorporates both lexical and structural information.

To better understand the performance of DIRE, we also compare to DEBIN, a different approach to generating more understandable decompiler output. DEBIN uses CRFs to learn models of binaries and directy generate DWARF debugging information for a binary, which can be used by a decompiler such as Hex-Rays.

The debugging information generated by DEBIN contains predicted identifiers, types, and names. To choose a variable name, DEBIN proceeds in two stages: it predicts which memory locations correspond to function-local arguments and variables, then predicts names for them. In

Table 3.3: Comparison of DIRE and DEBIN trained on 1% and 3% of our full corpus of 164,632 binaries. All accuracy values are percentages, higher accuracy is better. Note that DIRE is able to achieve much higher accuracy than DEBIN at all sampling sizes.

	1% of Corpus		3% of Corpus	
	DIRE	DEBIN	DIRE	DEBIN
Training Time (hours)	1.8	13.3	6.1	17.2
Accuracy – Overall	32.2	2.4	38.4	3.9
Accuracy – Body in Train	40.0	3.0	47.2	4.8
Accuracy – Body not in Train	5.3	0.6	8.6	0.7

contrast, DIRE leverages the decompiler to identify function offsets and local variables.

Building on top of the decompiler helps DIRE maintain the quality of pseudocode output. To demonstrate why this is important, refer to the example shown in Figure 3.8, which contains a C function for converting between a number `a1` and its Gray code representation in `a2` bits [134]. Figure 3.8a shows the output of Hex-Rays when passed a binary with no debug information. Although these variables do not have meaningful names, it is clear that `gray` is a function that takes two arguments and returns a `long`.

Figure 3.8b shows the output of Hex-Rays using debugging information generated using DEBIN's bundled model.[2] We observe that DEBIN does not accurately recover variable names in this case, perhaps since its model was trained on a different set of code.

However, this example also surfaces a fundamental limitation of the DEBIN approach: both the inferred structure and the types of the variables in the program have changed. This occurs because Hex-Rays prioritizes debugging information over its own analyses and heuristics. In this case, the debugging information generated by DEBIN does not indicate a return value of the `gray` function nor any arguments, misleading the decompiler. By starting at the point shown in Figure 3.8a, DIRE maintains structure and typing even in the presence of incorrect predictions.

To evaluate our performance compared to DEBIN, we trained it on binaries in our dataset. Due to time restrictions, we found it impractical to train DEBIN on the full dataset. For a fair comparison, we instead subsampled our training set at 1% and 3% and trained both DEBIN and DIRE on these sets. Although this might seem small, we note that the 3% subsampling we used corresponds to 30,238 binaries, a full order of magnitude more than the 3,000 binaries used to train DEBIN in their original paper [68]. After training, we ran DEBIN on binaries in our test set, extracted names using our corpus generation pipeline, and measured the accuracy of predictions. Our results are shown in Table 3.3.

We find that DIRE is able to outperform DEBIN at all sampling sizes. When trained on 1% of the corpus DIRE is able to exactly recover 32.2% of all identifiers, while DEBIN recovers 2.4%. On the 3% partition, DIRE is able to recover 38.4% of names, while DEBIN is able to recover 3.9%. The lower performance of DEBIN we observed could be attributed to compound error: in addition to variable names themselves, DEBIN must predict what memory locations correspond to variables. If a memory location is not predicted to be a variable, DEBIN cannot assign it a

[2]https://files.sri.inf.ethz.ch/debin_models.tar.gz, accessed April 10, 2019

name.

We also note that we were able to train DIRE much faster than DEBIN, although DIRE is GPU-accelerated, while DEBIN as distributed is limited to execution on the CPU.

RQ4 Answer: DIRE is a more accurate and more scalable technique for variable name selection than other state-of-the-art approaches.

3.4 Threats To Validity

When collecting code and binaries to generate our corpus, we did no filtering of the repositories beyond ensuring that they were written in C and able to be compiled. It is possible that the code we collected does not accurately represent the types of binaries that are typically targets of reverse-engineering effort.

Additionally, we did not experiment with binaries compiled with optimization enabled, nor did we experiment with intentionally obfuscated code. It is possible that DIRE does not perform as well on these binaries. However, reverse engineering of these binaries is a general challenge for decompilers, and we do not believe that our technique applies exclusively to the test code we experimented with.

Although we have found that it is possible to uniquely identify variables in Hex-Rays based on the code offsets where it is accessed, we have found that other decompilers do not have this property. In particular, our approach did not work well with the Ghidra decompiler [61]. One of the primary causes is the way that Hex-Rays and Ghidra utilize debug symbols to name variables. Hex-Rays uses debug symbols in a very straight-forward manner, and generally does not propagate local names outside of their function. Ghidra, however, will actually propagate variable names at some function calls. For example, if an unnamed variable is passed as an argument to a function whose parameter has a name, in some cases Ghidra will rename the variable to match the parameter's name. This behavior is problematic for corpus generation because it does not reflect the developer's intended names.

A new approach for corpus generation would be required for compatibility with Ghidra, but Ghidra's open-source nature (as opposed to Hex-Rays' closed model) allows potential modification of the decompiler, including disabling the problematic propagation of names at function calls. We leave Ghidra integration to future work.

3.5 Conclusion

The focus of this thesis is on the augmentation of decompiled code with both variable names and types. In this chapter, I described the Decompiled Identifier Renaming Engine (DIRE), a novel, probabilistic technique for variable name recovery which uses both lexical and structural information. I also presented a technique for generating corpora suitable for training DIRE and other recovery techniques, which was used to generate a corpus from 164,632 unique x86-64 binaries. The experiments presented show that DIRE is able to predict variable names identical to the names used in the original source code up to 74.3% of the time. I also demonstrated how the technique is scalable to more or less data, depending on the training time and resources available. I also demonstrated that DIRE is more accurate and scalable than other existing techniques.

Given these results, several questions still remain. First, is this technique also applicable to the recovery of variable types? Second, the name of a variable is obviously dependent on its type (e.g., a **`point`** type variable would be much more likely to be named **`pnt`** than **`size`**), does the inclusion of types improve the performance of renaming? In the next chapter, I will discuss the answers to these questions and describe specific challenges that arise when creating a model for the task of retyping.

Chapter 4

Augmenting Decompiler Output with Learned Types

In this chapter I will focus on the closely related problem of recovering meaningful variable *types*, an important additional layer of code documentation that can help improve readability and understandability [52, 145, 158]. Figure 4.1 shows an example of a simple function and its decompilation. The author of the original code in Figure 4.1a has defined a `pnt` structure that contains two `float` members used to refer to the X and Y coordinates of a point. This makes it possible to define a new point and refer to its members by name (e.g., `p1.x` and `p1.y`). Because the decompiler does not know about the `pnt` structure, it creates two `float` arrays instead of generating a `struct` (Figure 4.1b). This can harm understandability. First, it is not clear that `v1` and `v2` represent points. Second, even if better names were chosen, such as `point1` and `point2`, and a reverse engineer concluded that they represent 2D points, it is not clear which array index refers to which coordinate, or even that the coordinates are Cartesian (instead of polar).

Unlike names, types are constrained by memory layouts, and thus theoretically should be easier to recover (only types that fit that memory layout should be considered as candidates). In fact, decompilers already narrow down possible type choices using the fact that base types targeting a specific platform can only be assigned to variables with a specific memory layout

```c
typedef struct point {
  float x;
  float y;
} pnt;

void fun() {
  pnt p1, p2;
  p1.x = 1.5;
  p1.y = 2.3;
  // ...
  use_pts(&p1, &p2);
}
```

(a) Original code

```c
void fun() {
  float v1[2], v2[2];
  v1[0] = 1.5;
  v1[1] = 2.3;
  // ...
  use_pts(v1, v2);
}
```

(b) Decompiled `fun`

Figure 4.1: A function with a `struct` and its decompilation.

```
void fun() {
  // stack layout:
  // [xxx][p][yyyy]
  char x[3];
  int y;
  // ...
}
```

(a) Original code

```
void fun() {
  // stack layout:
  // [xxxx][yyyy]
  char x[4];
  int y;
  // ...
}
```

(b) Decompiled `fun`

Figure 4.2: A function illustrating the data layout problem in decompilation. In the stack layout the characters `x`, `y`, and `p` represent a single byte assigned to the variables `x` and `y`, or padding data respectively. The decompiler cannot recognize that the inserted padding data does not belong to the `x` array.

(e.g., on most platforms an `int` variable can never be retyped to a `char` because they require different amounts of memory). This already makes it possible for decompilers to infer base types and a small set of commonly-used `typedefs`.

On the other hand, despite performing a battery of complex binary analyses, the data layout inferred by the decompiler is often incorrect, which makes the problem harder. For example, consider the program shown in Figure 4.2. Two top-level variables are declared, `x`: a three-byte `char` array, and `y`: a four-byte `int`. During compilation, the compiler inserts a single byte of padding after the `x` array for alignment. When this function is decompiled, the decompiler can tell where `x` and `y` begin, but it cannot tell if `x` is a three-byte array followed by a single byte of padding, or a four-byte array whose last element is never used.

Prior work on reconstructing types falls into two groups. The first, such as TIE [96], attempt to recover *syntactic* types, e.g., `struct {float; float}`, but not the names of the structures or fields. The second, such as REWARDS [109], attempt to also recover the type name (i.e., *semantic* types). However, these systems typically only support a small set of manually-defined types and well-known library calls. Neither address the padding issue above described.

In contrast, our system, the DecompIled variable ReTYper (DIRTY) recovers both semantic and syntactic types, handles padding, and is not limited to a small set of manually-defined types. Instead, DIRTY supports 48,888 possible types encountered "in the wild" in open-source C code (compared to the 150 different type names in 84 standard library calls supported by REWARDS). At a high level, DIRTY is a *Transformer-based [161] neural network model to recommend types* in a particular context, which operates as a postprocessing step to decompilation. DIRTY takes a decompiled function as input, and outputs probable names and types for all of its variables.

To build DIRTY, we start by mining open-source C code from GITHUB, and then use a decompiler's typical ability to import variable names and types from DWARF debugging information to create a *parallel corpus of decompiled functions with and without their corresponding original names and types*. As a side effect of this large-scale mining effort, we also automatically compile a *library of types* encountered across our open-source corpus. We then train DIRTY on this data, introducing two task-specific innovations. First, we use a data layout encoder to incorporate memory layout information into DIRTY's predictions and simultaneously address a fundamental limitation of decompilers caused by padding. Second, we address both the variable renaming and retyping tasks simultaneously with a joint Multi-Task architecture, enabling them to benefit from each other.

We show that DIRTY can assign variable *types* that agree with those written by developers up to 75.8% of the time, and DIRTY also outperforms prior work on variable *names*.

Note that even though we implement DIRTY on top of the Hex-Rays[1] decompiler because of its positive reputation and its programmatic access to decompiler internals, our approach is not fundamentally specific to Hex-Rays, and should conceptually work with any decompiler that names variables using DWARF debug symbols.

In summary, the contributions of this chapter are:

- DIRT—the Dataset for Idiomatic ReTyping—a large-scale public dataset of C code for training models to retype or rename decompiled code, consisting of nearly 1 million unique functions and 368 million code tokens.
- DIRTY—the DecompIler variable ReTYper—an open-source Transformer-based neural network model to recover syntactic and semantic types in decompiled variables. DIRTY uses the *data layout* of variables to improve retyping accuracy, and is able to *simultaneously retype and rename* variables in decompiled code.

4.1 Model Design

In this section, we describe our machine learning model and design decisions, starting with relevant background. Our model is a *neural network* with an *encoder-decoder* architecture.

4.1.1 The Encoder-Decoder Architecture

Our task consists of generating variable types (and names) as output given individual functions in decompiled code as input. This means that unlike a traditional classification problem with a fixed number of classes, both our input and output are sequences of variable length: input functions (e.g., fed into the network as a sequence of tokens) can have arbitrarily many variables, each requiring a type (and name) prediction.

Therefore, we adopt an encoder-decoder architecture [36], commonly used for sequence-to-sequence transformations, as opposed to the traditional feed-forward neural network architecture used in classification problems with a fixed-length input vector and prediction target. More specifically, the *encoder* takes the variable-length input and encodes it as a fixed-length vector. Then, this fixed-length encoding is passed to the *decoder*, which converts the fixed-length vector into a variable-length output sequence. This architecture, further enhanced through the *attention* mechanism [12], has been shown to be effective in many tasks such as machine translation, text summarization [126], and image captioning [176].

4.1.2 Transformers

There are several ways to implement an encoder-decoder. Until recently, the standard implementation used a particular type of recurrent neural network (RNN) with specialized neurons called

[1]`https://www.hex-rays.com/products/decompiler/`

long short-term memory units; these neurons and networks constructed from them are commonly referred to as LSTMs [73]. More recently, Transformer-based models [19, 54, 136, 180], building on the original Transformer architecture [161], have been shown to outperform LSTMs and are considered to be the state-of-the-art for a wide range of natural language processing tasks, including machine translation [19], question answering and abstractive summarization [44, 97], and dialog systems [2]. Transformer-based models have also been shown to outperform convolutional neural networks such as ResNet [69] on image recognition tasks [46].

Transformers have several properties that make them a particularly good fit for our type prediction task. First, they capture long-range dependencies, which commonly occur in program code, more effectively than RNNs. For example, a variable declared at the beginning of a function may not be used until much later; an ideal model captures information about all uses of a variable. Second, transformers can perform more computations in parallel on typical GPUs than LSTMs. As a result, training is faster, and a Transformer can train on more data in the same amount of time. In our case, this enables us to train on our large-scale, real-world dataset, which consists of 368 *million* decompiled code tokens.

Although there have been a number of advances in neural machine translation since the original Transformer model [161], most recent advances focus on improvements on other factors, such as training data and objectives [19, 44, 97, 136], dealing with longer sequences [180], efficiency [35], and scaling [54], rather than changing the fundamental architecture. Moreover, most of these improvements are tailored for the natural language domain, making them less generalizable than the original model and inapplicable to our task. Instead, we keep our model simple, which allows different, better architectures or implementations to be used out-of-the-box in the future. For example, the recent Vision Transformer (ViT) [46], which also intentionally follows the original Transformer architecture "as closely as possible" when adapting Transformers to computer vision tasks.

I omit the technical details of Transformers, including multi-headed self-attention, positional encoding, and the specifics of training as they are beyond the scope of this thesis.

4.1.3 DIRTY's Architecture

In DIRTY, we cast the retyping problem as a transformation from a sequence of tokens representing the decompiled code to a sequence of types, one for each variable in the source code. This section details DIRTY's architecture. Figure 4.3 shows an overview of the architecture.

Code Encoder. The encoder converts the sequence of code tokens of the decompiled function (lower-left of Figure 4.3), $x = (x_1, x_2, \ldots, x_n)$, into a sequence of representations,

$$\mathbf{H} = (\mathbf{h_1}, \mathbf{h_2}, \ldots, \mathbf{h_n}), \tag{4.1}$$

where each continuous vector $\mathbf{h_i} \in \mathbb{R}^{d\_model}$ is the *contextualized representation* for the i-th token x_i. During training, the encoder learns to encode the information in the decompiled function x relevant to solving the task into $\mathbf{H}$. For example, for a code token $x_i =$ `v1`, useful information about `v1` in the context of x (e.g., operations using `v1`) is automatically learned and stored in $\mathbf{h_i}$.

Specifically, we denote the encoding procedure as

$$\mathbf{H} = f_{en}(x; \theta_{en}), \tag{4.2}$$

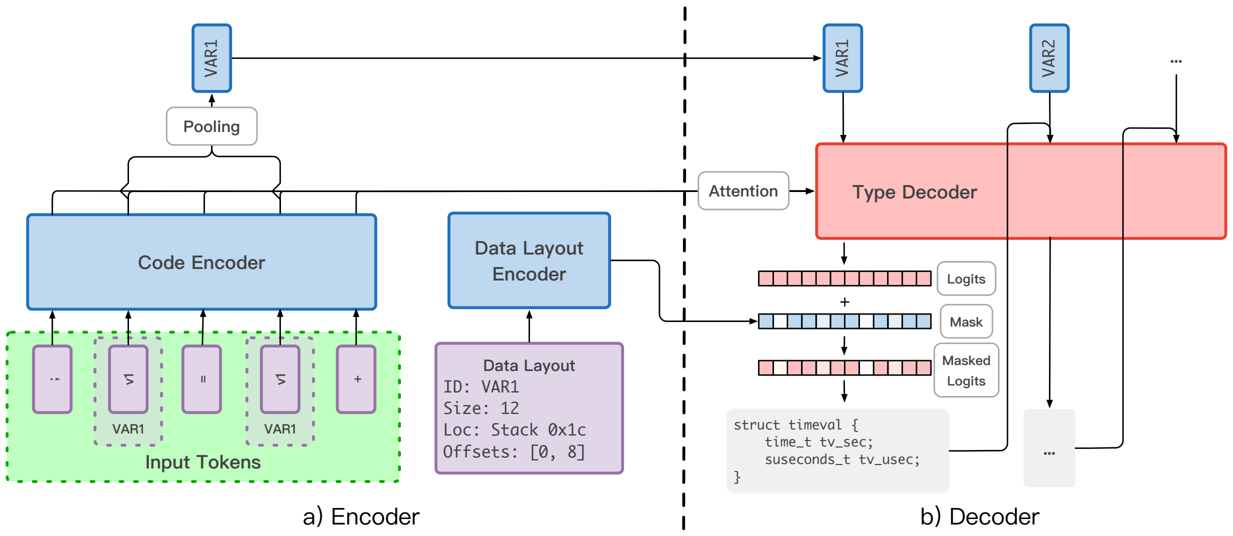


Figure 4.3: Overview of DIRTY's neural model architecture for predicting types. Decompiled code is sequentially fed into the Code Encoder. When the input of the code encoder corresponds to a specific variable (e.g., **`VAR1`**), it is pooled with other instances of the same variable to generate a single encoding for that variable. Each pooled encoding is then passed into the Type Decoder, which outputs a vector of the log-odds (logits) for predicted types. This vector is masked with a vector generated by the data layout encoder and the most probable type is chosen from the masked logits.

where the input $x = (x_1, x_2, \ldots, x_n)$ is the code token sequence of the decompiled function and the output $\mathbf{H} = (\mathbf{h_1}, \mathbf{h_2}, \ldots, \mathbf{h_n})$ is the sequence of deep contextualized representations. In this equation, f_{en} denotes the encoder, implemented with neural networks, and θ_{en} denotes the model's learnable parameters.

The ultimate goal of DIRTY is to make type predictions about each *variable* that appears in the decompiled function. However, the encoder produces hidden representations for every *code token* (e.g., "**`v1`**", "**`:`**", "**`=`**", "**`v1`**", "**`+`**", "**`1`**" are all tokens). Because a variable can appear multiple times in the code tokens of a function, we need a way to summarize all appearances of a variable. We achieve this through *pooling*, where the representation for the t-th variable[2] is computed based on all of its appearances in the code tokens, A_t, using average pooling [88]

$$\mathbf{v_t} = \text{AveragePool}_{x_i \in A_t} \mathbf{h_i}, \; t = 1, \ldots, m \tag{4.3}$$

where m is the number of variables in the function. This solution removes the burden of gathering all information about a variable throughout the function into a single token representation. The representation for the first variable, `VAR1`, is shown in the upper-left of Figure 4.3.

Type Decoder. Given the encoding of the decompiled tokens, the decoder predicts the most probable (i.e., idiomatic) types for all variables in the function. The decoder takes the encoded representations of the code tokens ($\mathbf{H}$) and identifiers ($\mathbf{v_t}$) as input and predicts the original types $\hat{y} = (\hat{y}_1, \hat{y}_2, \ldots, \hat{y}_m)$ for all m variables in the function. Unlike the encoder, the decoder predicts

[2] t is commonly used in RNN literature because it refers to a "timestep".

the output step-by-step using former predictions as input for later ones.[3]

At each time step t, the decoder tries to predict the type for the t-th variable as follows:

1. The decoder takes the code representations $\mathbf{H}$ and variable representation $\mathbf{v_t}$ from the encoder, and also previous predictions $\hat{y}_1, \hat{y}_2, \ldots, \hat{y}_{t-1}$ from itself, to compute a hidden representation $\mathbf{z_t} \in \mathbb{R}^{d\_model}$
$$\mathbf{z_t} = f_{de}(\hat{y}_1, \hat{y}_2, \ldots, \hat{y}_{t-1}, \mathbf{v_t}, \mathbf{H}; \theta_{de}) \tag{4.4}$$
where f_{de}, θ_{de} denotes the decoder and its parameters. The hidden representation $\mathbf{z_t}$ is then used for prediction.
2. The output layer of the decoder then uses its learnable weight matrix $\mathbf{W}$ and bias vector $\mathbf{b}$ to transform the hidden representation $\mathbf{z_t}$ to the *logits* for prediction
$$\mathbf{s_t} = \mathbf{W}\mathbf{z_t} + \mathbf{b}, \tag{4.5}$$
where $\mathbf{s_t} \in \mathbb{R}^{|\mathcal{T}|}$, $\mathbf{W} \in \mathbb{R}^{|\mathcal{T}| \times d\_model}$, $\mathbf{b} \in \mathbb{R}^{|\mathcal{T}|}$, and $|\mathcal{T}|$ is the number of types in the type library. The logits $\mathbf{s_t}$ is the unnormalized probability predicted by the model, or the model's *scores* on all types.
3. The $\mathrm{softmax}$ function computes a probability distribution over all possible types from $\mathbf{s_t}$
$$\Pr(\hat{y}_t | \hat{y}_1, \hat{y}_2, \ldots, \hat{y}_{t-1}, x) = \mathrm{softmax}\, \mathbf{s_t} \tag{4.6}$$

Note that the type library $\mathcal{T}$ is fixed, meaning DIRTY can only predict types that it has seen during training. We discuss this limitation, its implications, and potential mitigations in Section 4.3. However, DIRTY can recover structure types as well as normal types, as both are simply entries in $\mathcal{T}$.

The goal of the decoder is to find the *optimal* set of type predictions for all variables in a given function (i.e., the predictions with the highest combined probability): $\mathrm{argmax}_{\hat{y}} \Pr(\hat{y}|x)$. This probability can be factorized as the product of probabilities at each step:
$$\Pr(\hat{y} \mid x) = \prod_{t=1}^{m} \Pr\left(\hat{y}_t \mid \hat{y}_1, \hat{y}_2, \ldots \hat{y}_{t-1}, x\right). \tag{4.7}$$
We can compute $\Pr\left(\hat{y}_t \mid \hat{y}_1, \hat{y}_2, \ldots \hat{y}_{t-1}, x\right)$, but finding the optimal $\hat{y} = (\hat{y}_1, \hat{y}_2, \ldots, \hat{y}_m)$ is not an easy task, because each variable can have $|\mathcal{T}|$ possible predictions, and each prediction affects subsequent predictions. The time complexity of exhaustive search is $\mathcal{O}(|\mathcal{T}|^m)$. Therefore, finding the optimal prediction is often computationally infeasible for large functions. A simple approach is *greedy decoding*, selecting the most promising prediction at every step based on the previously selected predictions, i.e., taking the max $\hat{y}_t = \mathrm{argmax}_{y_t} Pr\left(\hat{y}_t = i \mid \hat{y}_1, \hat{y}_2, \ldots, \hat{y}_{t-1}, x\right)$. Greedy decoding is fast, but it often finds subpar predictions.

We use beam search [127], a compromise between greedy decoding and an exhaustive search. Rather than only taking the most promising prediction (greedy), beam search considers a configurable number of most promising predictions at each step. In practice, it is usually able to find good (but not optimal) predictions, but is significantly faster than an exhaustive search.

[3]This is known as an *autoregressive* model.

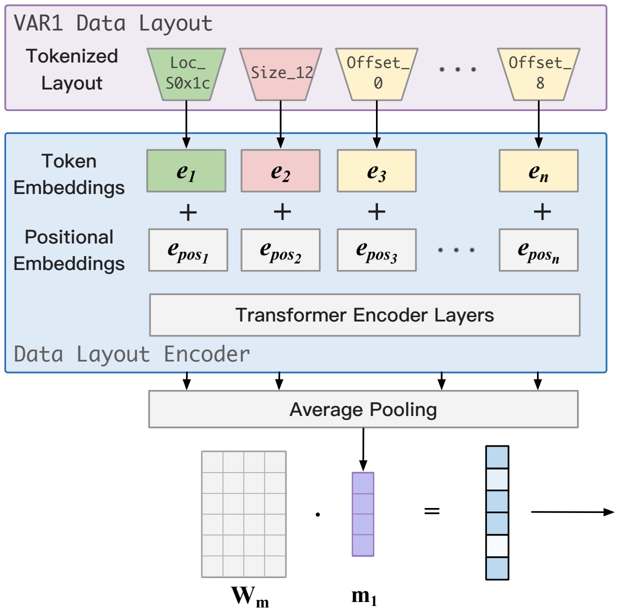


Figure 4.4: The data layout encoder of DIRTY. The data layout for a specific variable, including its location, size, and offsets of its members is passed into the layout encoder (top), generating a mask (bottom).

4.1.4 Data Layout Encoder

The model described so far only uses information encoded into the code tokens of the decompiled representation. But to actually *create* such an output, decompilers typically perform a battery of complex binary analyses. Some decompilers allow the user to programmatically access the interim results from some of these analyses. In particular, Hex-Rays provides information about the storage location (e.g., register or stack offset), size, nested data types (e.g., if the variable is a `struct`), and offsets of its members, if any (e.g., offsets in an array or of fields in a `struct`), for each variable in a function. Intuitively, this information can help DIRTY rule out bad predictions. For instance, a variable that is 4 bytes long could not be a `char` type because it would not fit.

A naïve approach would be to use this information as a hard constraint on the decoder's predictions, i.e., a *mask* which sets the probability of any "incompatible" types to 0. However, this runs into a problem when the decompiler incorrectly reconstructs the data layout (see Figure 4.2). To mitigate this, DIRTY learns a *soft* mask, reducing probabilities without setting them to 0. For example, DIRTY can learn based on many observations that a decompiled `char[4]` should be typed as a `char[3]` 5% of the time and `char[4]` 80%, and adjust the predictions of the type decoder accordingly. This allows the model to learn how best to incorporate the data layout information from the decompiler, including when the information is likely to be incorrect. Figure 4.3 illustrates where the data layout encoder fits into the overall architecture.

To implement the soft mask encoder, we jointly train another Transformer encoder to use data layout information to generate a mask. Figure 4.4 shows the internals of the data layout encoder. First, variable data layout is passed to the encoder. There are three parts to the data layout for a specific variable, each of which is simply converted to a token:

- *Location.* A variable can be located either in registers (tokenized as `[Loc_<Register Name>]`) or on the stack (`[Loc_S<Offset>]`). E.g., a variable stored 28 bytes below the stack pointer is tokenized as `[Loc_S0x1c]`.
- *Size.* Measured in bytes and tokenized as `[Size_<Size>]`.
- *Internal Offsets.* The offsets of members of the type (either array elements or struct fields), in bytes. E.g., the type **`int[2]`** would have the offsets $\{0, 4\}$, while a **`struct`** with two **`char`** fields would have the offsets $\{0, 1\}$. These are tokenized as a sequence of `[Offset_<Offset>]`. For consistency, we also use `[Offset_0]` for types without substructure (i.e., scalar types like **`int`**).

The tokenized data layout information is concatenated into a sequence denoted M_t and then encoded as

$$\mathbf{m}_t = f_{layout}\left(M_t; \theta_{layout}\right), \tag{4.8}$$

where $\mathbf{m}_t$ is the hidden representation of data layout information. Inspired by Michel and Neubig [121], we adjust the output type distribution with data layout information. Formally, we modify Equation (4.5) to fuse the data layout representation $\mathbf{m}_t$ into the final output layer:

$$\tilde{\mathbf{s}}_t = \mathbf{s}_t + \mathbf{W}_m\mathbf{m}_t = \mathbf{W}\mathbf{z}_t + \mathbf{W}_m\mathbf{m}_t + \mathbf{b}, \tag{4.9}$$

where $\mathbf{s}_t$ is the logits predicted by the Type Decoder, $\mathbf{W}_m\mathbf{m}_t$ is the "soft mask" produced by the data layout encoder, and $\tilde{\mathbf{s}}_t$ is the new masked logits. $\mathbf{W}_m \in \mathbb{R}^{|\mathcal{T}| \times d\_model}$ denotes the learnable weight matrix in the final layer of data layout encoder for transforming the data layout representation $\mathbf{m}_t \in \mathbb{R}^{d\_model}$ to the mask $\in \mathbb{R}^{|\mathcal{T}|}$. This implements a soft filter for type prediction using data layout information.

4.1.5 Multi-Task

Many variable names are indicative of their type. For example, **`i`** and **`j`** are often used to represent integers, **`s`** and **`str`** are often used to represent strings, etc. Intuitively, there is some connection between a variable's *name* and its *type*. Indeed, measuring the adjusted mutual information [162] between variable names and types in our dataset, we find a moderate association (0.41 on the scale $[0, 1]$). Since variable *names* can often be recovered from decompiled code using neural models [88], this may help us learn to predict variable types as well (and vice versa).

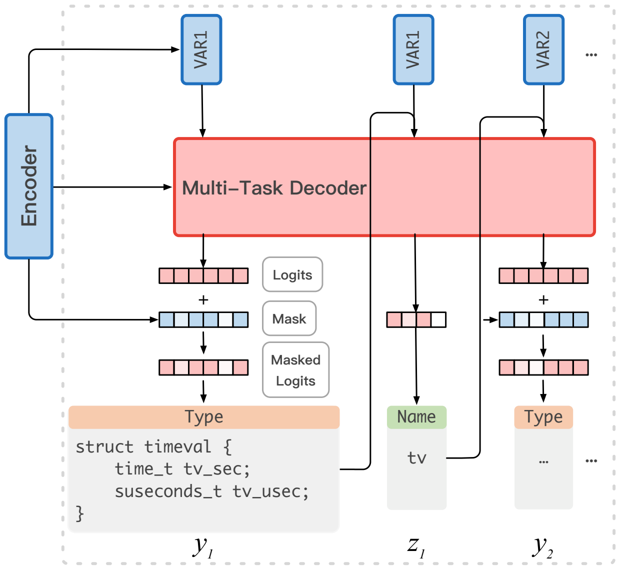
To test this, we extend DIRTY to also predict names with a single, integrated multi-task model. That is, we also predict a variable name for each variable in the function

$$\hat{z} = (\hat{z}_1, \hat{z}_2, \ldots, \hat{z}_m) \tag{4.10}$$

where $\hat{z}_t$ denotes the predicted name for the t-th variable.

DIRTY's decoder outputs are interleaved to predict names and types in parallel (Figure 4.5). The first time the decoder is invoked on the t-th variable, it outputs the predicted *type* ($\hat{y}_t$) and the second time it outputs a predicted *name* ($\hat{z}_t$).

The training and prediction procedures remain almost the same, with two notable exceptions. First, to improve performance, the data layout encoder is *not* activated when the decoder is predicting a variable's name. This is unnecessary because name prediction depends on the predicted type, which has already incorporated the data layout information. Preliminary experiments confirmed no improvement in accuracy when using the data layout encoder for name prediction.



```
struct timeval {
    time_t tv_sec;
    suseconds_t tv_usec;
}
```

Figure 4.5: The multi-task decoder for DIRTY, which predicts both variable types and names. The encoder architecture is the same as in Figure 4.3. Each variable is passed to the decoder twice, the first time a type is predicted (y_i), and the second time a name is predicted (z_i). Note that the data layout encoding of a variable is only used to weight type predictions.

Second, there are two ways to interleave the predictions of types and names: types first or names first. In theory, this does not matter because they are equivalent if the learned model and the decoding algorithm are ideal. In practice, we chose to predict types first because we believe the type prediction task should be easier (since there is more information) and it better reflects how developers define variables.

4.2 Evaluation

Our evaluation was structured to answer the following research questions:

- **RQ1**: How effective is DIRTY at idiomatic retyping?
- **RQ2**: How does DIRTY compare to existing work on other decompilation benchmarks?
- **RQ3**: How does each component of DIRTY contribute to its performance?
- **RQ4**: How does compiler optimization affect DIRTY's prediction accuracy?

4.2.1 Experimental Setup

First, we introduce the DIRT dataset we used for training DIRTY, and experimental setup details.

Dataset for Idiomatic ReTyping (DIRT). To create DIRT, we queried a 2017 version of the GHTORRENT[4] database, compiling a list of public GITHUB repositories predominantly written

[4]https://ghtorrent.org

in C. We then cloned these repositories locally using an open-source tool, GHCC,[5] to automatically build them. GHCC identifies build instructions (e.g., Makefiles) in repositories, creates a Docker container with the requisite libraries, and attempts to build the project. We used GCC version 9.2.0. For most experiments, we explicitly disable optimizations using the `-O0` compiler flag. We also evaluated DIRTY at higher optimization levels in Section 4.2.5. This process resulted in 4,346,134 automatically compiled 64-bit x86 binaries. After compilation, we then decompiled each binary using Hex-Rays and filtered out any functions that did not have variables requiring renaming or retyping. Following our earlier work in DIRE Chapter 3, we compiled each binary again with debugging information to align decompiler-assigned variable names (e.g., **`v1`**) and developer-assigned variable names (e.g., **`picture`**) to form training examples.

Since DIRE was only concerned with renaming, the previous dataset did not include variables which did not correspond to a named variable in the original source code. Many such variables are actually caused by mistakes in the decompiler during type recovery, for instance decompiling a structure to multiple scalar variables instead. Since the goal of DIRT is to enable type recovery and fix such mistakes, we label these instances as **`<Component>`** to denote that they are *components* of a variable in the source code. This allows the model to combine them with other variables into an array or a struct.

The final DIRT dataset consists of 75,656 binaries randomly sampled from the full set of 4,346,134 binaries to yield a dataset that we could fully process based on the computational resources we had available. We split the dataset per-binary as opposed to per-function, which ensures that different functions from the same binary cannot be in both the test and training sets. The training dataset consists of 997,632 decompiled functions, and a total number of 48,888 different types. We also preprocess the decompiled code with byte-pair encoding (BPE) [148], a widely adopted technique in NLP tasks to represent rare words with limited vocabulary by tokenizing them into subword units. After this step, the DIRT dataset consists of 368 *million* decompiled code tokens, and an average of 220.3 tokens per function.

Metrics. We evaluate DIRTY using two metrics:

- *Name Match*: We consider a variable name prediction correct if it exactly string matches the name assigned by the original developer. We compute the prediction accuracy as the average percentage of correct predictions across all functions in the test set.
- *Type Match*: A type prediction is considered correct only if the predicted type fully matches the ground truth type, including data layout, and the type and name of any fields if applicable. We serialize types to strings and use string matching to determine type matching.

Note that both metrics are conservative. Predictions may still be meaningful, even if not identical to the original names. A human study evaluating the quality of predicted types and names is described in Chapter 5.

Meaningful Subsets of the Test Data. We introduce several subsets of the DIRT test set to better interpret the results:

[5]`https://github.com/huzecong/ghcc`

Table 4.1: DIRTY has higher retyping accuracy than Frequency By Size (F_{Size}) and Hex-Rays (HR) on the DIRT dataset, both for all types (All) and on structural types alone (Struct).

	Overall		In Train		Not in Train	
Method	All	Struct	All	Struct	All	Struct
F_{Size}	23.6	9.7	23.5	9.1	23.8	10.4
HR	37.9	28.7	39.0	28.7	36.4	28.7
DIRTY	**75.8**	**68.6**	**89.9**	**79.2**	**56.4**	**54.6**

- *Function in training* vs *Function not in training*: As in our previous work, *Function in training* consists of the functions in the test set that also appear in the training set, which are mainly library functions. Allowing this duplication simulates the realistic use case of analyzing a new binary that uses well-known libraries. We also separately measure the cases where the function is not known during training (i.e., *Function not in training*) to measure the model's generalizability.
- *Structure types*: Only 1.8% of variables in DIRT have structure types. Because of this low percentage, examining overall accuracy may not reflect DIRTY's accuracy when predicting structure types, which we have found anecdotally to be more challenging. To mitigate this, we separately measure DIRTY's accuracy on structures in addition to its overall accuracy.

4.2.2 RQ1: Overall Effectiveness

We evaluate DIRTY on the retyping task and report its accuracy compared to several baselines.

Baselines. We measure accuracy with respect to two baselines for predicting variable types:

- *Frequency by Size*: The number of bytes a variable occupies is the most basic information for a type. For this technique, we predict the most common developer-assigned type for a given size (as reported by the decompiler). E.g., `int` is the most common 4-byte type, and `__int64` is the most common 8-byte type; this baseline simply assigns these types to variables of the respective size.
- *Hex-Rays* [71]: During decompilation, Hex-Rays predicts a type for each variable: we use these predictions as a baseline. However, Hex-Rays cannot predict developer-generated types without prior knowledge of them, e.g., Hex-Rays assigns `unsigned __int16` instead of the more common `uint16_t`, which puts it at an unfair disadvantage. For this baseline, we reassign the type chosen by Hex-Rays to the most common developer-chosen name associated with it (e.g., we replace every `unsigned __int16` with `uint16_t`.

Results. As shown in Table 4.1, DIRTY can correctly recover 75.8% of the original (developer-written) types from the decompiled code. In contrast, Hex-Rays, the highest scoring baseline, can only recover 37.9% of the original types.

As expected, DIRTY performs even better when it has seen a particular function before (In Train), generating the same type as the developer 89.9% of the time. This indicates that DIRTY

Table 4.2: Example variable types from the *Function not in training* testing partition. The top rows are the developer-assigned types and the columns show DIRTY's top-5 most frequent predictions. `<Component>` represents a prediction that the variable in the decompiled code does not correspond to a variable in the source code (e.g., because it corresponds to a member of a struct).

`int`		`char *`		`class std::string`	
`int`	88.8%	`char *`	60.3%	`class std::string`	47.5%
`unsigned int`	4.3%	`const char *`	11.4%	`char[32]`	24.2%
`<Component>`	2.7%	`<Component>`	4.4%	`char[47]`	14.6%
`uint32_t`	0.8%	`__int64`	4.1%	`class std::__cxx11::basic_string`	6.1%
`u_int32_t`	0.3%	`size_t`	1.8%	`char[40]`	3.5%

works particularly well on common code such as libraries. Even when a function has never been seen (Not in Train), DIRTY predicts the correct type 56.4% of the time.

Table 4.1 also shows the performance of DIRTY on structure types alone. Correctly predicting structure types is more difficult than predicting scalar types, and all models show a drop in performance. Despite this drop, DIRTY still achieves 68.6% accuracy overall, and 54.6% accuracy on the *Function not in training* category. Frequency By Size struggles on structures with only 9.7% accuracy; this is expected since structures of a given size can have many possible types. Hex-Rays is slightly more accurate at 28.7%, as the decompiler is able to analyze the layout of structures.

Table 4.2 shows several examples of retyping predictions from the *Function not in training* partition. These examples show that accuracy is not the full story; even when DIRTY is unable to predict the correct type, the differences are often minor (e.g., `unsigned int` v. `int`, and `const char *` v. `char *`). The bottom half of Table 4.2 shows prediction examples of structure types. DIRTY is able to recover the actual structure much of the time. At other times, DIRTY also produces some semantically reasonable but syntactically unacceptable predictions, like `char [32]` for `class std::string`.

RQ1 Answer: We find that DIRTY is effective at idiomatic retyping, correctly recovering 75.8% of the original types in decompiled code. When DIRTY has seen a particular function before, this performance raises to 89.9%.

4.2.3 RQ2: Comparison with Prior Work

We further compare DIRTY with recent work on type recovery [185] and our previous work on variable name recovery.

Type Recovery. While there is prior work on type recovery (see also Chapter 2), none of the existing approaches, TIE [96], Howard [152], Retypd [130], TypeMiner [119] and OSPREY [185], are publicly available. We are grateful to Zhang et al. [185], the authors of OSPREY, for kindly sharing their evaluation material so we could compare results.

Table 4.3: Accuracy comparison on the Coreutils benchmark.

	Coreutils			
Model	All	Visited	Non-Visited	Struct
OSPREY	71.6	**83.8**	32.4	26.6
DIRTY	76.8	79.1	69.6	15.7
DIRTY$_{Light}$	**80.1**	80.1	**80.1**	**27.7**

OSPREY is a recently proposed probabilistic technique for variable and structure recovery that outperforms existing work including Howard [152], Angr [151], Hex-Rays [71] and Ghidra [185]. The OSPREY authors provided us with the GNU coreutils[6] executables they used in their evaluation, which were compiled with `-O0` to disable optimization. We ran DIRTY on these executables, but only evaluated on stack and heap variables, since OSPREY does not recover register variables. This benchmark consists of 101 binaries and 17,089 variables. We also define two subsets of the dataset:

- *Visited.* A subset of 13,020 variables that are covered by BDA [187], a binary abstract interpretation tool that OSPREY relies on. OSPREY is expected to perform better on these covered functions than uncovered functions, which we also report as *Non-Visited*.[7] However, DIRTY is not subject to this limitation.
- *Struct.* A subset of 3,061 variables related to structure types. Following OSPREY, we include structs allocated on the stack, pointers to structs on the heap, and arrays of structs. These variables do *not* have to be in the Visited subset.

Because DIRTY can predict up to 48,888 different types, each including the full syntactic and semantic information, we convert its predictions in a post-hoc manner to make it comparable with OSPREY.[8]

Table 4.3 compares the accuracies of both systems. On the overall coreutils benchmark, DIRTY slightly outperforms OSPREY (76.8% vs 71.6%). OSPREY outperforms DIRTY on the Visited subset, but as expected, performs worse on the Non-Visited functions. Meanwhile, DIRTY is more consistent on Visited and Non-Visited. When only looking at structure types, OSPREY outperforms DIRTY (26.6% vs 15.7%).

However, this comparison puts DIRTY at a disadvantage, since OSPREY was designed for this task of recovering syntactic types, while DIRTY was trained to recover variable and type/-field names, and much of this information is thrown out for this evaluation. To address this, we trained a new model, DIRTY$_{Light}$, on DIRT, but tailored the training to OSPREY's simplified task. The accuracy of this model is also reported in Table 4.3. As expected, the DIRTY$_{Light}$ model outperforms the off-the-shelf DIRTY model, since it is trained specifically for this task. DIRTY$_{Light}$

[6]`https://www.gnu.org/software/coreutils/`

[7]A majority of uncovered functions are unreachable from the entry point of the binary, and others are indirect call targets which BDA fails to analyze.

[8]Specifically, we discard type names and field names. For example, **`bool`** and **`char`** are both converted to **`Primitive_1`**, which stands for a primitive type occupying 1 byte of memory, **`const char *`** and **`char *`** are converted to **`Pointer<Primitive_1>`**, and **`struct ImVec2 {float x; float y;}`** converted to **`Struct<Primitive_4, Primitive_4>`**.

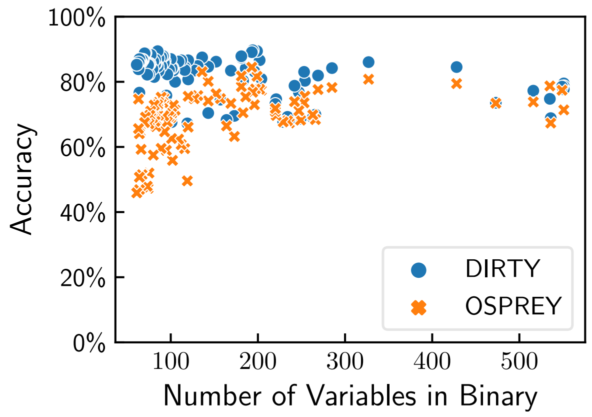


Figure 4.6: Accuracy of DIRTY and OSPREY on 101 individual programs in the coreutils benchmark with different number of variables. The two methods are competitive on large binaries, while DIRTY performs much better on small binaries.

greatly improves prediction accuracy on the Struct subset, even outperforming OSPREY.

To further get a fine-grained comparison with OSPREY, we calculate accuracy on 101 coreutils binaries individually, and show the prediction accuracies of DIRTY and OSPREY with respect to the number of variables in the programs in Figure 4.6.

We observe that DIRTY is competitive compared with OSPREY. Interestingly, while the results on large binaries are close, DIRTY performs better on small binaries. This suggests our learning-based method trained on GITHUB data might generalize better on rare patterns compared to empirical methods that might have been developed based on observations on a limited number of common and relatively larger programs.

In addition, DIRTY is also much faster and scalable. On average, OSPREY takes around 10 minutes to analyze one binary in coreutils, while it takes 75 seconds for DIRTY$_{Light}$ to finish inference on the whole coreutils benchmark.

Overall, we believe both methods are valuable. Since at this point DIRTY is using Hex-Rays recovered data layout as input to its data layout encoder, we believe a promising future direction is to combine these two methods—using OSPREY's results as the input to DIRTY's, and the combined approach can potentially achieve even better results.

Name Recovery. At the time of these experiments, Decompiled Identifier Renaming Engine (DIRE) was the state-of-the-art neural approach for decompiled variable name recovery. Recall that the DIRE model consists of both a lexical encoder and a structural encoder, utilizing both tokenized decompiled code and the reconstructed abstract syntax tree (AST). In contrast, DIRTY's simpler encoder only uses the tokenized decompiled code.

To fairly compare with DIRE, we trained DIRTY on the DIRE dataset and also trained DIRE on the DIRT dataset. Note that since DIRE is focused on variable renaming, there is no type information collected in the DIRE dataset and we cannot use the data layout encoder for these experiments. Instead, we only use our Code Encoder and Renaming Decoder. The accuracy of both systems is shown in Table 4.4. DIRTY significantly outperforms DIRE in overall accuracy

Table 4.4: Accuracy comparison of DIRE and DIRTY on the DIRE and DIRT datasets. Accuracy is reported overall (All), when functions are in the training set (FIT), and when functions are not in the training set (FNIT).

	DIRE Dataset			DIRT Dataset		
Model	All	FIT	FNIT	All	FIT	FNIT
DIRE	72.8	84.1	33.5	57.5	75.6	31.8
DIRTY	**81.4**	**92.6**	**42.8**	**66.4**	**87.1**	**36.9**

Table 4.5: Effect of model size. The accuracy columns show the overall and struct type accuracy.

	Accuracy	
Model	Overall	Struct
DIRTY$_S$	74.5	65.4
DIRTY	**75.8**	**68.6**

on both the DIRE (81.4% vs. 72.8%), and DIRT datasets (66.4% vs. 57.5%). DIRTY also generalizes better than DIRE: when functions are not in the training set, DIRTY outperforms DIRE on both the DIRE (42.8% vs. 33.5%) and the DIRT datasets (36.9% vs. 31.8%).

DIRTY outperforms DIRE in spite of the fact that it only leverages the decompiled code, whereas DIRE leverages both the decompiled code *and* the reconstructed AST from Hex-Rays. Since the primary difference between DIRTY without type prediction and DIRE is that it uses Transformer as its encoder and decoder network, we attribute this improvement to the power of Transformers, which allow modeling interactions between any pair of tokens, unrestricted to a sequential or tree structure as in DIRE.

DIRTY also trains faster than DIRE. We found that DIRTY surpassed DIRE in accuracy after training for 30 GPU hours, compared to the 200 GPU hours required to train DIRE on the full DIRT dataset, which we again attribute to the efficiency of the Transformer architecture.

RQ2 Answer: DIRTY is effective at recovering structural types, slightly outperforming OSPREY when its output is converted into a comparable form. DIRTY also outperforms DIRE on renaming tasks, leveraging the additional information encoded in types.

4.2.4 RQ3: Ablation Study

To understand how each component of DIRTY contributes to its overall performance, we perform an ablation study.

Model Size. Transformers have the merit of scaling easily to larger representational power by stacking more layers, increasing the number of hidden units and attention heads per layer [44, 161]. We compare DIRTY to a modified, smaller version DIRTY$_S$. DIRTY contains 167M

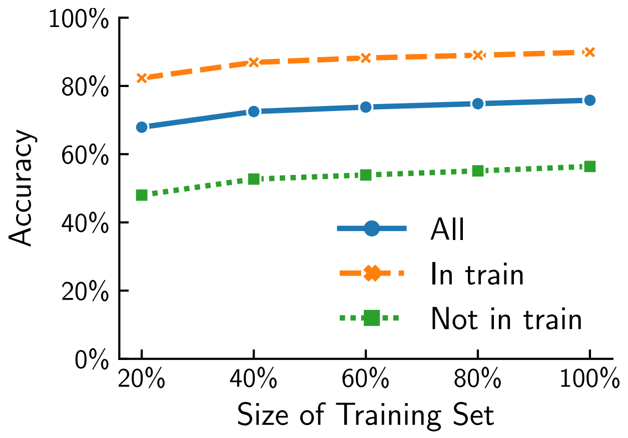


Figure 4.7: Effect of training data size. With 100% of the data, the accuracies of *All*, *In train*, and *Not in train* are 75.8%, 89.9%, and 56.4% respectively. With 20%, these drop to 67.9%, 82.3%, and 48.0% respectively.

parameters, while DIRTY$_S$ only 40M.

Table 4.5 shows overall DIRTY is 75.8% accurate vs. 74.5% for DIRTY$_S$'s. This indicates increasing the model size has a positive effect on retyping performance. The gain from increased model capacity is notably larger when comparing performance on structures. This improvement suggests that complex types are more challenging and require a model with larger representational capacity. We are not able to train a larger model due to limits on computation power.

Dataset Size. We examine the impact of training data size on prediction accuracy. As a data-driven approach, DIRTY relies on a large-scale code dataset; studying the impact of data size gives us insight into the amount of data to collect. We trained DIRTY on 20%, 40%, 60%, 80% and 100% portion of the full training partition and report the results in Figure 4.7.

Figure 4.7 shows the change in accuracy with respect to the percentage of training data. Increasing the size of training data has a significant effect on the accuracy. Between 20% and 100% of the full size the accuracy increases from 67.9% to 75.8%, a relative gain of 11.6%.

Notably, accuracy on *Function not in training* has a relative gain of 17.5% much larger than on the *Function in training* partition. This is likely because the *Function in training* partition contains common library functions shared by programs both in the training and test set, and even a smaller dataset will have programs that use these functions. In contrast, the *Function not in training* part is open-ended and diverse.

It is also worth noting that the accuracy drops sharply when the training set size is decreased from 40% to 20%, justifying the necessity for using a large-scale dataset.

Data Layout Encoder. We explore the impact of the data layout encoder on DIRTY's performance. We experiment with a new model with no data layout encoder, DIRTY$_{NDL}$.

Table 4.6 shows the accuracy results overall and on the *Function in training* and *Function not in training* partitions. The inclusion of the data layout encoder improves overall accuracy from 72.2% to 75.8%, indicating that the data layout encoder is effective. The results are even more

Table 4.6: Effect of the data layout encoder on the accuracy of DIRTY. Accuracy is reported for the model with (DIRTY) and without (DIRTY_{NDL}) the encoder.

Model	Overall	In train	Not in train
DIRTY_{NDL}	72.2	88.4	49.9
DIRTY	**75.8**	**89.9**	**56.4**

Table 4.7: Comparative examples from DIRTY with and without data layout encoder from the *Function not in training* partition. Predictions inside a gray box have a different data layout than the ground truth type. DIRTY effectively suppresses these, which helps guide the model to a correct prediction. The structure's full type is `struct __m128d {double[2] m128d_f64;}`.

DIRTY				DIRTY_{NDL}			
`__int64`		`struct __m128d`		`__int64`		`struct __m128d`	
`__int64`	74.3%	`struct __m128d`	78.7%	`__int64`	67.0%	`double`	33.1%
`<Component>`	5.7%	`<Component>`	15.4%	`int`	6.3%	`<Component>`	27.2%
`void *`	1.7%	`void`	2.9%	`<Component>`	6.0%	`__int64`	10.3%
`char *`	1.7%	`__int128`	2.2%	`unsigned int`	1.5%	`struct __m128d`	5.9%
`const char *`	1.6%	`double`	0.7%	`char *`	1.2%	`int`	3.7%

interesting when the results are broken into the two partitions. The relative gain on the *Function in not training* partition is 13% (49.9% to 56.4%), compared to 1.7% on the *Function in training* partition (88.8% to 89.9%). This suggests the data layout encoder greatly improves DIRTY's generalization ability.

Table 4.7 compares example predictions from DIRTY and DIRTY_{NDL} on the same types from the *Function not in training* partition. For the `__int64` example, the type predictions from DIRTY mostly have the correct size of 8 bytes. DIRTY_{NDL}, however, often incorrectly predicts `int` and `unsigned int`. This is understandable because in situations where the value doesn't exceed the 32-bit integer, `__int64` can be safely interchanged with `int`, these situations can be identified in some decompiled code. However, apart from the correctness of the retyped program, accuracy to the original binary, (i.e., allocating 8 bytes instead of 4), is also important. DIRTY achieves this better than DIRTY_{NDL}.

In the second example, the `struct __m128d` type occupies 16 bytes, and has two members at offset 0 and 8. DIRTY_{NDL} mainly mistakes this structure as a `double`, which might make sense semantically but is unacceptable syntactically. With the data layout encoder, DIRTY effectively reduces these errors. This demonstrates this component achieves the soft masking effect on type prediction as intended in Section 4.1.4.

Multi-Task Decoder. In this section we study the effectiveness of the Multi-Task decoder when compared to decoders designed for only retyping or only renaming. Inspecting the accuracy numbers reported in Table 4.8, the Multi-Task decoder has similar, but slightly lower overall accuracy on both tasks as the two specialized models (-0.8% for retyping and -1.3% for renaming). One

Table 4.8: Performance comparison of the Retyping-only, Renaming-only, and Multi-Task decoders. Overall performance is shown, in addition to performance on retyping when the name is correct (✓Name) and performance on renaming when the type is correct (✓Type).

	Retyping		Renaming	
Model	Overall	✓Name	Overall	✓Type
Retyping	**75.8**	90.6	-	-
Renaming	-	-	**66.4**	82.6
Multi-Task	74.9	**92.3**	65.1	**84.6**

possible reason is that the Multi-Task model has twice the length of decoding lengths than a specialized model, which makes greedy decoding harder.

Despite the small decrease in performance, the unified model has advantages. These are illustrated in the ✓Name and ✓Type columns of Table 4.8. ✓Name and ✓Type stand for the subsets of the full dataset where the Multi-Task decoder makes correct renaming predictions and correct retyping predictions, and we evaluate the retyping and renaming performance on them, respectively.[9] The Multi-Task decoder outperforms the specialized models by 1.9% and 2.4% relatively on these metrics, in spite of the longer decoding length. This means the type and name predictions from the Multi-Task decoder are more consistent with each other than from specialized models. In other words, making a correct prediction on one task increases the probability of success on the other task.

In practice, this offers additional flexibility and opens the opportunity for more applications. For example, consider a cooperative setting where a human decompilation expert uses DIRTY as an analysis tool. The human expert may be unsatisfied with the model's top prediction and want to switch to another one in the top-k candidates list. With a Multi-Task decoder, the model adjusts the name prediction for that variable, which is impossible with the specialized decoders.

RQ4 Answer: We find that each component of DIRTY contributes to its accuracy. The addition of the data layout encoder improves overall accuracy from 72.2% to 75.8%. Additionally, the Multi-Task Decoder enables the simultaneous prediction of both names and types for variables in decompiled code.

4.2.5 RQ4: Compiler Optimization Levels

We study the impact of compiler optimizations on DIRTY's accuracy. In keeping with the spirit of the OSPREY evaluation on coreutils compiled with `-O3`, we choose coreutils as our evaluation dataset. However, since we did not have access to the original dataset used by OSPREY except `-O0`, we recompiled GNU coreutils 3.2 ourselves using optimization levels `-O0`, `-O1`, `-O2`, and `-O3`. Table 4.9 shows how accurately DIRTY is able to recover the full type (including type and field names) informaition at each optimization level. As expected, DIRTY does best at `-O0`,

[9]The probability of success on the other task also increases by chance, because success on one task implies it is easier than average. We have eliminated this influence by, e.g., comparing 92.3 to 90.6, instead of 74.9.

Table 4.9: Accuracy comparison of DIRTY on the GNU coreutils benchmark compiled with `-O0`, `-O1`, `-O2`, and `-O3` optimization levels.

	GNU coreutils			
Model	`-O0`	`-O1`	`-O2`	`-O3`
DIRTY	48.20	46.01	46.04	46.00

since DIRTY is trained on `-O0` code and we believe `-O0` code to be simpler. Going from `-O0` to `-O1`, DIRTY's accuracy drops from 48.2% to 46.0%. However, there is little difference in performance between `-O1`, `-O2`, and `-O3`. This suggests that DIRTY does slightly better on the optimization level of code it was trained on, but that the effect of optimizations is small. We believe this is because Hex-Rays recognizes and will "undo" some optimizations so that the decompiled code will be very similar. For example, unoptimized code will often reference stack variables using a frame pointer, but optimized code will reference such variables using the stack pointer, or even maintain them in a register. But both implementations will look similar in the decompiled code, since the mechanism used to reference the variable is not important at the C level. Since DIRTY operates on the decompiled code, the decompiler effectively insulates DIRTY from these optimizations.

RQ4 Answer: We find that different optimization levels have a minimal impact on the accuracy of DIRTY: the reduction in performance between binaries compiled at `-O0` and `-O3` is only 2.2%.

4.2.6 Illustration

To gain more qualitative insights into DIRTY's predictions, consider the example in Figure 4.8. The code shown is the Hex-Rays output, cleaned for presentation. Here, we would like to rename and retype the arguments **`a1`**, **`a2`**, and **`a3`**, in addition to the variable **`v1`**. The table in Figure 4.8 shows the developer's chosen types and names together with DIRTY's suggestions. DIRTY suggests the same types and names as the developer for **`a3`** and **`v1`**, and the same type but a different name for **`a2`**. Although the names disagree for **`a2`**, we note that **`pic`** is an abbreviation for **`picture`**, so the disagreement is minor. We also observe that **`Picture_0 *`**, the type of **`a2`** itself carries a lot of semantic information; even if DIRTY was unable to suggest a meaningful name, **`Picture_0 *a2`** is still helpful for reverse engineering.

The developer and DIRTY disagree on both the name and the type of **`a1`**. In this case, the name chosen by DIRTY (**`s`**) would probably not be considered a very useful improvement over **`a1`**. However, the type suggested by DIRTY (**`MpegEncContext_0 *`**) could still be quite useful to a reverse engineer, even if it is inaccurate. It suggests that this argument is a "context", and hints that this function is used for video.

```c
int find_unused_picture(int a1, int a2, int a3) {
    int i, j, v1;
    if (a3) {
        for (i = <Num>;; ++i) {
        if (i > <Num>)
            goto LABEL_13;
        if (!*(*(<Num> * i + a2) + <Num>))
            break;
        }
        v1 = i;
    } else {
        for (j = <Num>;; ++j) {
        if (j > <Num>) {
        LABEL_13:
            av_log(a1, <Num>, <Str>);
            abort();
        }
        if (pic_is_unused(<Num> * j + a2))
            break;
        }
        v1 = j;
    }
    return v1;
}
```

ID	Developer	DIRTY
a1	AVCodecContext_0 *avctx	MpegEncContext_0 *s
a2	Picture_0 *picture	Picture_0 *pic
a3	int shared	int shared
v1	int result	int result

Figure 4.8: Simplified Hex-Rays output. <Num> and <Str> are placeholder tokens for constant numbers and strings respectively. The table summarizes the original developer names and types along with the names and types predicted by DIRTY.

4.3 Discussion

In this chapter I presented DIRTY, a novel deep learning-based technique for predicting variable types and names in decompiled code. Still, DIRTY is limited in several ways that provide key opportunities for future improvements.

Alternative Decompilers to Hex-Rays. We implement DIRTY on top of the Hex-Rays decompiler because of its positive reputation and the programmatic access it affords to decompiler internals. However, DIRTY is not fundamentally specific to Hex-Rays, and the technique should conceptually work with any decompiler that names variables using DWARF debug symbols. Note that, due to its recent popularity and promise, we attempted to evaluate our techniques using the newer, open-source Ghidra decompiler. Unfortunately, it is currently infeasible, because Ghidra routinely failed to accurately name stack variables based on DWARF. This appears to be a combination of specific issues[10] and the general design of the decompiler. Ghidra's decompiler consists of many *passes* which modify and augment the current decompilation. Some of these passes combine variables, but in doing so may combine a DWARF-named variable with others. Since the combined variable no longer corresponds directly with the DWARF variable information, Ghidra discards the name. We are optimistic, however, that when the above-mentioned issues are addressed, Ghidra may again be a reasonable target for our approach.

Generalizing to Unseen Types. A limitation of DIRTY's current decoder is that it can only predict types seen during training. Fortunately, there appears, empirically, to be sufficient redundancy across large corpora that DIRTY is still frequently able to successfully recover structural types. This lends credence to the hypothesis that code is *natural*, an observation that has been explored in several domains [43, 72]. It moreover appears that data layout is of particular importance here: layout information recovered from the decompiler impose key constraints on the overall prediction problem. Indeed, our results in Section 4.2.4 corroborate the intuition that the data layout encoder is especially important for succeeding on previously unseen code.

We envision meaningful future opportunities to more directly expand DIRTY's capabilities to predict unseen structures. This problem is analogous machine translation models that must deal with rare or compound words (e.g., xenophobia) that are not present in their dictionary. Byte Pair Encoding [148] (BPE) is the most frequently used technique to tackle this problem in the natural language domain. It automatically splits words into multiple tokens that are present in the dictionary (e.g., xeno and ##phobia). (The ## indicates the word is still part of the current word, instead of a new word next to it.) This technique greatly increases the number of words a model can handle despite a limited dictionary size, and enables the composition of new words that were not seen during training. This suggests that we can similarly extend DIRTY's decoder to predict previously unseen types by decomposing structure types into multiple pieces with BPE. For example, a structure type `struct timeval {time_t tv_sec; suseconds_t tv_usec;}` is split into four separate tokens, which are 1) `struct timeval`, 2) `time_t tv_vec ;`, 3) `suseconds_t tv_usec;`, and 4) `<end_of_struct>`.

[10]https://github.com/NationalSecurityAgency/ghidra/issues/2322

However, unfortunately, our preliminary experiments suggested that this hurts overall prediction accuracy. It also significantly slows down prediction, since it drastically increases the number of decoding steps. It moreover requires finer-grained accuracy metrics, like tree distance, to allow us to measure and credit partially correct predictions. Based on these observations, we believe unseen structure types should be handled specially with a tailored model, a problem we leave to future work.

Supporting Non-C Languages. A benefit of decompiling to C is that as a relatively low-level language, it can express the behavior of executables beyond those written in C. Although we designed DIRTY to be used with C programs and types, DIRTY can run on non-C programs, and will try to identify the C type that best captures the way in which that variable is being used. Thus, DIRTY provides value to analysts seeking to understand non-C programs, similar to how C decompilers such as Hex-Rays help analysts to understand C++ programs.

However, many compiled programming languages have type systems far richer than C's, and expressing these types in terms of C types may be confusing. For example, in C++, virtual functions are often implemented by reading an address out of a *virtual function table* [51, 147]. Although techniques like DIRTY can recognize such tables as structs or arrays of code pointers, it does not reveal the connection to the higher-level C++ behavior of virtual functions.

Extending DIRTY to support higher-level languages such as C++ is an interesting open problem. To some degree, as long as the decompiler is able to import the higher-level type information from debug symbols into the decompiler output, it should be possible to train DIRTY to recognize non-C types. For instance, 6% of the programs in DIRT are written in C++, and our evaluation measures DIRTY's ability to predict common C++ class types such as `std::string`. But recovering higher level properties of these types, especially for those never seen during training, is a challenging problem and is likely to require language-specific adaptations [51, 147].

Limited Input Length. As common with Transformers, we truncate the decompiled function if the length n exceeds some upper limit max_seq_length, which makes training more efficient. In our experiments we set max_seq_length to 512 for two reasons. First, 512 is the default value for max_seq_length in many Transformer models [44, 161]. Second, in DIRT, the average number of tokens in a function is 220.3, and only 8.8% of the functions have more than 512 tokens, i.e., we exclude relatively few of the possible inputs encountered in the wild. Still, if enough computational resources are available, we recommend using efficient Transformer implementations such as Big Bird [180] instead. These can deal with much larger max_seq_length and can be used out-of-the-box to replace our implementation.

4.4 Conclusion

In this chapter I addressed the problem of assigning decompiled variables meaningful names and types by statistically modeling how developers write code. While decompilers attempt to reverse compilation by transforming binaries into high-level languages, generating the same types originally written by the developer is impossible. However, in this chapter I presented the DecompIled variable ReTYper (DIRTY), a novel technique for improving the quality of decompiler output that uses machine learning to automatically generate meaningful variable names and types. Empirical evaluation of DIRTY on a novel dataset of C code mined from GitHub shows that DIRTY outperforms prior work approaches by a sizable margin, recovering the original names written by developers 66.4% of the time and the original types 75.8% of the time.

The evaluations used in this chapter and Chapter 3 have been useful for training machine learning models, but are only a proxy for real-world understandability. In the next chapter I will present a human study that I designed and ran to measure the *actual* impact of renaming and retyping on the performance of reverse engineers.

Chapter 5

A Human Study of Automatically Generated Decompiler Annotations

The experiments presented in Chapters 3 and 4 relied on exact comparisons to the names and types written by the original developer. This evaluation has the advantage that it is easy to automate and allows iterative improvement of techniques during their development and rapid training of machine learning algorithms. However, this metric also has disadvantages. First, it makes the assumption that the original developers' choices of variable names and types are objectively correct. Not only is this assumption not guaranteed to hold, is it sometimes completely incorrect: there have been instances of developers naming variables after famous athletes [153], and Star Wars characters [91]. Accurately predicting 75% of variable names and types that a developer chose might not be ideal if many of them were already poor or completely unrelated to their function in the code.

A second disadvantage is that it does not allow for abbreviations or synonyms. For example, such an evaluation does not consider **`str`** and **`string`**, **`size`** and **`length`**, or even **`xpoint`** and **`xPoint`** to be equivalent. It is possible to create heuristic methods to match these (e.g., manually creating a list of pairs to consider equivalent, using a word embedding technique like word2vec [122] to measure their difference, or creating a model specifically for variable names [31]), but it is not immediately clear how to adapt these to apply to user-defined types. An example is shown in Figure 5.1. A user-defined type for a node in a linked list is shown in Figure 5.1a, this structure has three fields: **`data`** stores the data in the current node, **`next`** is a pointer to the next node in the list, and **`head`** is a pointer to the head of the list. Two replacements

```
struct node {
  int data;
  struct node *next;
  struct node *head;
}
```

(a) Original type

```
struct node {
  int data;
  struct node *next;
}
```

(b) First replacement

```
struct node {
  int contents;
  struct node *following;
  struct node *first;
}
```

(c) Second replacement

Figure 5.1: A user-defined type to represent a node in a linked list, and two candidate replacements. A replacement score would depend on the context of the type being used.

are shown in Figures 5.1b and 5.1c. The first is identical to the original type but missing the `head` pointer, while the second has all of the same information, but the fields are renamed to `contents`, `following`, and `first` respectively. In this case, the replacement score should strongly depend on the context in which the type is used. For example, in a function that takes a pointer to the head of the list and searches for a data item, the head pointer might never be used and the first replacement would essentially be identical to the original type. However, in a different function that required access to the head pointer the second type, which does include a field for the head pointer, would need to be scored much higher.

Yet another disadvantage is that this metric does not consider how misleading the names and types are when a prediction is graded as incorrect. There is a large difference between replacing `str` with `string` and `point` with `size`. Similarly, the replacement type shown in Figure 5.1c is much different than the compatible type `struct person {int age; struct person *father ; struct person *mother;}`, but this is not captured by the metric used in Chapter 4.

An ideal metric is a proxy for what we would *actually* like to measure: the impact of annotations on the understandability of the code. We would like to measure how these renamings and retypings impact the performance of actual person using a decompiler. Does it make it easier to understand the code? Do they get the same answers more quickly? Does it cause them to get more questions correct?

To gain more insight into the real-world usefulness of a model trained using the naïve “exact match” metric, my collaborators and I developed a human study. This study was in the form of an online survey, which was presented to professional and amateur reverse engineers. Participants were asked to reason about real-world examples of decompiled code, with and without annotations from the DecompIled variable ReTYper (DIRTY) tool described in Chapter 4. We collected both qualitative and quantitative information about users’ performance designed to answer these questions. This chapter describes the results of this study. In short, the contributions of this chapter are:

- A human study protocol for measuring the effectiveness of renamings and retypings of variables in decompiled code.
- The results of a human study of 30 students, 9 full-time employees, and 1 unemployed person, all of whom do some amount software reverse engineering.

While we were unable to find statistical evidence that DIRTY has an overall meaningful impact on users’ correctness or completion time, the study did have interesting results. First, while there are instances where participants presented with DIRTY’s annotations are able to answer questions more correctly, there are others where DIRTY annotations actually *reduce* the correctness of answers. Second, DIRTY can lead more users to the correct answer in the same amount of time, but when names are particularly confusing it can cause users to take more time to reach the correct answer. Third, we found that users usually prefer types and universally prefer names that were automatically generated by DIRTY over those generated by the Hex-Rays decompiler. Finally, we found that users’ perceptions of the usefulness of DIRTY’s annotations do not always align with their performance on reverse engineering tasks.

5.1 Problem Overview

The goal of this chapter is to understand if the addition of automatically-generated variable names and types to decompiler output using a machine learning model trained with a naïve metric impacts humans' ability to reason about it. It is well known that code readability is essential for humans to understand the functionality of source code [21], but it is currently unclear how applicable this is to decompiler output. Decompilers tend to use standardized names for loop iterators (i.e., `i` and `j`), pull in names and types from standard library calls,[1] use `ret` for variables that are used for return types, and prepend the type of a variable to its name.[2] Votipka et al. have recently shown that renaming variables and reconstructing types are some of the most common tasks performed by reverse engineers [163]. However, there have been no recent human studies measuring the actual *impact* of renamings and retypings on the workflow of a reverse engineer.

We would like to answer the following research questions:

- **RQ1**: Do renamings and retypings allow reverse engineers to correctly answer more questions about decompiled code?
- **RQ2**: Do renamings and retypings allow reverse engineers to answer questions about decompiled code more quickly?
- **RQ3**: Do users of DIRTY perceive its renamings and retypings as improving their understanding?
- **RQ4**: Do users' perceptions of DIRTY's helpfulness always align with their performance?

These questions motivate our task selection and study design. We would like to represent tasks that closely resemble those performed on a day-to-day basis by a reverse engineer, rather than focusing purely on code readability.

5.2 Study Design

Our study is inspired by two existing works: one that measured the maintainability of patches automatically generated by computers [55], and another that measured the effectiveness of a research decompiler [178]. In these studies, participants were presented with snippets of code and asked to answer questions about them. Similarly, we presented participants with snippets of decompiled code generated by Hex-Rays v8.2 [71] and asked participants to analyze them and answer a number of questions about their functionality.

To measure the participants' opinion of the code, we also asked participants for feedback about the quality of the types and names, and asked them how much they agreed with general statements about their opinion of the code in general. Each participant was randomly given a code snippet that was either taken directly from the decompiler, or additionally augmented with the output of DIRTY.

[1] `https://hex-rays.com/products/ida/tech/flirt/`
[2] `https://ghidra-sre.org/`

5.2.1 Code Selection and Question Formulation

Our study design constrained the specific code snippets that we could use. First, since we were targeting a single hour for the study, and to avoid issues with scrolling, the snippets needed to be short enough to fit on a single screen together with the questions. This imposed a 50-line limit to the length of each code snippet. Second, the snippets needed to be "interesting" enough to be able to measure if there is a performance difference between groups. This requirement was addressed by requiring a nesting depth of at least 2 of either branching constructs like `if`, or control flow constructs like `for`. Third, the code snippet had to be entirely self-contained: it must be possible to ask a question about a snippet without requiring information about the behavior of called functions. Fourth, since we are interested in the impact of renaming and retyping tools on performance, each snippet must have at least three variables that are renamed or retyped.

We sourced our code from `lighttpd`, `coreutils`, and `openssl`. These projects were chosen because they represent common functionality that are often used by malware authors, like networking, encryption, and file access. To ensure that the questions were realistic, a professional reverse engineer was consulted. The selected snippets were the following:

1. **`array_extract_element_klen`** This function came from `lighttpd`. It is designed to find an element of a custom array type given a key, maintaining metadata inside it.
2. **`buffer_append_path_len`** This function also came from `lighttpd`. It is designed to take two file paths and concatenate them, ensuring that there is a maximum of one path divider between them (i.e., if two `char *` variables contain `"usr/"` and `"/bin"`, calling `buffer_append_path_len()` on them will return `"usr/bin"`).
3. **`postorder`** This function came from `coreutils`. It takes three arguments, a binary tree, a function pointer to call at every node, and auxiliary information to maintain. It calls the function in the function pointer at every node in the binary tree in postorder.
4. **`twos_complement`** This function came from `openssl`. It is designed to take an input buffer, an output buffer, a length, and copies the input buffer to the output buffer. If the padding argument is `0xff`, it also converts the input buffer into its two's complement form before copying.

Question Formulation

To measure the effectiveness of our technique on the reverse engineering process, we require questions that are similar to those that would be asked by reverse engineers during the process of reversing a binary. We asked questions of the following form:

- If the function is called with arguments X what will the value of Y be at line number Z?
- What is the purpose of the code on lines X through Y?
- What are the possible return values of this function?
- Which argument of this function corresponds to functionality X?

These questions are based on those asked in an earlier study [55], and were formulated together with a professional reverse engineer to ensure their realism. Using this formulation, we created 2 questions for each snippet, for a total of 8 questions. The code and questions asked about each of these snippets is contained in Appendix A.

5.2.2 Methodology

We conducted a between-subjects experiment where participants were asked to analyze code snippets either decompiled with Hex-Rays or decompiled with Hex-Rays and augmented with the output of DIRTY (cf. Chapter 4). The study was performed online with the LimeSurvey platform.[3] To begin, we provided each participant with a detailed explanation of the study and the experimental procedure. Participants were not allowed to use the Internet during the study, since our goal was to remove any variables other than the code quality itself.

Each participant was presented with a code snippet and asked a sequence of questions about it. All four snippets of code were shown to all participants, and the treatment was randomized per-snippet (i.e., a single participant could see the Hex-Rays version of **`buffer_append_path_len`** and the DIRTY version of **`postorder`**). We found that this type of randomization gave us more flexibility as a single participant not completing the survey would not remove a participant from an entire treatment group for all questions. We collected both timing and correctness statistics for each participant.

Variables and Conditions

In our experiment, we have two independent variables:

1. Treatment. Each code snippet was generated by the Hex-Rays decompiler, which is widely used by malware analysts. We tested with Hex-Rays version 8.2.230124, which was the latest version of Hex-Rays at the time of writing. We tested the output of the decompiler against the same output but augmented with annotations from DIRTY.
2. Questions. Each question has its own unique difficulty level, so must be considered individually.

User Perception

After finishing the two questions for each snippet, participants were shown a brief questionnaire. First, they were asked to rate how the type of each argument and the name of each argument affected their understanding on a 1-5 point scale. Second, they were asked to optionally suggest better types and names for each argument. Finally, they were asked on a 5-point scale to rate how much they agreed with statements about their subjective opinions about the code as a whole. The full list of these statements is contained in Appendix A.

5.3 User Study

In this section I will discuss the recruitment process for the study and results we collected.

[3]`https://limesurvey.org`

5.3.1 Participants

Since we aim to measure the impact of our tool on actual reverse engineering, we directly recruited participants based on prior knowledge of their reverse engineering ability. We sent personalized emails to professional reverse engineers at the Carnegie Mellon University Software Engineering Institute, engineers at GrammaTech, professors who teach reverse engineering courses, and members of capture-the-flag teams. Each participant was sent a slightly modified version of the following email:

> Hi, my name is Jeremy Lacomis and I'm a Ph.D. student at CMU. I'm currently working on a human study trying to measure the performance of our tool, DIRTY, that augments decompiled code with human-written names and types. I've been reaching out to people who have some experience with reverse engineering. I'd really appreciate it if you could take the survey and/or pass it along to other people you think might be a good fit.
>
> You can find the survey here [LINK]. It involves reasoning about decompiled code and is designed to take about an hour.

Overall we received responses from 31 students, 10 full-time employees, and 1 unemployed person. Participants were anonymous and were not offered any compensation. We did not require a response to every question. Since it is possible for someone to rapidly go through the survey while entering meaningless information, we ensured that the time spent on a snippet was at least as long as it took for the author to read the first question fully. This criterion was used to discard every answer from 1 student and 1 full-time employee, so they were removed from the study completely. Figure 5.2 shows the age, gender, and education breakdown of our participants.

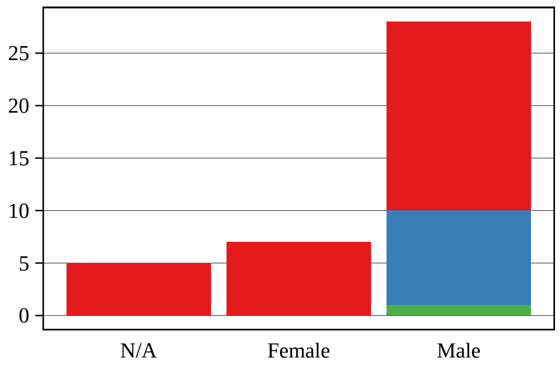
5.3.2 RQ1: Correctness

The first research question asks if DIRTY's annotations allow users to answer more questions correctly. Recall that each participant is randomly assigned to a treatment group for each of 4 snippets (i.e., a snippet with DIRTY annotations, or a snippet without DIRTY annotations). For each snippet we ask the participant 2 questions, for a total of 8 questions per participant.

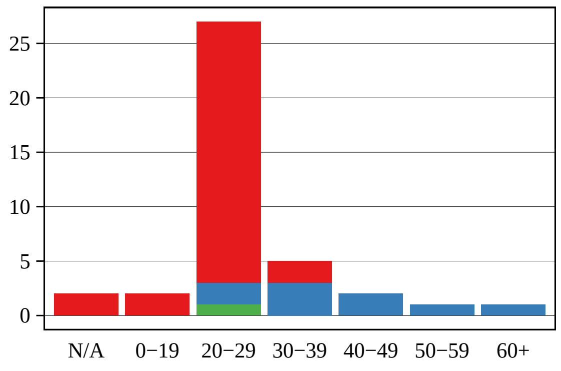
The key idea behind the analysis is to compare the distributions of variables between tasks completed in the treatment and control conditions. However there are some difficulties. First, each participant will have answered multiple questions over the course of the study. Second, not all tasks are of the same difficulty and more difficult tasks are likely to have fewer correct answers. Finally, the participants themselves have varying levels of experience with respect to computer science education and reverse engineering.

Therefore, instead of the common fixed-effects regression model, we used a mixed-effects regression [60]. Fixed-effects models assume that residuals are identically distributed, which is not the case in this study. For example, it is likely that a more experienced reverse engineer is able to answer all of the tasks more quickly and more correctly than less experienced reverse engineers. Unlike fixed-effects models, mixed-effects models group residuals by *random effects*.

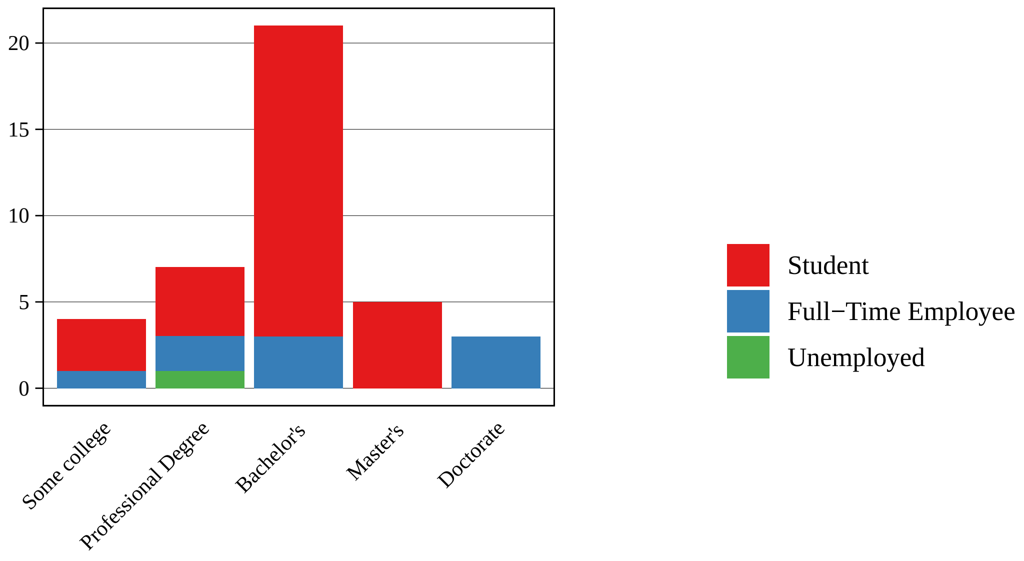
Since we expect the correctness to vary per-user and per-question, we choose these as our random effects. Our hypothesis is that DIRTY leads to more users correctly answering the question, so we use this as one of our fixed effects. At the end of the study, we also asked users to



(a) Gender



(b) Age



(c) Education

Figure 5.2: Age, gender, and education breakdown of out participants.

Table 5.1: GLMER Correctness Performance Model

	Dependent variable:
	Correctness
Uses DIRTY	-0.074
	(0.227)
General Coding	0.056^{*}
	(0.030)
Reverse Engineering	-0.024
	(0.044)
Constant	0.563
	(0.513)
Observations	273
Num Users	36
Num Questions	8
σ(Users)	0.85
σ(Questions)	1.14
R^2_m	0.041
R^2_c	0.405
Akaike Inf. Crit.	313.091
Bayesian Inf. Crit.	334.747

Note: *p<0.1; **p<0.05; ***p<0.01

self-report their years of experience with reverse engineering and general coding; we hypothesize that these also have an effect on the correctness of responses so also use these as fixed effects. Thus, in R syntax, we estimate the model:

$$\texttt{correctness} = \texttt{uses\_DIRTY} + \texttt{Exp\_Coding} + \texttt{Exp\_RE} + (1|\texttt{user}) + (1|\texttt{question}) \quad (5.1)$$

Correctness is a binary score, which means that we must use a logistic regression. We use the `glmer` function in R to estimate this model. For this test, we use the standard level of $p < 0.05$ for statistical significance of coefficients. Additionally, as indicators of goodness of fit, we report a marginal (R^2_m) and conditional (R^2_c) coefficient of determination. The marginal R^2 considers only the variance of the fixed effects, without the random effects, while the conditional R^2 considers both the fixed and random effects. For the generalized linear mixed model that we use for this experiment, we compute R^2 using the `r.squaredGLMM` function in R, which implements the algorithm described by Nakagawa et. al [125].

Table 5.1 summarizes the results of our model. This model fits our data reasonably well, with an R^2_c of 40.5%. We found no statistically significant difference when participants use DIRTY and do not have sufficient evidence to conclude DIRTY users answer more questions correctly.

However, Figure 5.3 shows the correctness of answers to the questions asked during the survey. From this, we can observe that the use of DIRTY allows increased correctness on **`array_extract_element_klen`** question 1, both **`buffer_append_path_len`** questions, **`postorder`** question 1, and **`twos_complement`** question 2. Although the data does not allow us to make a strong claim about its effect (a Fisher's exact test for count data on these five examples has a p-value of 0.1576), there does appear to be a clear correlation between the two. More data will need to be collected in future work to further explore this hypothesis. From this we can speculate that while DIRTY might not improve correctness on tasks in general, there might be specific instances where it is helpful.

DIRTY can be misleading. Occasionally, DIRTY can be misleading. An example is shown in Figure 5.4. In this question, **`postorder`** question 2, we told the participants that the three arguments represented a pointer to a tree structure, a function pointer to call on each node, and auxiliary information to maintain as the tree was being traversed and asked them to match the arguments to their description. The body of the code example makes it fairly clear that the first argument is the tree structure, meaning participants must reason more about the function pointer and auxiliary information. Figure 5.4a shows the original Hex-Rays version of the code, which strongly suggests that the second argument is the function pointer, however Figure 5.4b shows DIRTY's suggestions. DIRTY's suggestions are seemingly quite good; it correctly identifies that two of the arguments correspond to a tree and a comparison function. However, it *reverses the order of the last two suggestions*. Figure 5.3 shows that while almost every person who received only the Hex-Rays output answered this question correctly, nearly half of participants who received the DIRTY annotations answered this question incorrectly (a Fisher's exact test on this data confirms this confusion, p = 0.01059).

Anticipating this, we also asked participants to justify their answers by answering the question "Informally, how did you reach your conclusion?". For qualitative analysis we used the standard grounded theory approach of open coding [25]. Each response was individually coded, then these codes were synthesized and used to identify themes. Two main themes were identified

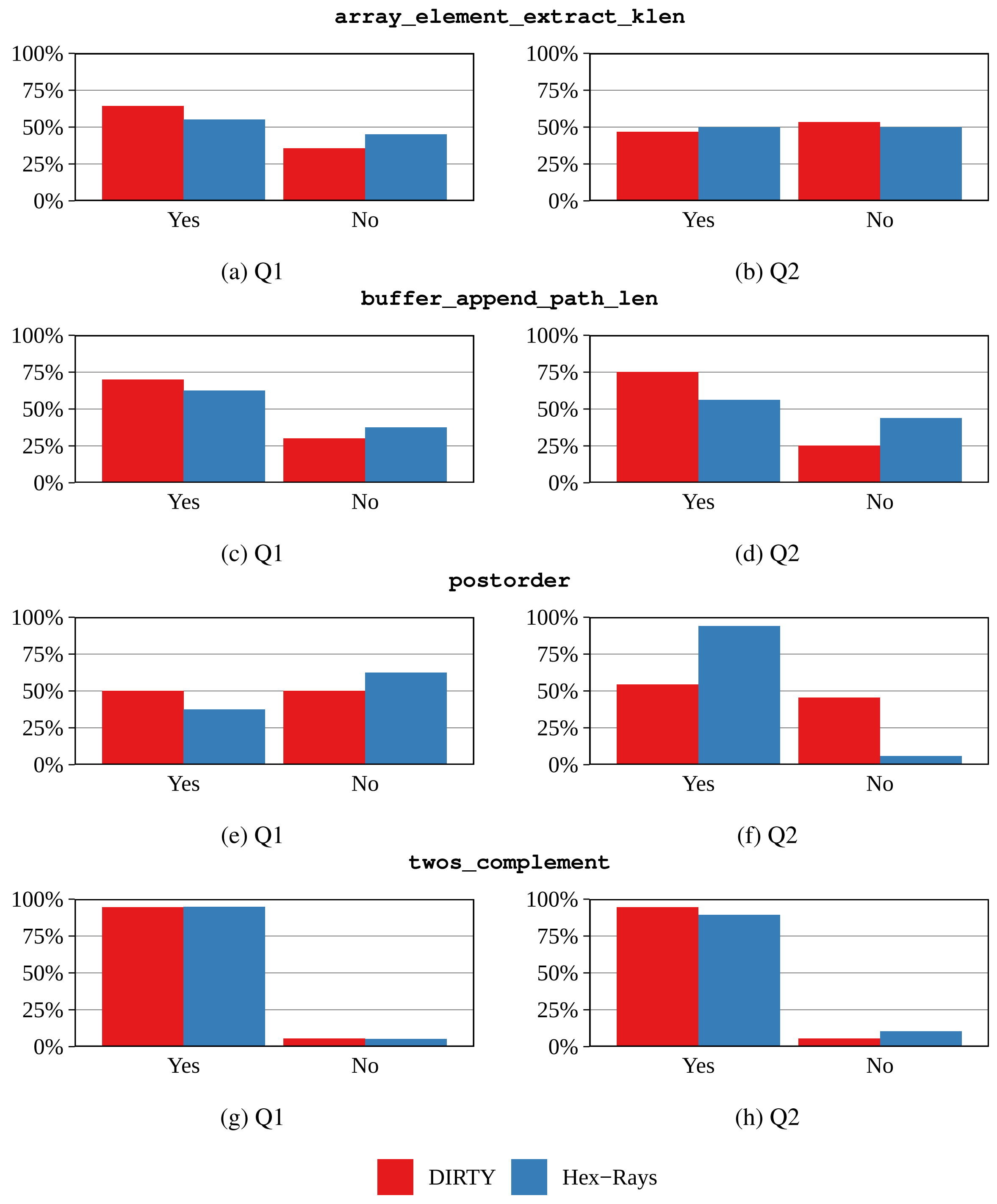


Figure 5.3: Answers to questions grouped by treatment. “Yes” means the answer was correct, “No” means incorrect.

```
1  __int64 __fastcall postorder(
2    _QWORD *a1,
3    __int64 (__fastcall *a2)(__int64, _QWORD *),
4    __int64 a3) {
5    // ...
6        v5 = a2(a3, a1);
7    // ...
8  }
```

(a) Hex-Rays

```
1  __int64 __fastcall postorder(
2    tree234 *t,
3    void *e,
4    cmpfn234 cmp) {
5    // ...
6        ret = (e)(cmp, t);
7    // ...
8  }
```

(b) DIRTY

Figure 5.4: **`postorder`** question 2, an example where DIRTY recommends reasonable types and names, but applies them to the wrong arguments. Participants were asked to match each argument to its purpose. In the code in (a), **`a1`** is a pointer to a tree structure, **`a2`** is a function pointer, and **`a3`** is auxiliary information. Note how in (b) DIRTY correctly identifies that there is a tree type and a function, but incorrectly reverses the meanings of **`a2`** and **`a3`**. Figure 5.3f shows that almost half of the participants who received the DIRTY suggestions answered incorrectly.

in answers from the users who received the DIRTY version of the code, which were directly correlated with correctness of their answer.

Among participants who received the DIRTY code and answered *correctly*, we identified the theme: **The usage of the variables inside the code demonstrate their purpose** (P5, P6, P7, P8, P9, P11, P14, P15, P16, P17, P18, P19). These participants indicated that they considered not just the names and types of the variables, but also their usage in the code. These participants summarize this thought process well:

> *"Line [6] shows the actual function call; that requires* `e` *to be the function and* `cmp` *to be an argument to it, at odds with the type information in the arguments" (P8)*

> *"Similar to the previous answer – I ignored the types and looked at the use. The only actual call through a function pointer is on line [6], so* `e` *is the visit/comparison function. It is passed in* `cmp` *(which is never changed, despite being confusingly named the same as the* `->cmp` *field), so the* `cmp` *argument is the additional information [...]" (P11)*

Participants who answered the question *incorrectly* reached their conclusion for a different reason: **The variable names and types themselves indicate their intended usage** (P1, P2, P3, P4, P6, P10, P12, P13). These participants took the types and names at face-value and did not consider their usage in the code. Indicative responses are:

> *"The variable names were very intuitive. For the tree, the type and its usage in the code was really helpful." (P1)*

> *"The main giveaway is the naming. Also I see that* `cmpfn234` *is defined as a function pointer. The naming are very descriptive and helped in identifying what each component does." (P13)*

Notice how participants who referenced the code itself and were skeptical of the types suggested by DIRTY got the answer correct, while participants who got the answer incorrect trusted the types it suggested. The pattern of accepting DIRTY's annotations at face value was not unique to the **`postorder`** example. For users who received DIRTY annotations, we compared groups based on their correctness by the Likert opinions participants assigned to the types DIRTY suggested using a Wilcoxon rank sum test with continuity correction. We found that participants

Table 5.2: LMER Timing Performance Model

	Dependent variable:
	Completion Time
Uses DIRTY	26.296
	(16.865)
General Coding	4.488^{*}
	(2.620)
Reverse Engineering	-5.647
	(3.948)
Constant	192.658^{***}
	(54.308)
Observations	296
Num Users	37
Num Questions	8
σ(Users)	94.77
σ(Questions)	130.96
R^2_m	0.025
R^2_c	0.431
Akaike Inf. Crit.	$4,026.521$
Bayesian Inf. Crit.	$4,052.354$

Note: *p<0.1; **p<0.05; ***p<0.01

who answered incorrectly tended to trust DIRTY's suggestions more than participants who answered correctly *across the board* (p = 0.02477). This suggests that it is important to train users to remain skeptical while reverse engineering, even with the types suggested by DIRTY. DIRTY is still inherently a machine learning process and its output is not guaranteed to be correct.

RQ1 Answer: We did not find statistical evidence that using tools like DIRTY allows for more users to reach a correct conclusion. However, there is some anecdotal evidence that DIRTY might help for specific tasks. We also observed one case where users were misled by DIRTY's incorrect predictions that appeared to be correct.

5.3.3 RQ2: Timing

Research question 2 asks if the DIRTY plugin has an impact on the timing of participants' answers. Similarly to correctness, we fit a mixed-effects model of the equation:

$$\texttt{timing} = \texttt{uses\_DIRTY} + \texttt{Exp\_Coding} + \texttt{Exp\_RE} + (1|\texttt{user}) + (1|\texttt{question}) \tag{5.2}$$

Unlike the binary "correctness" results, timings are continuous and we can use a standard linear mixed-effects model provided by the `lmer` function in R. Our results are shown in Table 5.2.

First, the model does fit our data reasonably well, with an R^2_c of 43.1%. As with the correct-

```
// Original
void buffer_append_path_len(buffer * restrict b, const char * restrict a, size_t alen)
// Hex-Rays
void *__fastcall buffer_append_path_len(__int64 a1, _BYTE *a2, size_t a3)
// DIRTY
void *__fastcall buffer_append_path_len(SSL *s, const char *str, size_t n)
```

(a) Function signature

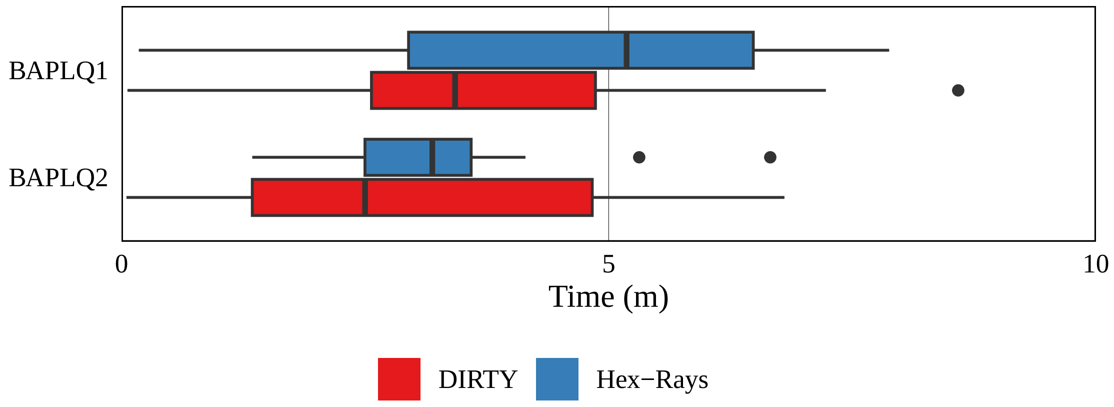


Figure 5.5: Signatures and completion time for both **`buffer_append_path_len`** tasks, per group.

ness results, none of our coefficients were statistically significant. We did not find statistically significant evidence to conclude that users of DIRTY were able to answer questions more quickly.

DIRTY can allow more participants to reach the correct conclusion without taking more time. Figure 5.3 shows a correlation between correctness and the use of DIRTY on **`buffer_append_path_len`** tasks. Figure 5.5 shows the completion times for both groups. There does not appear to be a statistically significant difference in the completion time between these groups (the Hex-Rays group has a mean of 256 seconds and a standard deviation of 145, while the DIRTY group has a mean of 242 seconds and a standard deviation of 202), but DIRTY does have some impact on reasoning ability. For example, notice how the signatures shown in Figure 5.5 indicate that DIRTY's choice of the type and name of the second argument suggest that is used as an input string.

Sometimes DIRTY can lead to people taking longer to reach the correct conclusion. Figure 5.6 shows the amount of time it took people to correctly answer **`array_extract_element_klen`** question 2. It took just over three and a half minutes longer for users who were shown the DIRTY output to reach a correct conclusion than users without it. From our experience, we suspect that this is for multiple reasons: First, DIRTY assigns the name **`ret`** to a variable that is never used for a return value, therefore users need to carefully scan and make sure they have spotted every **`return`** statement in the code. Second, the previously confusing statement on line 9 has become even *more* confusing. For a practiced reverse engineer,

```
 1  __int64 __fastcall array_extract_element_klen(__int64 a1, __int64 a2, unsigned int a3) {
 2    //...
 3    //...
 4    int index;
 5    __int64 v7;
 6    //...
 7    if ( index < 0 )
 8      return 0LL;
 9    v7 = *(_QWORD *)(8LL * index + *(_QWORD *)(a1 + 8));
10    //...
11    return v7;
12  }
```

(a) Hex-Rays output

```
 1  char *__fastcall array_extract_element_klen(array_t_0 *array, void *key, int index) {
 2    //...
 3    int indexa;
 4    int ret;
 5    char *next;
 6    //...
 7    if ( indexa < 0 )
 8      return 0LL;
 9    next = *(char **)(8LL * indexa + *(_QWORD *)&array->size);
10    //...
11    return next;
12  }
```

(b) DIRTY output

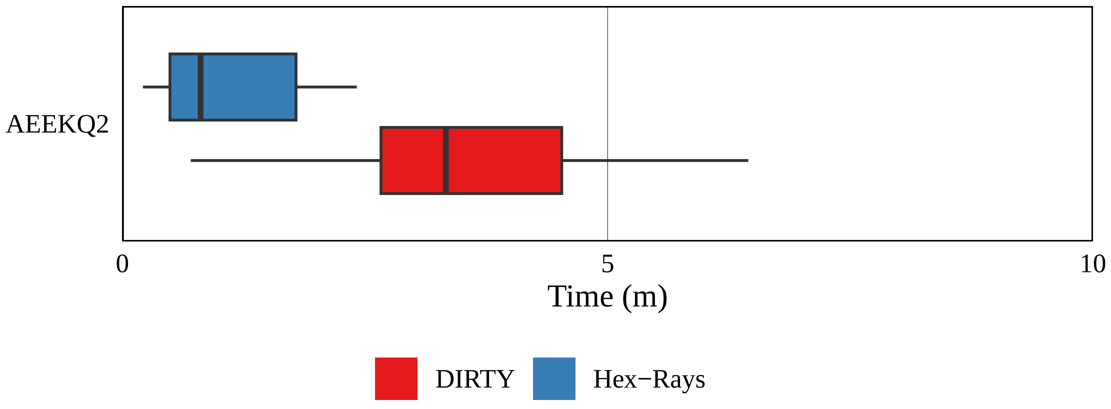


Figure 5.6: Completion time for correct answers to the `array_extract_element_klen` question “What does this function return?”. DIRTY’s answers take a statistically significantly longer time (Student’s T-test, p=0.03309). This is likely due to DIRTY suggesting the name `ret` for a non-return variable and the even more confusing decompilation on line 9.

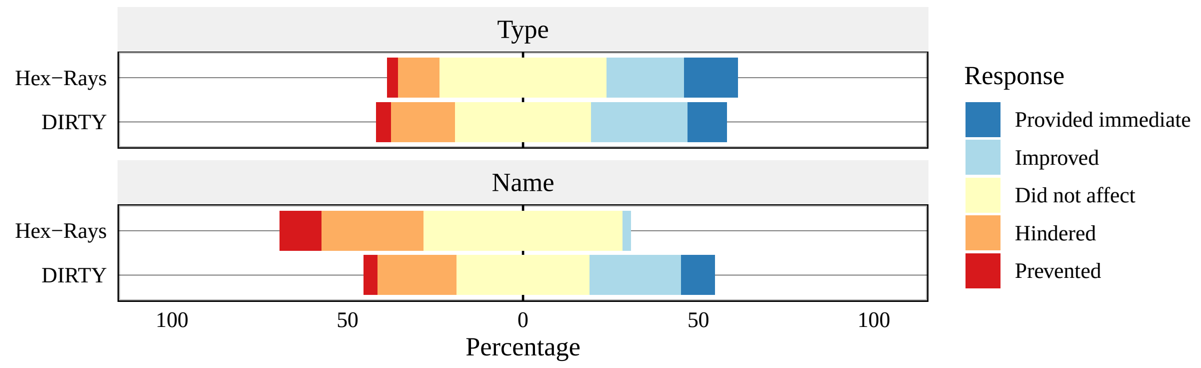


Figure 5.7: Participants' overall opinions of how the types (top) and names (bottom) of Hex-Rays and DIRTY output impacted their understanding.

the pattern `8 * index + *(a1 + 8)` indicates an access of an element of an array contained in a struct pointer (cf. the Chapter 1 example). DIRTY's annotation does not make this clearer, in fact it muddies the water. Although the type DIRTY predicts is in some sense "correct" (it predicts the type `array_t_0 *`, while the original code used the type `array *`), the layouts of these types are different. The `size` field of `array_t_0` should instead be a pointer to an array.

RQ2 Answer: We were unable to find statistically significant evidence that DIRTY reduces the time that users spend on reaching a correct conclusion. However, we find that in some cases DIRTY can lead more users to the correct answer in the same amount of time. We also find that particularly confusing names can lead users to take more time to reach the correct conclusion.

5.3.4 RQ3: Opinions

The third research question asks about users' perception of the usefulness of the renamings and retypings provided by DIRTY. To answer this, for each argument in a snippet, we asked participants to fill in the blank in the statement "The type and name of this argument _____ understanding:" with "Prevented", "Hindered", "Did not affect", "Improved", or "Provided immediate".

Figure 5.7 shows the participants' overall opinion of how the types and names from both DIRTY and Hex-Rays impacted their understanding of the code. We performed a Wilcoxon rank sum test with continuity correction on these results and found that in general, users prefer when variables are at least given *some* name, even if it does not agree with the actual use of the variable (p = 5.072e-14, difference in location = 0.9999579). This result is not surprising: the names generated by Hex-Rays in this study are themselves rarely indicative of the purpose of the variable, except in cases like `ret` for a return variable or `src` and `dest` for source and destination variables. DIRTY is better at assigning names that carry more nuanced semantic meaning.

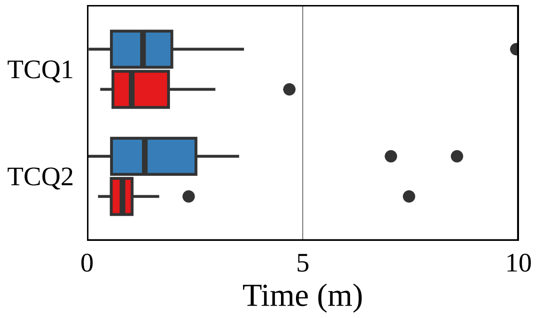
There is an inability to make a overall statistical claim about users' opinions about types, but with the `twos_complement` snippet, DIRTY's suggestions are considered quite poor by users. I will discuss this specific example later, but by excluding it the Wilcoxon rank sum test does show

```c
__int64 __fastcall twos_complement(
  __int64 a1,
  __int64 a2,
  __int64 a3,
  char a4){
  // ...
  *(&savedregs - 3) = a1;
  *(&savedregs - 4) = a2;
  *(&savedregs - 5) = a3;
  *((_BYTE *)&savedregs - 44) = a4;
  *((_DWORD *)&savedregs - 1) = *((_BYTE *)&
      savedregs - 44) & 1;
  if ( *(&savedregs - 5) )
  // ...
}
```

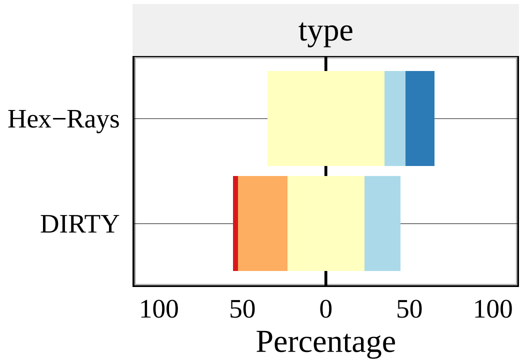
(a) Hex-Rays

```c
__int64 __fastcall twos_complement(
  void *p1,
  void *p2,
  void *data,
  int i2) {
  // ...
  i = (__int64)p1;
  j = (__int64)p2;
  k = (__int64)data;
  v8 = i2 & 1;
  if ( data )

  // ...

}
```

(b) DIRTY



(c) Timing for correct answers.



(d) Likert scores of types for correct answers.

Figure 5.8: **`twos_complement`**, an example where DIRTY recommends reasonable types and names, but users disliked the types. Red bars represent DIRTY, while blue bars represent Hex-Rays. Notice that Figure 5.3 shows that approximately the same proportion of users answered correctly, but DIRTY users took less time (c), despite disliking the types (d).

statistical significance, with a p-value of 0.01158. We take this as a suggestion that DIRTY's types are usually considered helpful by users.

RQ3 Answer: We find that users universally prefer the names provided by DIRTY to be more helpful than the names provided by Hex-Rays. We also find that, in most cases, users perceive that the types provided by DIRTY are more helpful than the default types.

5.3.5 RQ4: Users' perception vs. performance

Sometimes it is the case that users do not prefer the DIRTY-suggested types, despite them being helpful. In Figures 5.8a and 5.8b, we can see snippets of code without augmentation, and with augmentation, respectfully. Notice in Figure 5.3 that the proportion of users who correctly answered two questions about these snippets is approximately the same, however Figure 5.8c shows that users who had the DIRTY augmentations *reached the correct answers more quickly* than users without them. Despite this, Figure 5.8d shows that the opinions of the types assigned by DIRTY in this example were lower than the types originally assigned by Hex-Rays.

The original signature of this function was `static void twos_complement(unsigned char *dst, const unsigned char *src, size_t len, unsigned char pad)`. When asked for suggestions about types, two participants suggested that DIRTY's suggestion of `void *data` for the third argument was incorrect, and should instead be `size_t size` and indicated that the type and name of this argument "Hindered" and "Prevented" their understanding respectively. Note that Hex-Rays' suggestion was `__int64 a3`, and *no participants indicated that any type "Hindered" or "Prevented" their understanding*. This suggests that users have a higher expectation of a system that suggests type information, even when it actively improves their performance on a task.

RQ4 Answer: We find that users perceptions of the usefulness of DIRTY's annotations do not always align with their actual performance on reverse engineering tasks.

5.3.6 Generative Large Language Models

At the time of this dissertation, Generative Large Language Models (LLMs) such as ChatGPT[4] have been exploding in popularity. These models are able to receive input text from a user (e.g., a natural language question such as "Can you generate some reasonable acronyms to use as the name of my new technique?") and generate highly probable text based on training data. While it is still an open question if these models would be useful for reverse engineers, there have been several plugins developed for decompilers such as Ghidra that are designed to explain the purpose of decompiled functions.[5]

With this in mind, we additionally performed an experiment where we presented ChatGPT with the entire text of the reasoning questions we asked participants in this study (for both treatment groups). We were surprised to find that ChatGPT gave very thorough correct answers for all of the questions, except one. The only question it answered incorrectly was when DIRTY incorrectly labeled the arguments of a function and it took the names and types at face value (this also fooled half of our participants, see "DIRTY can be misleading" in Section 5.3.2).

While this seems promising, ChatGPT is not perfect. For example, in response to one of the questions, it said "This appears to be a function written in x86-64 assembly language", despite the code having features such as `if` statements and `while` loops that are not a part of x86-64 assembly. This is an instance of a common problem with generative language models called "hallucination", where text that is either nonsensical or unrelated to the input is generated [82]. These issues are not minor and have come up in the real world, with ChatGPT summarizing non-existent New York Times articles,[6] or more seriously, completely fabricating cases in legal

[4]https://chat.openai.com/chat

[5]https://github.com/evyatar9/GptHidra

[6]https://en.wikipedia.org/wiki/Hallucination_(artificial_intelligence)#/media/File:ChatGPT_hallucination.png

research presented in court filings.[7] It remains an open question if LLMs are a reasonable choice for the task of reverse engineering.

5.4 Discussion and Future Work

As the demand for reverse engineers increases the tools that we provide to them must also become more advanced. This chapter shows that there are at least two distinct problems with the development of new decompiler techniques. First, as decompiler output becomes more readable, people put more trust in it and make mistakes that they would not have given lower-level output. Second, there is indeed a need for a more nuanced metric for measuring the output of augmentation techniques.

This study demonstrates that users of decompilers will believe the information they are given, even when they know that the output is unreliable. Users occasionally misinterpreted the annotations or relied on them too heavily, leading them to incorrect answers. This highlights the need for caution when using annotations as a crutch, and the importance of developing an understanding of the decompiled code itself. While these annotations could be a valuable tool for reverse engineers, they should be used in conjunction with other strategies for code comprehension. A shortcoming of this study was that it was not interactive; as one participant noted:

> *I find RE difficult if the process is isolated to one type of technique. In my experiences the RE process requires jumping between both static and dynamic analysis to get a complete picture. It'd be nice if the decompilers better fit into the RE process from this standpoint.*

The inability of participants to actually run the code that we presented to them, use a debugger, or be able to annotate the code that they were examining likely led to some incorrect answers. Future studies should add these features to gain more insight into the specific needs of engineers.

There is indeed a need for a more nuanced metric for understandability than direct comparison to the original code. This metric should be able to capture not only the relationship of the name itself to its original name, but also capture details about the process by which a reverse engineer reaches a conclusion about code, and the relative "importance" of a specific variable. It is likely much better to get a single type/name correct that is used often on a critical path of reasoning than to correctly assign types and names to variables that lie off of these paths. I have personally developed a technique called VarCLR [31] that uses contrastive learning to minimize the distance between variable name tokens in an embedding space. While this technique does work well, further research is needed for it to be extended to types as well.

To avoid the confusion caused by DIRTY guessing correct labels but assigning them to the wrong variable, a future technique could instead suggest a set of types and names that a user could reference when reasoning about code. Another alternative would be to suggest types and names to users, but indicate when it is reasonable to exchange them. The **`postorder`** example demonstrates the need for a metric to measure how *confusing* specific annotations are. Future techniques should also investigate optimal levels of annotation, for example by tuning a confidence boundary below which annotations are not produced at all.

[7] `https://apnews.com/article/artificial-intelligence-chatgpt-courts-e15023d7e6fdf4f099aa122437dbb59b`

In Section 5.3.3, I observed that there are times when DIRTY's typing information can be helpful while being distrusted by users. While skepticism is healthy, it can also be counterproductive. Future work might investigate ways to modulate users' trust in machine learning, for example by allowing the user to switch between the original version and the augmented version of the code to sanity-check their own intuition.

5.5 Threats to Validity

A threat to validity of this study was that our technique was only tested on open-source snippets of software. This software is (ideally) specifically designed to be laid out as sensibly as possible. This is rarely the case with real-world malware, however. Obfuscating control flow, fully stripping function names, using self-modifying code, even hand-writing assembly that cannot be generated by a compiler, are all ways that malware authors attempt to hide the purpose of their binaries. Additionally, our method of sampling from GITHUB naturally biased our dataset to projects that are well-known.

Our study only measured the output of the Hex-Rays decompiler, and did not consider alternatives such as Ghidra [61]. This was primarily because DIRTY was tailored specifically to Hex-Rays' representation of a binary (recall from Chapter 4 that one of the inputs to DIRTY is a binary's memory layout). It is possible that other decompilers' output is easier to understand and would not be improved as much as Hex-Rays.

Our code snippets are not indicative of the entire process of reverse-engineering, only a small portion of it. The requirement that all of our code be self-contained on a single page is particularly restrictive. Integrating directly with a decompiler would likely improve results.

5.6 Conclusion

In this chapter, I described a novel human study to measure the impact of the DecompIled variable ReTYper (DIRTY) on the performance of reverse engineering tasks. In this study, I asked professional and amateur reverse engineers to answer questions designed to simulate real-world reverse engineering tasks. I recorded correctness and timing information, and collected qualitative data asking users' opinions about the quality of the renamings, retypings, and overall structure of the code.

Even though the experiments in this chapter have been limited to a single decompiler and renaming/retyping technique, the methodology here can be applied to measure the effectiveness of other techniques. Additionally, this methodology can be built on, for example by allowing navigation between code snippets, in-place editing of code, or even execution of the code itself.

Chapter 6

Conclusions and Final Remarks

In summary, this thesis contains techniques for automatically augmenting the output of decompilers with more meaningful variable names and types under the hypothesis that this will decrease the cognitive burden of reasoning about source code. I believe that the techniques presented here will save reverse engineers valuable time that could be instead spent reasoning about the higher-level functionality of code. I also believe that these techniques can flatten the learning curve, allowing more novices to enter the field of reverse engineering.

The human study I presented in Chapter 5 provides some evidence that is might be the case. The names and types annotated by DIRTY were found to be perceived as helpful to users' understanding and often allowed more users to reach correct conclusions. There were also cases where the annotations were found to increase the number of users who answered questions *incorrectly*, together, these observations provide evidence that the choice of names and types has a real-world impact on the understandability of decompiler output.

Meaningful variable names and types are not nearly the only problem that affects the readability of decompiler output. In fact, my coauthors and I have a taxonomy of comprehension issues that are induced by decompilers under submission to USENIX now. This taxonomy includes issues such as decompilers not generating code that should be generated, generation of incorrect code, and obfuscated control flow (e.g., converting a `switch` statement into a sequence of `if` statements). A simple example, and one that was represented in the study in Chapter 5 is the representation of literals. In the `buffer_append_path_len` code example, the number 47 is used in place of the character `'/'`, which is used to determine if a path begins with a forward slash. This example shows how reverse engineers need to reason about even the smallest details.

Reverse engineers are in high demand. At the time of writing, AV-ATLAS[1] reported that 4,914,123 new instances of malware were detected *in the past 14 days*, while the United States Cybersecurity & Infrastructure Security Agency has 2,758 employees [120]. There are more malware threats detected every day than there are employees at the agency responsible for ensuring the cybersecurity of the United States. Additionally, the applications of reverse engineering are not limited to malware. Reverse engineering can be used for "clean room" design to avoid copyright infringement. One famous example is Phoenix Technologies using reverse engineering to create a version of the previously IBM-proprietary BIOS [90], which was required to

[1]`https://portal.av-atlas.org`, Accessed May 10, 2023

develop their own IBM-compatible PCs. Engineers at Phoenix examined IBM's Technical Reference manual, writing a set of specifications without including examples of actual code. These specifications were then passed to a single programmer who had never worked on an 8086 microprocessor before, which allowed Phoenix Technologies to defend itself from IBM lawsuits. As the number of tasks that demand reverse engineers increase, tools must improve to keep up.

A shortcoming of the techniques presented here is their scalability. Machine learning is powerful, but the resources required to train and deploy them are quite high. Even after the collection of data, which took 2 months, and post-processing, which took another 2 weeks, training the DIRTY model to simultaneously predict names and types took 5 days of time on NVIDIA Tesla V100 GPUs, and not everyone has access to expensive hardware. Future name and type prediction techniques will require engineering advances to enable models that can be trained and run on less expensive hardware.

One of the core challenges with using supervised machine learning techniques to augment decompiler output is the need for training data. The techniques presented in this paper were successful, but the generation of the training data took multiple months and required scraping a large amount of source code. This is a well-known problem in the machine learning world in general, and research into ways to automatically generate more training data is ongoing. Another solution to this problem is to use unsupervised machine learning, which requires no or very little training data, but these techniques are almost universally less performant than supervised techniques.

Training data is important not only for improving model performance, but also for adapting models to different tasks. There is nothing about the tools developed for this thesis is specific to a particular decompiler, but the training data is decompiler-specific. Our technique for generating training data only relies on a decompiler's ability to import DWARF debugging information. Since the publication of DIRTY, other researchers have been able to adapt it to Ghidra [24]. Similarly, our technique is not inherently language-specific (i.e., it is not restricted to C-like languages). Future research might focus on extending DIRTY to output suggestions for other languages such as Go or Haskell.

The techniques described here must be run post-hoc on the output of the decompiler, and are not integrated directly. Direct integration with a decompiler could enable some other interesting features. For example, a user could individually chose a suggested type and the system could use this information to refine the suggestions for other surrounding types.

DIRTY and DIRE both assume knowledge of function names and use this information when assigning types and names. This is a reasonable assumption when using a decompiler in a non-adversarial situation such as updating the functionality of a binary without its source code. However in malware, where obfuscation is extremely common, function names are almost always stripped. The name of a function is usually quite a strong indicator of likely types and names in a function, for example a function called `read_from_buf` will probably take `buffer` and `size` as arguments. Future researchers should investigate ways to lift this assumption.

This thesis contributes in the following ways:

1. It demonstrates that viewing the problem of postprocessing decompiler output as an instance of translation allows the effective application of techniques from the domain of natural language translation.

2. It introduces a novel technique for generating input/output examples suitable for training natural language translation models on decompiled code.
3. It presents a technique that automatically renames variables in decompiled code by leveraging the naturalness of code.
4. It presents a black-box technique for postprocessing decompiled code with a Transformer-based neural network to recommend user-created variable types.
5. It produces two datasets suitable for training and evaluating models of decompiled code.
6. It identifies specific challenges with automated fitness functions for measuring the effectiveness of these techniques.
7. It provides a human study protocol for measuring the impact of decompiler augmentations on user performance.
8. It describes the results of a human study testing the effectiveness of both techniques.

Overall, this thesis identifies difficulties experienced by users of decompilers and proposes approaches to automatically augment their output. It also identifies the challenges of automatically evaluating the effectiveness of these models and motivates future research. I hope that the techniques proposed here can lower the difficulty of interacting with decompilers to allow current reverse engineers to use their time more productively and also enable more computer scientists to become reverse engineers.

Appendix A

User Study

A.1 `array_extract_element_klen`

A.1.1 Hex-Rays

For this task, users were asked the following questions:

1. If `a1 + 8` points to an array and the `array_get_index` call on line 8 returns an index, what is the purpose of the `if` and `memmove` on lines 13-17?
2. Under the same assumptions (`a1 + 8` points to an array and the `array_get_index` call on line 8 returns an index) what does this function return?

```
 1  __int64 __fastcall array_extract_element_klen(__int64 a1, __int64 a2,
        unsigned int a3)
 2  {
 3    unsigned int i; // [rsp+24h] [rbp-1Ch]
 4    int index; // [rsp+28h] [rbp-18h]
 5    unsigned int v6; // [rsp+2Ch] [rbp-14h]
 6    __int64 v7; // [rsp+30h] [rbp-10h]
 7
 8    index = array_get_index(a1, a2, a3);
 9    if ( index < 0 )
10      return 0LL;
11    v7 = *(_QWORD *)(8LL * index + *(_QWORD *)(a1 + 8));
12    v6 = --*(_DWORD *)(a1 + 16);
13    if ( v6 != index )
14      memmove(
15        (void *)(8LL * index + *(_QWORD *)(a1 + 8)),
16        (const void *)(8LL * index + *(_QWORD *)(a1 + 8) + 8),
17        8LL * (v6 - index));
18    if ( v7 != *(_QWORD *)(8LL * v6 + *(_QWORD *)a1) )
19    {
20      for ( i = 0; v7 != *(_QWORD *)(8LL * i + *(_QWORD *)a1); ++i )
21        ;
22      *(_QWORD *)(*(_QWORD *)a1 + 8LL * i) = *(_QWORD *)(*(_QWORD *)a1 + 8LL
          * v6);
23    }
```

```
  *(_QWORD *)(8LL * v6 + *(_QWORD *)a1) = 0LL;
  return v7;
}
```

A.1.2 DIRTY

For this task, users were provided with the definition of the `array_t_0` type:

```
struct array_t_0 {
  char *pointer;
  unsigned int size;
  unsigned int next;
  unsigned int item_size;
}
```

and asked the following questions:

1. If `array->size` points to an array and the `array_get_index` call on line 8 returns an index, what is the purpose of the `if` and `memmove` on lines 13-17?
2. Under the same assumptions (`array->size` points to an array and the `array_get_index` call on line 8 returns an index) what does this function return?

```
1  char *__fastcall array_extract_element_klen(array_t_0 *array, void *key,
     int index)
2  {
3    unsigned int i; // [rsp+24h] [rbp-1Ch]
4    int indexa; // [rsp+28h] [rbp-18h]
5    int ret; // [rsp+2Ch] [rbp-14h]
6    char *next; // [rsp+30h] [rbp-10h]
7
8    indexa = array_get_index((__int64)array, (__int64)key, index);
9    if ( indexa < 0 )
10     return 0LL;
11   next = *(char **)(8LL * indexa + *(_QWORD *)&array->size);
12   ret = --array->item_size;
13   if ( ret != indexa )
14     memmove(
15       (void *)(8LL * indexa + *(_QWORD *)&array->size),
16       (const void *)(8LL * indexa + *(_QWORD *)&array->size + 8),
17       8LL * (unsigned int)(ret - indexa));
18   if ( next != *(char **)&array->pointer[8 * ret] )
19   {
20     for ( i = 0; next != *(char **)&array->pointer[8 * i]; ++i )
21       ;
22     *(_QWORD *)&array->pointer[8 * i] = *(_QWORD *)&array->pointer[8 * ret
         ];
23   }
24   *(_QWORD *)&array->pointer[8 * ret] = 0LL;
25   return next;
26 }
```

A.2 buffer_append_path_len

A.2.1 Hex-Rays

For this task, users were asked the following questions:

1. If this function is called in the following way, what will the value of `v3` be at the end of this function?

   ```
   /* "GNU" is equivalent to:
    * {0x47, 0x4E, 0x55, 0x00}
    * or
    * {71, 78, 85, 0}
    */
   const char *str = "GNU";
   buffer_append_path_len(a1, str, a2);
   ```

   You can make the following assumptions:

   - `a1` is properly initialized.
   - The call to `buffer_string_prepare_append` on line 11 has no side effects and returns successfully.
   - The value pointed to at `(dest - 1)` after the call on line 11 is 20.
   - The value pointed to at `(a1 + 8)` is 1.

2. If this function is called in the following way, what value will `src` point to in the `memcpy` on line 34? **Note: this is different than the call in the previous question**:

   ```
   /* "/usr/bin" is equivalent to:
    * {0x2F, 0x75, 0x73, 0x72, 0x2F, 0x62, 0x69, 0x6E, 0x00}
    * or
    * {47,  117, 115, 114, 47, 98, 105, 110, 0}
    */
   const char *str = "/usr/bin";
   buffer_append_path_len(a1, str, 5);
   ```

   You can make the following assumptions:

   - `a1` is properly initialized.
   - The call to `buffer_string_prepare_append` on line 11 has no side effects and returns successfully.
   - The value pointed to at `(dest - 1)` after the call on line 11 is 20.
   - The value pointed to at `(a1 + 8)` is 2.

```
1  void *__fastcall buffer_append_path_len(__int64 a1, _BYTE *a2, size_t a3)
2  {
3    _BOOL4 v3; // eax
4    char *v4; // rax
5    size_t n; // [rsp+8h] [rbp-28h]
6    void *src; // [rsp+10h] [rbp-20h]
7    char *dest; // [rsp+28h] [rbp-8h]
8
```

```
  src = a2;
  n = a3;
  dest = (char *)buffer_string_prepare_append(a1, a3 + 1);
  v3 = n && *a2 == 47;
  if ( *(_DWORD *)(a1 + 8) > 1u && *(dest - 1) == 47 )
  {
    if ( v3 )
    {
      src = a2 + 1;
      --n;
    }
  }
  else
  {
    if ( !*(_DWORD *)(a1 + 8) )
      *(_DWORD *)(a1 + 8) = 1;
    if ( !v3 )
    {
      v4 = dest++;
      *v4 = 47;
      ++*(_DWORD *)(a1 + 8);
    }
  }
  *(_DWORD *)(a1 + 8) += n;
  dest[n] = 0;
  return memcpy(dest, src, n);
}
```

A.2.2 DIRTY

For this task, users were presented with the definition of the **SSL** type:

```
typedef SSL {
  int version;
  int type;
  const SSL_METHOD *method;
  // ...
}
```

and asked the following questions:

1. If this function is called in the following way, what will the value of **v3** be at the end of this function?

   ```
   /* "GNU" is equivalent to:
    * {0x47, 0x4E, 0x55, 0x00}
    * or
    * {71, 78, 85, 0}
    */
   const char *str = "GNU";
   buffer_append_path_len(a1, str, a2);
   ```

   You can make the following assumptions:

- `s` is properly initialized.
- The call to `buffer_string_prepare_append` on line 11 has no side effects and returns successfully.
- The value pointed to at `(p - 1)` after the call on line 11 is 20.
- The value of `s->method` is 1.

2. If this function is called in the following way, what value will `stra` point to in the `memcpy` on line 34? **Note: this is different than the call in the previous question**:

```
/* "/usr/bin" is equivalent to:
 * {0x2F, 0x75, 0x73, 0x72, 0x2F, 0x62, 0x69, 0x6E, 0x00}
 * or
 * {47,  117, 115, 114, 47, 98, 105, 110, 0}
 */
const char *str = "/usr/bin";
buffer_append_path_len(a1, str, 5);
```

You can make the following assumptions:

- `s` is properly initialized.
- The call to `buffer_string_prepare_append` on line 11 has no side effects and returns successfully.
- The value pointed to at `(p - 1)` after the call on line 11 is 20.
- The value of `s->method` is 2.

```
1  void *__fastcall buffer_append_path_len(SSL *s, const char *str, size_t n)
2  {
3    _BOOL4 v3; // eax
4    char *v4; // rax
5    size_t l; // [rsp+8h] [rbp-28h]
6    const char *stra; // [rsp+10h] [rbp-20h]
7    char *p; // [rsp+28h] [rbp-8h]
8
9    stra = str;
10   l = n;
11   p = (char *)buffer_string_prepare_append(s, n + 1);
12   v3 = l && *str == 47;
13   if ( s->method > 1u && *(p - 1) == 47 )
14   {
15     if ( v3 )
16     {
17       stra = str + 1;
18       --l;
19     }
20   }
21   else
22   {
23     if ( !s->method )
24       s->method = 1;
25     if ( !v3 )
26     {
      v4 = p++;
      *v4 = 47;
      ++s->method;
    }
  }
  s->method += 1;
  p[l] = 0;
  return memcpy(p, stra, l);
}
```

A.3 `postorder`

A.3.1 Hex-Rays

For this example, users were asked the following questions

1. After the loop on lines 7-13, what does `a1` point to?
2. This function traverses a tree and applies a function at each node, passing additional information for the function to use. Which argument corresponds to each?

	Tree	Comparison function	Additional information
`_QWORD *a1`	○	○	○
`__int64(__fastcall *a2)(__int64, _QWORD *)`	○	○	○
`__int64 a3`	○	○	○

3. (Optional) Informally, how did you reach your conclusion?

```
1  __int64 __fastcall postorder(_QWORD *a1, __int64 (__fastcall *a2)(__int64,
       _QWORD *), __int64 a3)
2  {
3    unsigned int v5; // [rsp+2Ch] [rbp-14h]
4    _QWORD *v7; // [rsp+38h] [rbp-8h]
5
6  LABEL_5:
7    while ( a1[1] || a1[2] )
8    {
9      if ( a1[1] )
10       a1 = a1[1];
11     else
12       a1 = a1[2];
13   }
14   while ( 1 )
15   {
16     v5 = a2(a3, a1);
17     if ( v5 )
18       return v5;
19     if ( !*a1 )
20       return 0LL;
```

```
      v7 = a1;
      a1 = *a1;
      if ( v7 != a1[2] && a1[2] )
      {
        a1 = a1[2];
        goto LABEL_5;
      }
    }
}
```

A.3.2 DIRTY

For this example, users were asked the following questions

1. After the loop on lines 7-13, what does `t` point to? For reference, `tree234` and `cmpfn234` are:

   ```
   typedef int (*cmpfn234) (void *, void *);

   typedef struct tree234 {
     node234 *root;
     cmpfn234 cmp;
   }
   ```

2. This function traverses a tree and applies a function at each node, passing additional information for the function to use. Which argument corresponds to each?

	Tree	Comparison function	Additional information
`cmpfn234 cmp`	○	○	○
`void *e`	○	○	○
`tree234 *t`	○	○	○

3. (Optional) Informally, how did you reach your conclusion?

```
 1  __int64 __fastcall postorder(tree234 *t, void *e, cmpfn234 cmp)
 2  {
 3    int ret; // [rsp+2Ch] [rbp-14h]
 4    node234 *n; // [rsp+38h] [rbp-8h]
 5
 6  LABEL_5:
 7    while ( t->cmp || t[1].root )
 8    {
 9      if ( t->cmp )
10        t = t->cmp;
11      else
12        t = t[1].root;
13    }
14    while ( 1 )
15    {
16      ret = (e)(cmp, t);
17      if ( ret )
```

```
      return ret;
    if ( !t->root )
      return 0LL;
    n = t;
    t = t->root;
    if ( n != t[1].root && t[1].root )
    {
      t = t[1].root;
      goto LABEL_5;
    }
  }
}
```

A.4 twos_complement

A.4.1 Hex-Rays

For this example, users were asked the following questions:

1. Which argument of this function controls when the loop on lines 16-25 terminates?
2. If **a4** is passed the value **0xFF**, what will the value of **(savedregs - 1)** be when execution reaches line 16 for the first time?

```
1  __int64 __fastcall twos_complement(__int64 a1, __int64 a2, __int64 a3,
      char a4)
2  {
3    __int64 result; // rax
4    __int64 savedregs; // [rsp+0h] [rbp+0h]
5
6    *(&savedregs - 3) = a1;
7    *(&savedregs - 4) = a2;
8    *(&savedregs - 5) = a3;
9    *((_BYTE *)&savedregs - 44) = a4;
10   *((_DWORD *)&savedregs - 1) = *((_BYTE *)&savedregs - 44) & 1;
11   if ( *(&savedregs - 5) )
12   {
13     *(&savedregs - 3) += *(&savedregs - 5);
14     *(&savedregs - 4) += *(&savedregs - 5);
15   }
16   while ( 1 )
17   {
18     result = *(&savedregs - 5);
19     *(&savedregs - 5) = result - 1;
20     if ( !result )
21       break;
22     *((_DWORD *)&savedregs - 1) += (unsigned __int8)(*((_BYTE *)&savedregs
         - 44) ^ *(_BYTE *)--*(&savedregs - 4));
23     *(_BYTE *)--*(&savedregs - 3) = *((_DWORD *)&savedregs - 1);
24     *((_DWORD *)&savedregs - 1) >>= 8;
25   }
26   return result;
27 }
```

A.4.2 DIRTY

For this example, users were asked the following questions:

1. Which argument of this function controls when the loop on lines 18-23 terminates?
2. If `i2` is passed the value `0xFF`, what will the value of `v8` be when execution reaches line 18 for the first time?

```
 1 void __fastcall twos_complement(void *p1, void *p2, void *data, int i2)
 2 {
 3   __int64 k; // [rsp+4h] [rbp-28h]
 4   __int64 j; // [rsp+Ch] [rbp-20h]
 5   __int64 i; // [rsp+14h] [rbp-18h]
 6   int v8; // [rsp+28h] [rbp-4h]
 7   unsigned int v9; // [rsp+28h] [rbp-4h]
 8 
 9   i = (__int64)p1;
10   j = (__int64)p2;
11   k = (__int64)data;
12   v8 = i2 & 1;
13   if ( data )
14   {
15     i = (__int64)p1 + (_QWORD)data;
16     j = (__int64)p2 + (_QWORD)data;
17   }
18   while ( k-- )
19   {
20     v9 = (unsigned __int8)(i2 ^ *(_BYTE *)--j) + v8;
21     *(_BYTE *)--i = v9;
22     v8 = v9 >> 8;
23   }
24 }
```

A.5 General Questions About the Code

In addition to the above questions, participants were also asked the following questions about each example:

1. For each type and name of each argument in an example, they were asked to rate: "The type and name of each argument _____ understanding:"
   - Prevented
   - Hindered
   - Did not affect
   - Improved
   - Provided immediate

2. For each entire example, they were asked to rate their feelings about the following statements on a 5 point scale ranging from Strongly Disagree to Strongly Agree:
    - The code was easily readable.
    - It was easy to understand what this code did.
    - This code looks similar to the way I write code.
    - This code is well structured.
    - I am sure that I correctly understood what this code does.
    - It was easy to understand what the variables mean.
    - I trust that the decompiled code is correct.
    - I would rather analyze this code than the assembly code.

A.6 Exit Interview

At the end of the study, we asked the following questions in an exit interview:

- What is your age?
- What is your gender?
- What is your highest level of education?
- What is your current employment status?
- How many years of computer-science education do you have?
- How many years of general coding experience do you have?
- How many years of C coding experience do you have?
- How many years of professional coding experience do you have?
- How many years of (professional and amateur) reverse engineering experience do you have?
- How many years of experience do you have using decompilers?
- Is there other information about your experience with decompilers, reverse engineering, or the questions in this survey that you would like to share with us?
- Do you have any general comments or feedback about the survey?

Bibliography

[1] Uroburos: Highly complex espionage software with Russian roots. Technical report, G Data SecurityLabs, 2014. 1, 2.1.1

[2] Daniel Adiwardana, Minh-Thang Luong, David R. So, Jamie Hall, Noah Fiedel, Romal Thoppilan, Zi Yang, Apoorv Kulshreshtha, Gaurav Nemade, Yifeng Lu, and Quoc V. Le. Towards a human-like open-domain chatbot. *arXiv preprint arXiv:2001.09977*, 2020. 2.1.3, 4.1.2

[3] Aysh Al-Hroob, Ayad Tareq Imam, and Rawan Al-Hesia. The use of artificial neural networks for extracting actions and actors from requirements document. *Information and Software Technology*, 2020. 2.2.3

[4] Miltiadis Allamanis. The adverse effects of code duplication in machine learning models of code. In *International Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, Onward! ACM, 2019. 3.3.1

[5] Miltiadis Allamanis, Earl T Barr, Christian Bird, and Charles Sutton. Learning natural coding conventions. In *Symposium on the Foundations of Software Engineering*, FSE, pages 281–293, 2014. 2.2.3

[6] Miltiadis Allamanis, Earl T. Barr, Christian Bird, and Charles Sutton. Suggesting accurate method and class names. In *Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering*, ESEC/FSE, pages 38–49, 2015. 2.2.3

[7] Miltiadis Allamanis, Hao Ping, and Charles Sutton. A convolutional attention network for extreme summarization of source code. In *International Conference on Machine Learning*, PMLR, 2016. 2.2.3

[8] Miltiadis Allamanis, Earl T. Barr, Premkumar Devanbu, and Charles Sutton. A survey of machine learning for big code and naturalness. *ACM Computing Surveys (CSUR)*, 51(4): 81, 2018. 1, 2.2.3, 3.1.2

[9] Miltiadis Allamanis, Marc Brockschmidt, and Mahmoud Khademi. Learning to represent programs with graphs. In *International Conference on Learning Representations*, ICLR, 2018. 2.2.3, 3.1.2, 3.3

[10] Uri Alon, Meital Zilberstein, Omer Levy, and Eran Yahav. A general path-based representation for predicting program properties. In *Programming Language Design and Implementation*, PLDI, pages 404–419, 2018. 2.2.1, 3

[11] Uri Alon, Omer Levy, and Eran Yahav. code2seq: Generating sequences from structured representations of code. In *ICLR*, 2019. 2.2.3

[12] Dzmitry Bahdanau, Kyunghyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. In *International Conference on Learning Representations*, ICLR, 2015. 2.1.3, 4.1.1

[13] D. Balbinot and L. Petrone. Decompilation of Polish code in BASIC. In *Rivista di Informatica*, 1979. 2.1.2

[14] Matej Balog, Alexander L. Gaunt, Marc Brockschmidt, Sebastian Nowozin, and Daniel Tarlow. DeepCoder: Learning to write programs. In *International Conference on Learning Representations*, ICLR, 2017. 2.2.3

[15] Penny Barbe. The PILER system of computer program translation. Technical report, Probe Consultants Inc., 1974. 2.1.2

[16] Antoine Barbez, Foutse Khomh, and Yann-Gaël Guéhéneuc. Deep learning anti-patterns from code metrics history. In *International Conference on Software Maintenance and Evolution*, ICSME, 2017. 2.2.3

[17] R. Bavishi, M. Pradel, and K. Sen. Context2Name: A deep learning-based approach to infer natural variable names from usage contexts. Technical report, TU Darmstadt, Department of Computer Science, November 2017. 2.2.1, 3

[18] Binutils. objdump, 2019. URL `https://www.gnu.org/software/binutils/`. 1, 2.1.1

[19] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*, 2020. 2.1.3, 4.1.2

[20] Rudy Bunel, Matthew J. Hausknecht, Jacob Devlin, Rishabh Singh, and Pushmeet Kohli. Leveraging grammar and reinforcement learning for neural program synthesis. In *International Conference on Learning Representations*, ICLR, 2018. 2.2.3

[21] Raymond P. L. Buse and Westley Weimer. Learning a metric for code readability. *Transactions on Software Engineering*, 36(4):546–558, jul 2010. 5.1

[22] Juan Caballero and Zhiqiang Lin. Type inference on executables. *ACM Computing Surveys*, 2016. 2.1.2

[23] Juan Caballero, Noah M. Johnson, Stephen McCamant, and Dawn Song. Binary code extraction and interface identification for security applications. In *Network and Distributed System Security Symposium*, NDSS, 2010. 2.1.2

[24] Kevin Cao and Kevin Leach. Revisiting deep learning for variable type recovery. *arXiv:2304.03854*, 2023. 6

[25] Kathy Charmaz. *Constructing Grounded Theory: A Practical Guide Through Qualitative*

Analysis. SAGE Publications, 2006. 5.3.2

[26] Prantik Chatterjee, Abhijit Chatterjee, José Campos, Rui Abreu, and Subhajit Roy. Diagnosing software faults using multiverse analysis. In *International Joint Conferences on Artificial Intelligence*, IJCAI, 2020. 2.2.3

[27] Chunyang Chen, Ting Su, Gouzhu Meng, Zhenchang Xing, and Yang Liu. From UI design image to GUI skeleton: A neural machine translator to bootstrap mobile GUI implementation. In *International Conference on Software Engineering*, ICSE, 2018. 2.2.3

[28] Jeishan Chen, Chunyang Chen, Zhenchang Xing, Xin Xia, Liming Zhu, John Grundy, and Jinshui Wang. Wireframe-based UI design search through image autoencoder. *ACM Transactions on Software Engineering and Methodology*, 2020. 2.2.3

[29] Ligeng Chen, Zhongling He, and Bing Mao. CATI: Context-assisted type inference from stripped binaries. In *International Conference on Dependable Systems and Networks*, DSN, 2020. 2.1.2, 2.2.2

[30] Qibin Chen, Jeremy Lacomis, Edward J. Schwartz, Claire Le Goues, Graham Neubig, and Bogdan Vasilescu. Augmenting decompiler output with learned variable names and types. In *USENIX Security Symposium*, USENIX, 2022. 1.3

[31] Qibin Chen, Jeremy Lacomis, Edward J. Schwartz, Graham Neubig, Bogdan Vasilescu, and Claire Le Goues. VarCLR: Variable semantic representation pre-training via contrastive learning. In *International Conference on Software Engineering*, ICSE, 2022. 1.2, 5, 5.4

[32] Xinyun Chen, Chang Liu, and Dawn Song. Execution-guided neural program synthesis. In *International Conference on Learning Representations*, ICLR, 2018. 2.2.3

[33] Xinyun Chen, Chang Liu, and Dawn Song. Towards synthesizing complex programs from input-output examples. In *International Conference on Learning Representations*, ICLR, 2018. 2.2.3

[34] Shasha Cheng, Xuefeng Yan, and Arif Ali Khan. A similarity integration method based information retrieval and word embedding in bug localization. In *International Conference on Software Quality, Reliability and Security*, QRS, 2020. 2.2.3

[35] Rewon Child, Scott Gray, Alec Radford, and Ilya Sutskever. Generating long sequences with sparse transformers. *arXiv preprint arXiv:1904.10509*, 2019. 4.1.2

[36] Kyunghyun Cho, Bart van Merriënboer, Caglar Gulcehre, Dzmitry Bahdanau, Fethi Bougares, Holger Schwenk, and Yoshua Bengio. Learning phrase representations using RNN encoder-decoder for statistical machine translation. In *Conference on Empirical Methods in Natural Language Processing*, EMNLP, 2014. 2.1.3, 3.1.2, 4.1.1

[37] Kyunghyun Cho, Aaron Courville, and Yoshua Bengio. Describing multimedia content using attention-based encoder-decoder networks. *IEEE Transactions on Multimedia*, 17 (11):1875–1886, 2015. 3.1.3

[38] Cristina Cifuentes. *Reverse Compilation Techniques*. PhD thesis, Queensland University of Technology, 1994. 2.1.2, 2.1.2

[39] Adelina Ciurumelea, Sebastian Proksch, and Harald C. Gall. Suggesting comment com-

pletions for Python using neural language models. In *International Conference on Software Analysis, Evolution and Reegnineering*, SANER, 2020. 2.2.3

[40] Anthony Cozzie, Frank Stratton, Hui Xue, and Samuel T. King. Digging for data structures. In *USENIX Security Symposium*, USENIX, 2008. 2.1.2

[41] Yaniv David, Uri Alon, and Erah Yahav. Neural reverse engineering of stripped binaries, 2019. 2.2.1, 3

[42] Jayati Deshmukh, Annervaz K M, Sanjay Podder, Shubhashis Sengupta, and Neville Dubash. Towards accurate duplicate bug retrieval using deep learning techniques. In *International Conference on Software Maintenance and Evolution*, ICSME, 2017. 2.2.3

[43] Premkumar Devanbu. New initiative: The naturalness of software. In *International Conference on Software Engineering*, ICSE, pages 543–546, 2015. 1, 2.2.3, 3, 4.3

[44] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Annual Conference of the North American Chapter of the Association for Computational Linguistics*, NAACL-HLT, pages 4171–4186, 2019. 2.1.3, 4.1.2, 4.2.4, 4.3

[45] Elizabeth Dinella, Hanjun Dai, Ziyang Li, Mayur Naik, Le Song, and Ke Wang. Hoppity: Learning graph transformations to detect and fix bugs in programs. In *International Conference on Learning Representations*, ICLR, 2020. 2.2.3

[46] Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale. *arXiv preprint arXiv:2010.11929*, 2020. 2.1.3, 4.1.2

[47] Luke Dramko, Jeremy Lacomis, Pengcheng Yin, Edward J. Schwartz, Miltiadis Allamanis, Graham Neubig, Bogdan Vasilescu, and Claire Le Goues. DIRE and its data: Neural decompiled variable renamings with respect to software class. *Transactions on Software Engineering and Methodology*, 2022. 1.3

[48] Lukas Durfina, Jakub Kroustek, and Petr Zemek. PsybOt malware: A step-by-step decompilation case study. In *Working Conference on Reverse Engineering*, WCRE, pages 449–456, 2013. 1, 2.1.1

[49] Michael J. Eager. Introduction to the DWARF debugging format, April 2012. URL `http://www.dwarfstd.org/doc/Debugging%20using%20DWARF-2012.pdf`. 2.1.2, 3.2

[50] Zachary Eberhart, Alexander LeClair, and Collin McMillan. Automatically extracting subroutine summary descriptions from unstructured comments. In *International Conference on Software Analysis, Evolution and Reegnineering*, SANER, 2020. 2.2.3

[51] Rukayat Ayomide Erinfolami and Aravind Prakash. Devil is virtual: Reversing virtual inheritance in C++ binaries. In *Proceedings of the ACM Conference on Computer and Communications Security*, CCS, pages 133–148, 2020. 4.3

[52] Javier Escalada, Ted Scully, and Francisco Ortin. Improving type information inferred by

decompilers with supervised machine learning. *arXiv preprint arXiv:2101.08116*, 2021. 2.2.2, 4

[53] Sarah Fakhoury, Venera Arnaoudova, Cedric Noiseux, Foutse Khomh, and Guiliano Antoniol. Keep it simple: Is deep learning good for linguistic smell detection? In *International Conference on Software Analysis, Evolution and Reegnineering*, SANER, 2018. 2.2.3

[54] William Fedus, Barret Zoph, and Noam Shazeer. Switch transformers: Scaling to trillion parameter models with simple and efficient sparsity. *arXiv preprint arXiv:2101.03961*, 2021. 2.1.3, 4.1.2

[55] Zachary P. Fry, Bryan Landau, and Westley Weimer. A human study of patch maintainability. In *International Symposium on Software Testing and Analysis*, ISSTA, pages 177–187, July 2012. 1.2, 5.2, 5.2.1

[56] Cheng Fu, Huili Chen, Haolan Liu, Xinyun Chen, Yuandong Tian, Farinaz Koushanfar, and Jishen Zhao. Coda: An end-to-end neural program decompiler. In *Conference on Neural Information Processing Systems*, NeurIPS, 2019. 2.2.2

[57] Cuiyun Gao, Jichuan Zeng, Xin Xia, David Lo, Michael R. Lyu, and Irwin King. Automating app review response generation. In *International Conference on Automated Software Engineering*, ASE, 2019. 2.2.3

[58] Sa Gao, Chunyang Chen, Zhenchang Xing, Yukun Ma, Wen Song, and Shang-Wei Lin. A neural model for method name generation from functional description. In *International Conference on Software Analysis, Evolution and Reegnineering*, SANER, 2019. 2.2.3

[59] Edward M. Gellenbeck and Curtis R. Cook. An investigation of procedure and variable names as beacons during program comprehension. Technical report, Oregon State University, 1991. 1, 2.1.2

[60] Andrew Gelman and Jennifer Hill. *Data Analysis Using Regression and Multilevel/Hierarchical Models*. Cambridge University Press, 2006. 5.3.2

[61] Ghidra. The Ghidra decompiler, 2019. URL `https://ghidra-sre.org/`. 1, 2.1.1, 2.1.2, 3.4, 5.5

[62] Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl. Neural message passing for quantum chemistry. In *Proceedings of the 34th International Conference on Machine Learning-Volume 70*, pages 1263–1272. JMLR. org, 2017. 3.1.2, 3.1.2

[63] David Gros, Hariharan Sezhiyan, Prem Devanbu, and Zhou Yu. Code to comment "Translation": Data, metrics, baselining & evaluation. In *International Conference on Automated Software Engineering*, ASE, 2020. 2.2.3

[64] Ilfak Guilfanov. A simple type system for program reengineering. In *Working Conference on Reverse Engineering*, WCRE, 2001. 2.1.2

[65] Kavi Gupta, Peter Ebert Christensen, Xinyun Chen, and Dawn Song. Synthesize, execute, and debug: Learning to repair for neural program synthesis. In *Conference on Neural Information Processing Systems*, NeurIPS, 2020. 2.2.3

[66] Rahul Gupta, Aditya Kanade, and Shirish Shevade. Neural attribution for semantic bug-

localization in student programs. In *Conference on Neural Information Processing Systems*, NeurIPS, 2019. 2.2.3

[67] Maurice Howard Halstead. *Machine-Independent Computer Programming*, chapter 11, pages 143–150. Spartan Books, 1962. 2.1.2, 2.1.2

[68] Jingxuan He, Pesho Ivanov, Petar Tsankov, Veselin Raychev, and Martin Vechev. DEBIN: Predicting debug information in stripped binaries. In *Conference on Computer and Communications Security*, CCS, 2018. 2.2.1, 3, 3.3.4, 3.3.4

[69] Kaiming He, Xiangyu Zhang, Shaoqing Ren, and Jian Sun. Deep residual learning for image recognition. In *IEEE Conference on Computer Vision and Pattern Recognition*, CVPR, pages 770–778, 2016. 2.1.3, 4.1.2

[70] Vincent J. Hellendoorn, Christian Bird, Earl T. Barr, and Miltiadis Allamanis. Deep learning type inference. In *Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering*, ESEC/FSE, 2018. 2.2.2

[71] Hex-Rays. The Hex-Rays decompiler, 2019. URL `https://www.hex-rays.com/products/decompiler/`. 1, 2.1.2, 4.2.2, 4.2.3, 5.2

[72] Abram Hindle, Earl T Barr, Zhendong Su, Mark Gabel, and Premkumar Devanbu. On the naturalness of software. In *International Conference on Software Engineering*, ICSE, pages 837–847. IEEE, 2012. 1, 2.2.3, 3, 4.3

[73] Sepp Hochreiter and Jürgen Schmidhuber. Long short-term memory. *Neural Computation*, 9(8):1735–1780, 1997. 2.1.3, 4.1.2

[74] G.L. Hopwood. *Decompilation*. PhD thesis, University of California, Irvine, 1978. 2.1.2

[75] Xing Hu, Ge Li, Xin Xia, David Lo, and Zhi Jhi. Deep code comment generation. In *International Conference on Program Comprehension*, ICPC, 2018. 2.2.3

[76] Xuan Huo and Ming Li. Enhancing the unified features to locate buggy files by exploiting the sequential nature of source code. In *International Joint Conferences on Artificial Intelligence*, IJCAI, 2017. 2.2.3

[77] Xuan Huo, Ming Li, and Zhi-Hua Zhou. Learning unified features from natural and programming languages for locating buggy source code. In *International Joint Conferences on Artificial Intelligence*, IJCAI, 2016. 2.2.3

[78] Xuan Huo, Ferdian Thung, Ming Li, David Lo, and Shu-Ting Shi. Deep transfer bug localization. *Transactions on Software Engineering*, 2019. 2.2.3

[79] Abū Yūsuf Ya'qūb ibn 'Isḥāq aṣ-Ṣabbāḥ al Kindī. *Manuscript on Deciphering Cryptographic Messages*. self, ca. 850. 1

[80] IDA. Ida, 2023. URL `https://www.hex-rays.com/products/ida/`. 1, 2.1.1, 2.1.2

[81] Alan Jaffe, Jeremy Lacomis, Edward J. Schwartz, Claire Le Goues, and Bogdan Vasilescu. Meaningful variable names for decompiled code: A machine translation approach. In *International Conference on Program Comprehension*, ICPC, pages 20–30, May 2018. 3, 3.1.1, 3.2, 3.2, 3.3.4

[82] Ziwei Ji, Nayeon Lee, Rita Frieske, Tiezheng Yu, Dan Su, Yan Xu, Etsuko Ishii, Ye Jin Bang, Andrea Madotto, and Pascale Fung. Survey of hallucination in natural language generation. *ACM Computing Surveys*, 2023. 5.3.6

[83] Deborah S. Katz, Jason Ruchti, and Eric Schulte. Using recurrent neural networks for decompilation. In *International Conference on Software Analysis, Evolution and Reegnineering*, SANER, pages 346–356, 2018. 2.1.2, 2.2.2

[84] Omer Katz, Yuval Olshaker, Yoav Goldberg, and Eran Yahav. Towards neural decompilation. *arXiv preprint arXiv:1905.08325*, 2019. 2.2.2

[85] Philipp Koehn. *Statistical machine translation*. Cambridge University Press, 2009. 3

[86] Jinkyu Koo, Charitha Saumya, Milind Kulkarni, and Saurabh Bagchi. Pyse: Automatic worst-case test generation by reinforcement learning. In *International Conference on Software Testing, Verification, and Validation*, ICST, 2019. 2.2.3

[87] Taku Kudo and John Richardson. SentencePiece: A simple and language independent subword tokenizer and detokenizer for neural text processing. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing: System Demonstrations*, pages 66–71, 2018. 3.1.2

[88] Jeremy Lacomis, Pengcheng Yin, Edward J. Schwartz, Miltiadis Allamanis, Claire Le Goues, Graham Neubig, and Bogdan Vasilescu. DIRE: A neural approach to decompiled identifier naming. In *International Conference on Automated Software Engineering*, ASE, 2019. 1.3, 4.1.3, 4.1.5

[89] Ngoc Lam, Anh Tuan Nguyen, Hoan Anh Nguyen, and Tien N. Nguyen. Bug localization with combination of deep learning and information retrieval. In *International Conference on Program Comprehension*, ICPC, 2017. 2.2.3

[90] James Langdell. Phoenix says its BIOS may foil IBM's lawsuits. *PC Mag*, page 56, July 1994. 6

[91] Dawn Lawrie, Henry Feild, and David Binkley. Quantifying identifier quality: An analysis of trends. *Empirical Software Engineering*, 2006. 5

[92] Dawn Lawrie, Christopher Morrell, Henry Feild, and David Binkley. What's in a name? A study of identifiers. In *International Conference on Program Comprehension*, ICPC, pages 3–12, 2006. 1, 2.1.2

[93] Tien-Duy B. Le and David Lo. Deep specification mining. In *International Symposium on Software Testing and Analysis*, ISSTA, 2018. 2.2.3

[94] Alexander LeClair, Siyuan Jiang, and Collin McMillan. A neural model for generation natural language summaries of program subroutines. In *International Conference on Software Engineering*, ICSE, 2019. 2.2.3

[95] Alexander LeClair, Sakib Haque, Lingfei Wu, and Collin McMillan. Improved code summarization via a graph neural network. In *International Conference on Program Comprehension*, ICPC, 2020. 2.2.3

[96] JongHyup Lee, Thanassis Avgerinos, and David Brumley. TIE: Principled reverse engineering of types in binary programs. In *Network and Distributed System Security Sympo-*

sium, NDSS, 2011. 2.1.2, 2.2.2, 4, 4.2.3

[97] Mike Lewis, Yinhan Liu, Naman Goyal, Marjan Ghazvininejad, Abdelrahman Mohamed, Omer Levy, Veselin Stoyanov, and Luke Zettlemoyer. BART: Denoising sequence-to-sequence pre-training for natural language generation, translation, and comprehension. In *Annual Meeting of the Association for Computational Linguistics*, ACL, pages 7871–7880, 2020. 2.1.3, 4.1.2

[98] Guangjie Li, Hui Liu, Jiahao Jin, and Qasim Umer. Deep learning based identification of suspicious return statements. In *International Conference on Software Analysis, Evolution, and Reengineering*, SANER, 2020. 2.2.3

[99] Jian Li, Yue Wang, Michael R. Lyu, and Irwin King. Code completion with neural attention and pointer networks. In *International Joint Conferences on Artificial Intelligence*, IJCAI, 2018. 2.2.3

[100] Mingyang Li, Lin Shi, Ye Yang, and Qing Wang. A deep multitask learning approach for requirements discovery and annotation from open forum. In *International Conference on Automated Software Engineering*, ASE, 2020. 2.2.3

[101] Mingyang Li, Ye Yang, Lin Shi, Qing Wang, Jun Hu, Xinhua Peng, Weimin Liao, and Guizhen Pi. Automated extraction of requirement entities by leveraging LSTM-CRF and transfer learning. In *International Conference on Software Maintenance and Evolution*, ICSME, 2020. 2.2.3

[102] Yang Li. Feature and variability extraction from natural language requirements specifications. In *International Systems and Software Product Line Conference*, SPLC, 2018. 2.2.3

[103] Yang Li, Sandro Schulze, and Gunter Saake. Reverse engineering variablity from requirement documents based on probabilistic relevance and word embedding. In *International Systems and Software Product Line Conference*, SPLC, 2018. 2.2.3

[104] Yangguang Li, Zhen Ming (Jack) Jiang, Heng Li, Ahmed E. Hassan, Cheng He, Ruirui Huang, Zhengda Zeng, Mian Wang, and Pinan Chen. Predicting node failures in an ultra-large-scale cloud computing platform: An AIOps solution. *ACM Transactions on Software Engineering and Methodology*, 2020. 2.2.3

[105] Yi Li, Shaohua Wang, Tien N. Nguyen, and Son Van Nguyen. Improving bug detection via context-based code representation learning and attention-based neural networks. *Proceedings of the ACM on Programming Languages*, (OOPSLA), 2019. 2.2.3

[106] Yujia Li, Daniel Tarlow, Marc Brockschmidt, and Richard Zemel. Gated graph sequence neural networks. *arXiv preprint arXiv:1511.05493*, 2015. 2.1.3

[107] Zijie Li, Long Zhang, Jun Yan, Jian Zhang, Zhenyu Zhang, and T. H. Tse. PEACEPACT: Prioritizing examples to accelerate perturbation-based acerversary generation for DNN classification testing. In *International Conference on Software Quality, Reliability and Security*, QRS, 2020. 2.2.3

[108] Yuding Liang and Kenny Zhu. Automatic generation of text descriptive comments for code blocks. In *AAAI Conference on Artificial Intelligence*, AAAI, 2018. 2.2.3

[109] Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. Automatic reverse engineering of data structures from binary execution. In *CERIAS Annual Security Symposium*, CERIAS, 2010. 2.1.2, 2.2.2, 4

[110] Fang Liu, Ge Li, Bolin Wei, Xin Xia, Zhiyi Fu, and Zhi Jin. A self-attentional neural architecture for code completion with multi-task learning. In *International Conference on Automated Software Engineering*, ASE, 2020. 2.2.3

[111] Kui Liu, Dongsun Kim, Tegawendé F. Bissyandé, Taeyoung Kim, Kisub Kim, Anil Koyuncu, Suntae Kim, and Yves Le Traon. Learning to spot and refactor inconsistent method names. In *International Conference on Software Engineering*, ICSE, 2019. 2.2.3

[112] Muyang Liu, Ke Li, and Tao Chen. DeepSQLi: Deep semantic learning for testing SQL injection. In *International Symposium on Software Testing and Analysis*, ISSTA, 2020. 2.2.3

[113] Peng Liu, Xiangyu Zhang, Marco Pistoia, Yunhui Zheng, Manoel Marques, and Lingfei Zeng. Automatic text input generation for mobile testing. In *International Conference on Software Engineering*, ICSE, 2017. 2.2.3

[114] Xiao Liu, Xiaoting Li, Rupesh Prajapati, and Dinghao Wu. Deepfuzz: Automatic generation of syntax valid C programs for fuzz testing. In *AAAI Conference on Artificial Intelligence*, AAAI, 2019. 2.2.3

[115] Zhe Liu, Chunyang Chen, Junjie Wang, Yuekai Huang, Jun Hu, and Qing Wang. Owl Eyes: Spotting UI display issues via visual understanding. In *International Conference on Automated Software Engineering*, ASE, 2020. 2.2.3

[116] Zhongxin Liu, Xin Xia, Meng Yan, and Shanping Li. Automating just-in-time comment updating. In *International Conference on Automated Software Engineering*, ASE, 2020. 2.2.3

[117] Thang Luong, Hieu Pham, and Christopher D. Manning. Effective approaches to attention-based neural machine translation. In *Conference on Empirical Methods in Natural Language Processing*, EMNLP, 2015. 3.1.3

[118] Aman Madaan, Alexander G. Shypula, Uri Alon, Milad Hashemi, Parthasarathy Ranganathan, Yiming Yang, Graham Neubig, and Amir Yazdanbakhsh. Learning performance-improving code edits. In *Deep Learning for Code Workshop*, DL4C, 2023. 2.2.3

[119] Alwin Maier, Hugo Gascon, Christian Wressnegger, and Konrad Rieck. TypeMiner: Recovering types in binary programs using machine learning. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*, DIMVA, 2019. 2.2.2, 4.2.3

[120] Alejandro N. Mayorkas. FY 2023 budget in brief. Technical report, U.S. Department of Homeland Security, 2023. 6

[121] Paul Michel and Graham Neubig. Extreme adaptation for personalized neural machine translation. In *Annual Meeting of the Association for Computational Linguistics (Short Papers)*, ACL, pages 312–318, 2018. 4.1.4

[122] Tomas Mikolov, Ilya Sutskever, Kai Chen, Greg Corrado, and Jeffery Dean. Distributed representations of words and phrases and their compositioniality. *Computing Research Repository (CoRR)*, abs/1310.4546, 2013. 5

[123] Kevin Patrick Moran, Carlos Bernal-Cárdenas, Michael Curcio, Richard Bonett, and Denys Poshyvanyk. Machine learning-based prototyping of graphical user interfaces for mobile apps. *Transactions on Software Engineering*, 2020. 2.2.3

[124] Alan Mycroft. Type-based decompilation. In *European Symposium on Programming/European Joint Conferences on Theory and Practice of Software*, ESOP/ETAPS, 1999. 2.1.2

[125] Shinichi Nakagawa, Paul C. D. Johnson, and Holger Schielzeth. The coefficient of determination R^2 and intra-class correlation coefficient from generalized linear mixed-effects models revisited and expanded. *Journal of the Royal Society Interface*, 2017. 5.3.2

[126] Ramesh Nallapati, Bowen Zhou, Caglar Gulcehre, and Bing Xiang. Abstractive text summarization using sequence-to-sequence RNNs and beyond. In *SIGNLL Conference on Computational Natural Language Learning*, CoNLL, pages 280–290, 2016. 2.1.3, 4.1.1

[127] Hermann Ney, Dieter Mergel, Andreas Noll, and Annedore Paeseler. A data-driven organization of the dynamic programming beam search for continuous speech recognition. In *International Conference on Acoustics, Speech, and Signal Processing*, ICASSP, 1987. 4.1.3

[128] Anh Tuan Nguyen, Trong Duc Nguyen, Hung Dang Phan, and Tien N. Nguyen. A deep neural network language model with contexts for source code. In *International Conference on Software Analysis, Evolution and Reegnineering*, SANER, 2018. 2.2.3

[129] Son Nguyen, Hung Phan, Trinh Le, and Tien N. Nguyen. Suggesting natural method names to check name consistencies. In *International Conference on Software Engineering*, ICSE, 2020. 2.2.3

[130] Matthew Noonan, Alexey Loginov, and David Cok. Polymorphic type inference for machine code. In *Conference on Programming Language Design and Implementation*, PLDI, pages 27–41, 2016. 2.2.2, 4.2.3

[131] Sajad Norouzi, Keyi Tang, and Yanshuai Cao. Code generation from natural language with less prior knowledge and more monolingual data. In *International Joint Conferences on Natural Language Processing*, IJCNLP, 2021. 2.2.3

[132] Irene Vlassi Pandi, Earl T Barr, Andrew D Gordon, and Charles Sutton. OptTyper: Probabilistic type inference by optimising logical and natural constraints. *arXiv preprint arXiv:2004.00348*, 2020. 2.2.2

[133] Kishore Papineni, Salim Roukos, Todd Ward, and Wei-Jing Zhu. Bleu: A method for automatic evaluation of machine translation. In *Proceedings of the 40th Annual Meeting of the Association for Computational Linguistics*, pages 311–318, 2002. 1.2

[134] Wiliiam H. Press, Saul A. Teukolsky, William T. Vetterling, and Brian P. Flannery. *Numerical Recipes in C: The Art of Scientific Computing*, chapter 20.2 Gray Codes, page 896. Cambridge University Press, 2nd edition, 1992. 3.8, 3.3.4

[135] Florian Pudlitz, Florian Brokhausen, and Andreas Vogelsang. Extraction of system states

from natural language requirements. In *International Requirements Engineering Conference*, RE, 2019. 2.2.3

[136] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. Exploring the limits of transfer learning with a unified text-to-text transformer. *Journal of Machine Learning Research*, 21: 1–67, 2020. 2.1.3, 4.1.2

[137] Ganesan Ramalingam, John Field, and Frank Tip. Aggregate structure identification and its application to program analysis. In *Symposium on Principles of Programming Languages*, POPL, 1999. 2.1.2

[138] Veselin Raychev, Martin Vechev, and Andreas Krause. Predicting program properties from "Big Code". In *Symposium on Principles of Programming Languages*, POPL, pages 111–124, 2015. 2.2.1, 2.2.3, 3

[139] Shuo Ren, Daya Gou, Shuai Lu, Long Zhou, Shujie Liu, Duyu Tang, Neel Sundaresan, Ming Zhou, Ambrosio Blanco, and Shuai Ma. CodeBLEU: A method for automatic evaluation of code synthesis. *arXiv:2009.10297*, 2020. 1.2

[140] Christian Rossow, Dennis Andriesse, Tillman Werner, Brett Stone-Gross, Daniel Plohmann, Christian J. Dietrich, and Herbert Bos. SoK: P2PWNED — Modeling and evaluating the resilience of peer-to-peer botnets. In *Symposium on Security and Privacy*, SOSP, pages 97–111, 2013. 1, 2.1.1

[141] Devjeet Roy, Ziyi Zhang, Maggie Ma, Venera Arnaoudova, Annibale Panichella, Sebastiano Panichella, Danielle Gonzalez, and Mehdi Mirakhorli. DeepTC-Enhancer: Improving the readability of automatically generated tests. In *International Conference on Automated Software Engineering*, ASE, 2020. 2.2.3

[142] David E Rumelhart, Geoffrey E Hinton, Ronald J Williams, et al. Learning representations by back-propagating errors. *Cognitive Modeling*, 5(3):1, 1988. 2.1.3

[143] William A. Sassaman. A computer program to translate machine language into FORTRAN. In *Spring Joint Computer Conference*, AFIPS, 1966. 2.1.2

[144] Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE Transactions on Neural Networks*, 20(1): 61–80, 2009. 2.1.3

[145] Eric Schulte, Jason Ruchti, Matt Noonan, David Ciarletta, and Alexey Loginov. Evolving exact decompilation. In *Workshop on Binary Analysis Research*, BAR, 2018. 1, 4

[146] Edward J. Schwartz, JongHyup Lee, Maverick Woo, and David Brumley. Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring. In *USENIX Security Symposium*, USENIXSEC, pages 353–368, 2013. 1, 2.1.1

[147] Edward J. Schwartz, Cory F. Cohen, Michael Duggan, Jeffrey Gennari, Jeffrey S. Havrilla, and Charles Hines. Using logic programming to recover C++ classes and methods from compiled executables. In *Conference on Computer and Communications Security*, CCS, 2018. 4.3

[148] Rico Sennrich, Barry Haddow, and Alexandra Birch. Neural machine translation of rare words with subword units. *arXiv preprint arXiv:1508.07909*, 2015. 4.2.1, 4.3

[149] Lin Shi, Mingzhe Xing, Mingyang Li, Yawen Wang, Shoubin Li, and Qing Wang. Detection of hidden feature requests from massive chat messages via deep Siamese network. In *International Conference on Software Engineering*, ICSE, 2020. 2.2.3

[150] Richard Shin, Miltiadis Allamanis, Marc Brockschmidt, and Alex Polozov. Program synthesis and semantic parsing with learned code idioms. In *Conference on Neural Information Processing Systems*, NeurIPS, 2019. 2.2.3

[151] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Andrew Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. (State of) The art of war: Offensive techniques in binary analysis. In *Symposium on Security and Privacy*, SP, pages 138–157, 2016. 4.2.3

[152] Asia Slowinska, Traian Stancescu, and Herbert Bos. Howard: A dynamic excavator for reverse engineering data structures. In *Network and Distributed System Security Symposium*, NDSS, 2011. 2.1.2, 4.2.3

[153] Harry M. Sneed. Object-oriented COBOL recycling. In *Working Conference on Reverse Engineering*, WCRE, 1996. 5

[154] Venkatesh Srinivasan and Thomas Reps. Recovery of class hierarchies and composition relationships from machine code. In *Compiler Construction*, 2014. 2.1.2

[155] Brett Stone-Gross, Marco Cova, Lorenzo Cavallaro, Bob Gilbert, Martin Szydlowski, Richard Kemmerer, Christopher Kruegel, and Giovanni Vigna. Your botnet is my botnet: Analysis of a botnet takeover. In *ACM Conference on Computer and Communications Security*, CCS, November 2009. 1, 2.1.1

[156] Alexey Svyatkovskiy, Shao Kun Deng, Shengyu Fu, and Neel Sundaresan. Intelli-code compose: Code generation using transformer. In *Symposium on the Foundations of Software Engineering*, FSE, 2020. 2.2.3

[157] Hannes Thaller, Lukas Linsbauer, and Alexander Egyed. Feature maps: A comprehensible software representation for design pattern detection. In *International Conference on Software Analysis, Evolution and Reegnineering*, SANER, 2019. 2.2.3

[158] Katerina Troshina, Yegor Derevenets, and Alexander Chernov. Reconstruction of composite types for decompilation. In *Working Conference on Source Code Analysis and Manipulation*, SCAM, pages 179–188, 2010. 4

[159] Michael James Van Emmerik. *Static Single Assignment for Decompilation*. PhD thesis, University of Queensland, 2007. 1, 1, 2.1.1

[160] Bogdan Vasilescu, Casey Casalnuovo, and Premkumar Devanbu. Recovering clear, natural identifiers from obfuscated JavaScript names. In *Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering*, ESEC/FSE, pages 683–693, 2017. 2.2.1, 2.2.3, 3

[161] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. Attention is all you need. In *Conference*

on Neural Information Processing Systems, NeurIPS, pages 6000–6010, 2017. 2.1.3, 4, 4.1.2, 4.2.4, 4.3

[162] Nguyen Xuan Vinh, Julien Epps, and James Bailey. Information theoretic measures for clusterings comparison: Variants, properties, normalization and correction for chance. *The Journal of Machine Learning Research*, 11:2837–2854, 2010. 4.1.5

[163] Daniel Votipka, Seth M. Rabin, Kristopher Micinski, Jeffrey S. Foster, and Michelle M. Mazurek. An observational investigation of reverse engineers' processes. In *USENIX Security Symposium*, USENIX, 2020. 1, 1.2, 5.1

[164] Xiaohui Wan, Zheng Zheng, Fangyun Qui, Yu Qiao, and Kishor S. Trivedi. Supervised representation learning approach for cross-project aging-related bug prediction. In *International Symposium on Software Reliability Engineering*, ISSRE, 2019. 2.2.3

[165] Yao Wan, Zhou Zhao, Min Yang, Guandong Xu, Haochao Ying, Jian Qu, and Philip S. Yu. Improving automatic source code summarization via deep reinforcement learning. In *International Conference on Automated Software Engineering*, ASE, 2018. 2.2.3

[166] Weiyue Wang, Jan-Thorsten Peter, Hendrik Rosendahl, and Hermann Ney. CharacTer: Translation edit rate on character level. In *WMT '16*, pages 505–510, 2016. 3.3

[167] Wentao Wang, Nan Nui, Hui Liu, and Zhendong Niu. Enhancing automated requirements traceability by resolving polysemy. In *International Requirements Engineering Conference*, RE, 2018. 2.2.3

[168] Yauhui Wang, Hui Xu, Yangfan Zhou, Michael R. Lyu, and Xin Wang. Textout: Detecting text-layout bugs in mobile apps via visualization-oriented learning. In *International Symposium on Software Reliability Engineering*, ISSRE, 2019. 2.2.3

[169] Cody Watson, Michele Tufano, Kevin Moran, Gabriele Bavota, and Denys Poshyvanyk. On learning meaningful assert statements for unit test cases. In *International Conference on Software Engineering*, ICSE, 2020. 2.2.3

[170] Bolin Wei, Ge Li, Xin Xia, Zhiyi Fu, and Zhi Jin. Code generation as a dual task of code summarization. In *Conference on Neural Information Processing Systems*, NeurIPS, 2019. 2.2.3

[171] Bolin Wei, Yongmin Li, Ge Li, Xin Xia, and Zhi Jin. Retrieve and refine: Exemplar-based neural comment generation. In *International Conference on Automated Software Engineering*, ASE, 2020. 2.2.3

[172] Jiayi Wei, Maruth Goyal, Greg Durrett, and Isil Dillig. LambdaNet: Probabilistic type inference using graph neural networks. In *International Conference on Learning Representations*, ICLR, 2020. 2.2.2

[173] Jonas Paul Winkler, Jammis Grönberg, and Andreas Vogelsang. Predicting how to test requirements: An automated approach. In *International Requirements Engineering Conference*, RE, 2019. 2.2.3

[174] Wensheng Xia, Ying Li, Tong Jia, and Zhonghai Wu. Bugidentifier: An approach to identifying bugs via log mining for accelerating bug reporting stage. In *International Conference on Software Quality, Reliability and Security*, QRS, 2019. 2.2.3

[175] Yan Xiao, Jacky Keung, Kwabena E. Bennin, and Qing Mi. Improving bug localization with word embedding and enhanced convolutional neural networks. *Information and Software Technology*, 2019. 2.2.3

[176] Kelvin Xu, Jimmy Ba, Ryan Kiros, Kyunghyun Cho, Aaron Courville, Ruslan Salakhudinov, Rich Zemel, and Yoshua Bengio. Show, attend and tell: Neural image caption generation with visual attention. In *International Conference on Machine Learning*, ICML, pages 2048–2057, 2015. 2.1.3, 4.1.1

[177] Khaled Yakdan, Sebastian Eschweiler, Elmar Gerhards-Padilla, and Matthew Smith. No more gotos: Decompilation using pattern-independent control-flow structuring and semantics-preserving transformations. In *Network and Distributed System Security Symposium*, NDSS, 2015. 1, 2.1.1

[178] Khaled Yakdan, Sergej Dechand, Elmar Gerhards-Padilla, and Matthew Smith. Helping Johnny to analyze malware: A usability-optimized decompiler and malware analysis user study. In *Symposium on Security and Privacy*, SP, pages 158–177, 2016. 1, 1.2, 2.1.1, 5.2

[179] Fangke Ye, Jisheng Zhao, and Vivek Sarkar. Advanced graph-based deep learning for probabilistic type inference. *arXiv preprint arXiv:2009.05949*, 2020. 2.2.2

[180] Manzil Zaheer, Guru Guruganesh, Avinava Dubey, Joshua Ainslie, Chris Alberti, Santiago Ontanon, Philip Pham, Anirudh Ravula, Qifan Wang, Li Yang, et al. Big Bird: Transformers for longer sequences. *arXiv preprint arXiv:2007.14062*, 2020. 2.1.3, 4.1.2, 4.3

[181] Jian Zhang, Xu Wang, Hongyu Zhang, Hailong Sun, Yanjun Pu, and Xudong Liu. Learning to handle exceptions. In *International Conference on Automated Software Engineering*, ASE, 2020. 2.2.3

[182] Jinglei Zhang, Riu Xie, Wei Ye, Yuhan Zhang, and Shikun Zhang. Exploiting code knowledge graph for bug localization via bi-directional attention. In *International Conference on Program Comprehension*, ICPC, 2020. 2.2.3

[183] Mingwei Zhang, Aravind Prakash, Xiaolei Li, Zhenkai Liang, and Heng Yin. Identifying and analysing pointer misuses for sophistocated memory-corruption exploit diagnosis. In *Network and Distributed System Security Symposium*, NDSS, 2012. 2.1.2

[184] Yating Zhang, Wei Dong, Daiyan Wang, Binbin Liu, and Jiaxin Liu. Accuracy improvement for neural program synthesis via attention mechanism and program slicing. In *Computers, Software, and Applications Conference*, COMPSAC, 2020. 2.2.3

[185] Z. Zhang, Y. Ye, W. You, G. Tao, W. Lee, Y. Kwon, Y. Aafer, and X. Zhang. OSPREY: Recovery of variable and data structure via probabilistic analysis for stripped binary. In *Symposium on Security and Privacy*, SP, pages 872–891, 2021. 2.2.2, 4.2.3, 4.2.3

[186] Zhuo Zhang, Yan Lei, Xiaoguang Mao, and Panpan Li. CNN-FL: An effective approach for localizing faults using convolutional neural networks. In *International Conference on Software Analysis, Evolution and Reegnineering*, SANER, 2019. 2.2.3

[187] Zhuo Zhang, Wei You, Guanhong Tao, Guannan Wei, Yonghwi Kwon, and Xiangyu Zhang. BDA: Practical dependence analysis for binary executables by unbiased whole-

program path sampling and per-path abstract interpretation. *Proceedings of the ACM on Programming Languages*, 3(OOPSLA):1–31, 2019. 4.2.3

[188] Dehai Zhao, Zhenchang Xing, Chunyang Chen, Xin Xia, and Guoqiang Li. Action-Net: Vision-based workflow action recognition from programming screencasts. In *International Conference on Software Engineering*, ICSE, 2019. 2.2.3

[189] Hui Zhao, Zhihui Li, Hansheng Wei, Jianqi Shi, and Yanhong Huang. SeqFuzzer: An industrial protocol fuzzing framework from a deep learning perspective. In *International Conference on Software Testing, Verification, and Validation*, ICST, 2019. 2.2.3

[190] Yutong Zhao, Lu Xiao, Pouria Babvey, Lei Sun, Sunny Wong, Angel A. Martinez, and Xiao Wang. Automatically identifying performance issue reports with heuristic linguistic patterns. In *Joint Meeting of the European Software Engineering Conference and the Symposium on the Foundations of Software Engineering*, ESEC/FSE, 2020. 2.2.3

[191] Cheng Zhou, Bin Li, and Xiaobing Sun. New deep learning method to detect code injection attacks on hybrid applications. *The Journal of Systems and Software*, 2018. 2.2.3

[192] Cheng Zhou, Bin Li, and Xiaobing Sun. Improving software bug-specific named entity recognition with deep neural network. *The Journal of Systems and Software*, 2020. 2.2.3

[193] Shuyan Zhou, Uri Alon, Frank F. Xu, Zhengbao Jiang, and Graham Neubig. DocPrompting: Generating code by retrieving the docs. In *International Conference on Learning Representations*, ICLR, 2023. 2.2.3

[194] Yu Zhou, Xin Yan, Wenhua Yang, Taolue Chen, and Zhiqiu Huang. Augmenting Java method comments generation with context information based on neural networks. *The Journal of Systems and Software*, 2019. 2.2.3