A Taxonomy of Program Comprehension Challenges in Decompiled Code

Anonymous Authors
Anonymous Institutuion

Abstract

Decompilation is an important part of threat analysis for cybersecurity. Unfortunately, decompiled code is missing information relative to the corresponding original source code, which makes this process more difficult for reverse engineers that manually perform threat analysis. Thus, the interpretability of decompiled code matters, as it can influence reverse engineers' productivity. There is some existing work in predicting some of the missing information using statistical methods, but these focus largely on variable names and variable types. In this work, we more holistically evaluate decompiler output and use our findings to inform directions for future decompiler development. More specifically, we use open-coding techniques to identify defects in decompiled code beyond missing names and types. To ensure that our study is robust, we compare and evaluate several different decompilers. We analyze our results and build a taxonomy of decompiler defects. Using this taxonomy to reason about classes of issues, we suggest specific approaches that can be used to mitigate interpretability issues in decompiled code.

1 Introduction

Decompilation—the process of analyzing a compiled program and recovering a source-code program that portrays the same behavior—is a crucial tool in computer security, as it allows security practitioners to more quickly gain a deep understanding of the behavior of compiled programs. This is particularly useful in security scenarios such as analyzing malware and commercial-off-the-shell software (COTS), where the source code may be unavailable. By converting executables into human-readable C-like code, decompilation allows security practitioners to more effectively understand and respond to the threats posed by malware [29]. An example of this can be seen in the study by Ďurfina et al. [35], where analysts employed a decompiler to analyze the `Psyb0t` worm, a piece of malware that infects routers to build a botnet.

Analyzing and understanding the behavior of executable code is significantly more difficult than analyzing source code due to information that is removed by the compilation process. Indeed, while high-level programming languages contain abstractions and constructs such as variable names, types, comments, and control-flow structures that make it easier for humans to write and understand code [28], executable programs do not. These abstractions are not necessary for an executable program to run, and thus they are discarded, simplified, or optimized away by compilers in the interest of minimizing executable size and maximizing execution speed. This means that those useful abstractions are not present when it comes time to analyze an executable program, such as malware, without access to its source code.

Original

```
1  void cbor_encoder_init(CborEncoder *encoder, uint8_t
       *buffer, size_t size, int flags)
2  {
3    encoder->ptr = buffer;
4    encoder->end = buffer + size;
5    encoder->added = 0;
6    encoder->flags = flags;
7  }
```

Decompiled

```
1  long long cbor_encoder_init(long long a1, long long
       a2, long long a3, int a4)
2  {
3    long long result;
4    *((_QWORD *) a1) = a2;
5    *((_QWORD *) (a1 + 8)) = a3 + a2;
6    *((_QWORD *) (a1 + 16)) = 0LL;
7    result = a1;
8    *((_DWORD *) (a1 + 24)) = a4;
9    return result;
10 }
```

Figure 1: A decompiled function and its source definition. Decompilers can't recover many of the abstractions that make source code readily readable by human developers. Furthermore, they may incorrectly recover semantics, as demonstrated by the decompiled function's extra return statement.

Traditionally, security practitioners would reverse engineer executables by using a disassembler to represent the semantics of the program as assembly code. While better than nothing, assembly code is still far from readable. Decompilers fill this gap by analyzing an executable's behavior and attempting to recover a plausible source code representation of the behavior. Despite a great deal of work, decompilation is a notoriously difficult problem and even state-of-the-art decompilers emit source code that is a mere shell of its former self [14, 20, 27, 31, 32]. Despite this, decompilers are one of the most popular tools used by reverse engineers.

Figure 1 shows what a decompiled function looks like compared to the original function definition. Although the decompiled code is C source code,[1] it is arguably quite different from the original, and quite awkward to read. We say that decompiled C code is not *idiomatic*; that is, though it is grammatically legal C code, it does not use common conventions for ensuring that source code is readable. Further, as Figure 1 also illustrates, decompiler output may be incorrect; that is, it may be semantically nonequivalent to the code in its executable form. We collectively call these issues—readability and correctness issues—*interpretability* issues. (See Section 3.1 for a more thorough discussion of interpretability).

Interpretability issues are problematic because decompiled code is usually created to be *manually* read by reverse engineers. Reverse engineering is a painstaking process which involves much time spent rebuilding these abstractions as the reverse engineer develops an understanding of what the executable binary does [29]. Thus, the interpretability of decompiled code matters, as it can significantly impact reverse engineers' productivity.

Improving the functionality and usability of decompilers has long been an active research area, with many contemporary efforts [6, 11, 12, 23, 30]. A recent trend in this direction is using deep learning-based techniques to improve the process of decompilation [10, 13, 15, 16, 25], or directly the output of decompilers [1, 4, 9, 19, 24], inspired by advances in natural language processing. The latter strands of work have the potential benefit of building on top of mature tools like Hex-Rays and Ghidra instead of operating on binaries, and have already seen promising results for recovering missing variable names and types. Here, researchers have been developing models that *learn* to suggest meaningful information in a given context with high accuracy, after seeing many examples of original source code drawn from open-source repositories like the ones hosted on GitHub.

However, while variable names and types are certainly important for program comprehension, including in a reverse engineering context [6, 29, 33], there are many more characteristics of decompiled code that make it difficult to interpret, and relatively little knowledge of what they are, how they vary across decompilers, how they impact interpretability, and what are the implications for learning-based approaches aiming to improve the interpretability of decompiled code.

We argue that before designing more advanced solutions, we first need a deeper understanding of the problem. Consequently, in this paper we set out with the **Research Goal** of developing a comprehensive taxonomy of the characteristics that make decompiled code difficult to interpret. Concretely, we start by curating a sample of open-source functions decompiled with the Hex-Rays,[2] Ghidra,[3] and retdec[4] decompilers. Next, we use thematic analysis, a qualitative research method for systematically identifying, organizing, and offering insights into patters of meaning (themes) across a dataset [5], to analyze the decompiled functions for interpretability defects, using those functions' original source code as an oracle. To minimize subjectivity, we develop a novel abstraction for determining correspondence between code pairs, which we refer to as *alignment*. Using this abstraction, we precisely and unambiguously define interpretability defects in decompiled code, creating a taxonomy consisting of 14 top-level issue categories with 48 in total. In turn, we use our taxonomy to suggest how the issues could be addressed, framing our discussion around the role that deterministic static analysis and learning-based approaches could play. Our results are robust both across different researchers as well as the three decompilers we considered.

In summary, we make the following contributions:

- A comprehensive, hierarchical taxonomy covering interpretability issues in decompiled code beyond names and types.
- 160 coded decompiled/original function pairs, identifying over one thousand instances of issues in our taxonomy.
- A novel abstraction for assigning code correspondence in source code pairs and a framework built on this abstraction for rigorously applying open coding to those source code pairs.
- A comparison of three different modern decompilers.
- A thorough analysis describing classes of decompiler issues and suggestions on what techniques could be used to fix them.

2 Related Work

2.1 Decompilation

Significant efforts have been made in recent years to improve the performance of decompilers. A large portion of these

[1] Decompiled code is not always syntactically correct C code.

[2] https://hex-rays.com/decompiler/

[3] https://ghidra-sre.org/

[4] https://github.com/avast/retdec

efforts concentrate on addressing the core challenges in program analysis that are fundamental to decompilation, which mainly include type recovery and control flow structuring.

Type recovery is the process of identifying variables and assigning them reasonable types by analyzing the behavior of the executable. We refer readers to an excellent survey of type recovery systems [7], but also review some notable security-oriented work developed in recent years. This includes systems that operate on dynamic runtime traces, such as REWARDS [21], and follow-on systems that statically recover the types by analyzing the executable code at rest. TIE [20] is an examplar static recovery system whose design has been used in the academic Phoenix [27] and DREAM [31, 32] decompilers. The Hex-Rays decompiler uses its own static type recovery system [14].

Control flow structuring is the process of converting a control flow graph (CFG), which is unstructured, into the structured control flow commands that are more common in source languages, such as if-then-else and while loops. The Phoenix [27] decompiler introduced a control flow structuring algorithm that was designed explicitly for decompilation in that it was semantics preserving, unlike other more general structuring algorithms. One of the main challenges of control flow structuring is how to handle code that cannot be completely structured, which can be caused by using non-structured language constructs such as gotos. Although the Phoenix algorithm preserved semantics, it emitted gotos for unstructured code, which could make the decompiled code hard to read. The DREAM [31] and DREAM++ [32] decompilers followed up on this work by introducing a new control flow structuring algorithm and other changes that were intended to improve the usability of the decompiler. Notably, their structuring algorithm duplicated some code to avoid emitting gotos, which they found to improve readability.

Some researchers [8, 13, 16, 17] have focused on using machine learning to model the entire decompilation process. This approach, known as *neural decompilation*, attempts to map a low-level program representation, such as assembly, directly to the source code of a high-level language like C using a machine learning model, sometimes in multiple phases. To some extent, these approaches sidestep the need to identify specific interpretability defects in decompiled code because they are theoretically capable of generating decompiler output that is completely identical to the original source code for that program. However, neural decompilation often fails to match the original source code.

2.2 Improving decompilation through learning

A recent area of work has studied whether it is possible to correct some of the limitations of current decompilers through learning-based methods. A popular focus of this work is recovering variable names for decompiled code, which is a problem that is not well-suited to traditional program analysis since the variable names are not explicitly stored in the executable. Lacomis et al. [19] and Nitin et al. [24] found that the generic variable names used in most decompilers (v1, v2, etc.), make it more difficult to read decompiled code than the original. In response, they propose machine-learning-based tools that propose meaningful variable names in decompiled code to help alleviate this issue. Chen et al. [9] also found that variable *types* are often recovered incorrectly by decompilers, especially composite types like C-language struct, array, and union types. They build a machine-learning-based tool to predict missing variable names and types at the same time, and note that variable names inform variable types and vice versa. However, while variable names and types are important interpretability defects, they represent only a subset of all readability defects in decompiled code. Our study develops a more complete taxonomy of readability defects in decompiled code.

2.3 Taxonomy of decompilation defects

Liu and Wang [22] do provide a taxonomy of some defects in decompiled code, but their study is orthogonal to ours. They focus only on those defects that produce semantic differences in the source code, while we more broadly investigate the defects in decompiled code that cause reverse engineers difficulty in determining the functionality of the decompiled code, including semantic differences. Further, their taxonomy differentiates these semantic defects by the phases of decompilation in which they originate rather than by the nature of the defects themselves. In short, their study focuses on the decompiler itself while ours focuses on the interpretability of the decompiler output.

3 Methodology

We used open-coding techniques [18] to identify the defects that make decompiled code difficult to analyze. In particular, we used pairs of function representations: a decompiled function and the corresponding original function. Open coding is typically used on textual data, such as interview data, to provide a systematic strategy for analyzing patterns. Coding pairs of source code functions like this offered some unique challenges. First, we detail the philosophy we developed to help overcome those challenges. Next, we discuss practices we used to help guide the coding process. Finally, we discuss the process we used to code our examples and develop the codebook.

In open-coding techniques, features of the analyzed entity are assigned labels called "codes." Unfortunately, "code" is also a word used to describe text written in a programming language. In this paper, for clarity, we will use "label" to refer to open-coding codes and "code" or "source code" to refer to text written in a programming language. "Original code"

and "decompiled code" are types of source code, the former (most likely) written by a human programmer and the latter generated by a decompiler from an executable program. We continue to use the term "codebook" to indicate the collection of all open-coding labels rather than the term "labelbook."

3.1 Interpretability

Our study is intended to identify, characterize, and catalog the characteristics of decompiled code that make it difficult to analyze; that is, to determine the features of decompiled code that inhibit the analyst from understanding what a given piece of code does. "What a given piece of code does" is, most precisely, its runtime behavior, but running arbitrary executables is cumbersome and observing runtime behavior is nearly impossible. (We can only easily observe a subset of behavior: output). As a result, our proxy for runtime behavior—our oracle—is the original source code. Derived from this principle, the labels in our codebook are expressed in terms of *differences relative to the original source code* rather than in terms of concrete features of the decompiled code. Henceforth, we use the term "difference" to refer to the difference between a piece of decompiled code and the corresponding piece or original code.

There are several ways that a source code representation (e.g. decompiled code) of some abstract true functionality (e.g. runtime behavior) could inhibit an analyst from understanding that true functionality:

- if the code represents functionality that is different from the true functionality. An analyst might be mislead into thinking that the functionality is something other than what it actually is. In our case, this occurs when there is a semantic difference between the decompiled code and our oracle, the corresponding original code. We refer to this as a *correctness issue*.
- if the code is semantically equivalent but is communicated using language that is difficult to interpret. We refer to these as *readability issues*.

Our oracle is most suitable for diagnosing correctness issues. A human researcher can look at the decompiled code and, with sufficient effort, determine if it is semantically equivalent to the corresponding original code. It is less suitable for identifying readability issues. After all, there is no guarantee that the original source code is readable. Furthermore, readability itself is somewhat subjective. What some might consider readable others might not.

To shore up this source of subjectivity, we introduce a second test. To determine if a difference affects readability, we ask if the difference simply reflects a difference between two common idiomatic styles. For example, the decompiler may place opening curly-brackets on a new line after an if-statement conditional, while the original code might place them on the same line as the conditional. Both styles are common and idiomatic, so this does not constitute a readability issue. Each unique readability issue, along with each correctness issue, is assigned a label.

Unfortunately, our solution to the readability oracle problem does not eliminate subjectivity. Rather, it shifts the subjectivity to a different place—what styles are idiomatic? We estimate idiomaticity by asking if a given style is common in C-language source code. For example, some original code functions include extraneous code like a `do { ... } while(0);` loop. In each instance we observed this, the decompiled code does not include the extraneous code. We consider this difference to be benign. Another example is inverted conditional statements. The original code might have an if statement of the form `if (!a) b else c`, while the decompiled code might represent that if statement as `if (a) c else b`. One ordering of the clauses is not necessarily more idiomatic than the other. Missing `volatile` or `static` keywords in the decompiled code are also benign differences because they do not affect the computations the function performs. We provide a list of differences that we do not consider to be non-idiomatic, and thus are not classified as readability issues, in the appendix. Further, as prior work has identified [9, 19, 24], variable names and types are often different in decompiled code. While these do constitute readability issues, we do not include them in our taxonomy so as to focus on interpretability issues not identified by prior research.

We treat borderline cases, as well as differences which only sometimes result in non-idiomatic code, as readability issues. A summary of the decision process we used to determine if a code should be added to the codebook is shown in Figure 2.

3.2 Coding Standards

Performing open-coding on our data presented some interesting challenges. We detail them and our solutions to them here.

3.2.1 Alignment

A critical assumption made in Section 3.1 was that differences between decompiled and original code could be easily identified. However, it is not immediately evident what precisely constitutes a "difference." Our goal is to determine if and how clearly the functionality present in the original code is communicated. Thus, we want to determine which pieces of the the code are supposed to represent the same functionality. We say two code fragments that perform the same functionality (or are intended to) are *aligned*. If the text of aligned code fragments is not identical, this constitutes a difference.

We have found that it is often easy to align code by hand, but it is challenging to define what it means for code to be aligned. Figure 3 demonstrates several factors that make align-

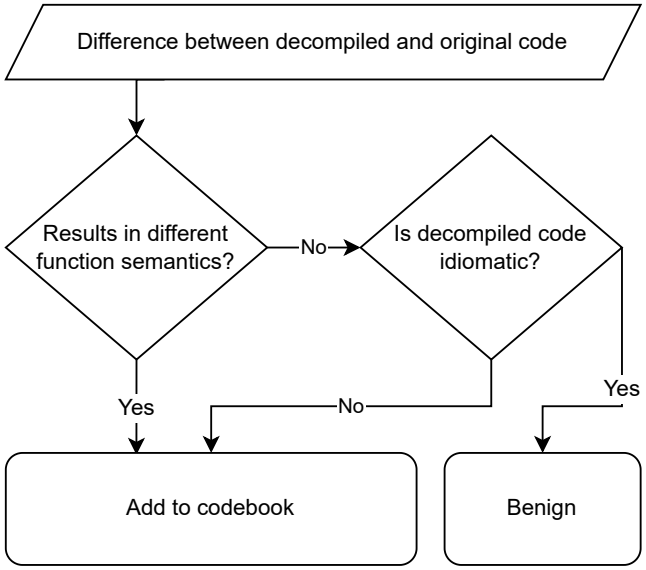

Figure 2: The process by which a difference between the decompiled code and corresponding original code was added to the codebook. Differences added to the codebook are considered meaningful defects, while others are considered benign differences in style. When in doubt, differences were added to the codebook.

ment difficult. Decompilers may introduce extra statements, such as declarations and operations for variables that have no equivalent in the original source code (e.g., decompiled line 3 of Figure 3). Decompilers may also break complex expressions down into multiple statements, as happens when line 9 in the original code is broken into two statements in the decompiled code (lines 7 and 8). The opposite can be true as well—decompilers can inline expressions that are separate in the original code, and can even exclude code from the original function like redundant or extraneous assert statements. Thus, line numbers and other simple heuristics are generally unsuitable for aligning statements. In fact, many-to-one and one-to-many mappings between statements, like Figure 3's original line 9 and decompiled lines 7 and 8, means that alignment can't even be thought of as a mapping between individual statements.

To alleviate these challenges and to precisely differentiate labels, we introduce a formal definition of alignment. We model functions as seven-element tuples $(F, A, P, C, \eta, B, \xi)$:

- F: a set of operators. An operator is the finest-grained source-level unit of functionality from the perspective of an individual function. The idea of an operator is analogous to that of a machine instruction, but at the source level. Operators can be function calls like `set_error` in Figure 3 or `==`. The former is an operator that accepts four arguments, the latter accepts two. We assume without loss of generality that operators in functions can be uniquely identified.
- A: a set of operator argument variables. A contains all arguments for all operators in the function. For example, in Figure 3, $f_{\text{set\_error}} = f_3$ has for argument variables $a_{3,1}, a_{3,2}, a_{3,3},$ and $a_{3,4}$.
- P: a set of parameters $p_1, ..., p_n$ to the function.
- C: a set of constants $c_1, ..., c_m$ used in the function.
- $\eta : A \to P \cup C \cup F$: a dataflow mapping determining from which values are passed as arguments to each function. For example, in Figure 3, if $f_{\text{obj\_dump\_template}} = f_6$, then $a_{6,1} \to p_1$, $a_{6,2} \to p_2$, $a_{6,3} \to f_{\text{set\_error}}$, and $a_{6,4} \to 0$. Taking inspiration from Single Static Assignment (SSA) [2, 26], we use the notation $\phi()$ to indicate that a given argument variable might be assigned different values based on control flow e.g. $a_{i,j} \to \phi(c_0, f_7)$.
- B: A set of maximal basic blocks. Each basic basic block is a partially ordered subset of 2^F and consists of operators that execute without a change in control flow. The partial ordering reflects operators with side effects which must be executed in a specific order.
- ξ: B x B: A set of control-flow edges that connect basic blocks.

Finally, an alignment is a mapping between operators in different functions. An alignment mapping can be one-to-many or many-to-one; that is there are cases when multiple operators in the decompiled code represent the same functionality as one in the decompiled code and vice versa. A perfect alignment is one in which the aligned operators perform the same functionality and for which their control and data dependencies match. If there are correctness issues in a piece of decompiled code, then an alignment cannot be perfect. In these cases, we must settle for a good approximate match.

In most cases, alignment is eminently evident when coding an example, and it is unnecessary to consult the formal definition. However, the formal definition is useful for corner cases and for further precisely defining certain labels.

3.2.2 Multi-coding

Certain decompiler defects exhibit the characteristics of several different labels at once. In these cases, we allow for multiple labels to identify the same issue. We carefully select labels so that each fundamental issue receives its own label.

3.3 Dataset

We drew examples for our study from a large dataset derived from open-source projects on GitHub. This dataset was generated in an automated fashion by scraping GitHub through its

Original

```
void help_object(obj_template_t *obj, help_mode_t mode)
{
  if (obj == 0)
  {
    set_error("help.c", 436, ERR_INTERNAL_PTR, 0);
    return;
  }

  obj_dump_template(obj, mode, get_nestlevel(mode), 0);
}
```

Decompiled

```
long long help_object(long long a1, unsigned int a2)
{
  unsigned int v3;
  if (!a1)
    return set_error("help.c", 436LL, 2LL, 0LL);

  v3 = get_nestlevel(a2);
  return obj_dump_template(a1, a2, v3, 0LL);
}
```

Figure 3: A decompiled function along with the corresponding original definition. Aligning decompiled code with the original is often easy to perform by hand, but even a relatively simple function like this one illustrates some of the challenges in defining alignment precisely. Except for return behavior, the decompiled function exhibits the same functionality as the original. However, the decompiled code uses a different, though equivalent, test in the `if` conditional, contains extra variables, reorganizes expressions, and uses a constant instead of the original code's macro.

API to collect majority-C language repositories. In total, our dataset contains functions from 81,137 repositories. A build of each project was attempted by looking for build scripts such as `Makefile`s and executing those to build executable binaries. All binaries generated by the build process were collected. Next, each binary was decompiled using the Hex-Rays decompiler, and all functions in the binaries were collected for a total of 8,857,873 across all projects. These functions were matched with the corresponding original function definitions in the original code. These functions were divided by size. Functions with more than 512 sub-words (which together make up an identifier name) and AST nodes were sorted into in the large dataset (31% of the total); those with more were sorted into the larger dataset (69% of the total). We mostly used functions from the small dataset, though we also coded ten functions from the large dataset.

3.4 Coding Procedure

We selected a random sample of 200 decompiled/original function pairs. We performed coding in several phases. All steps, except step five, were performed by the first author:

1. **Collecting Differences.** We examined the first 100 examples from the small dataset, building a set of differences without making judgement as to which semantics-preserving differences constituted readability differences. This also allowed us to get a sense of common, idiomatic practices from amongst the original code samples.

2. **Building the Codebook.** Next, we assembled a codebook from the differences. To do this, we first labeled each semantics-preserving differences as either a readability issue or a benign difference. All readability issues and non-semantics-preserving differences (correctness issues) were assigned a label. Labels were organized hierarchically; higher-level labels generalizing several related labels were created when necessary.

3. **Coding Examples** With a complete codebook, we then labeled the next 100 examples from the small dataset as well as 10 examples from the large dataset. This was an iterative process which ultimately involved refining the codebook and deriving precise definitions for each label. There was some interplay between this phase and the next.

4. **Generalization Across Decompilers** The process so far had been dependent on code that was decompiled by the popular Hex-Rays decompiler. However, we wanted to ensure that our results were representative of issues faced by decompilers in general and not specific to a certain tool. Thus, we randomly sampled 25 of the second 100 examples, decompiled the corresponding binaries using two other decompilers, Ghidra and retdec, and extracted the requisite functions. We then labeled those examples using the same codebook.

5. **Generalization Across Coders** To ensure that our codebook was robust, we performed several rounds of intercoder reliability testing [3]. We gave the completed codebook, along with thorough documentation and examples, to a researcher with reverse engineering experience. We randomly sampled 25 out of the second 100 examples to gave these to him to code. We then computed interrater agreement using kohen's kappa between his labels and the corresponding labels of the first author. Next, we measured the agreement and analyzed the results for any source of disagreement. We then updated the codebook and performed the process again with a different researcher. Our evaluation of intercoder agreement is shown in Table 1.

4 Results

Our study yielded a comprehensive codebook of defects in decompiled code. Our codebook is organized hierarchically,

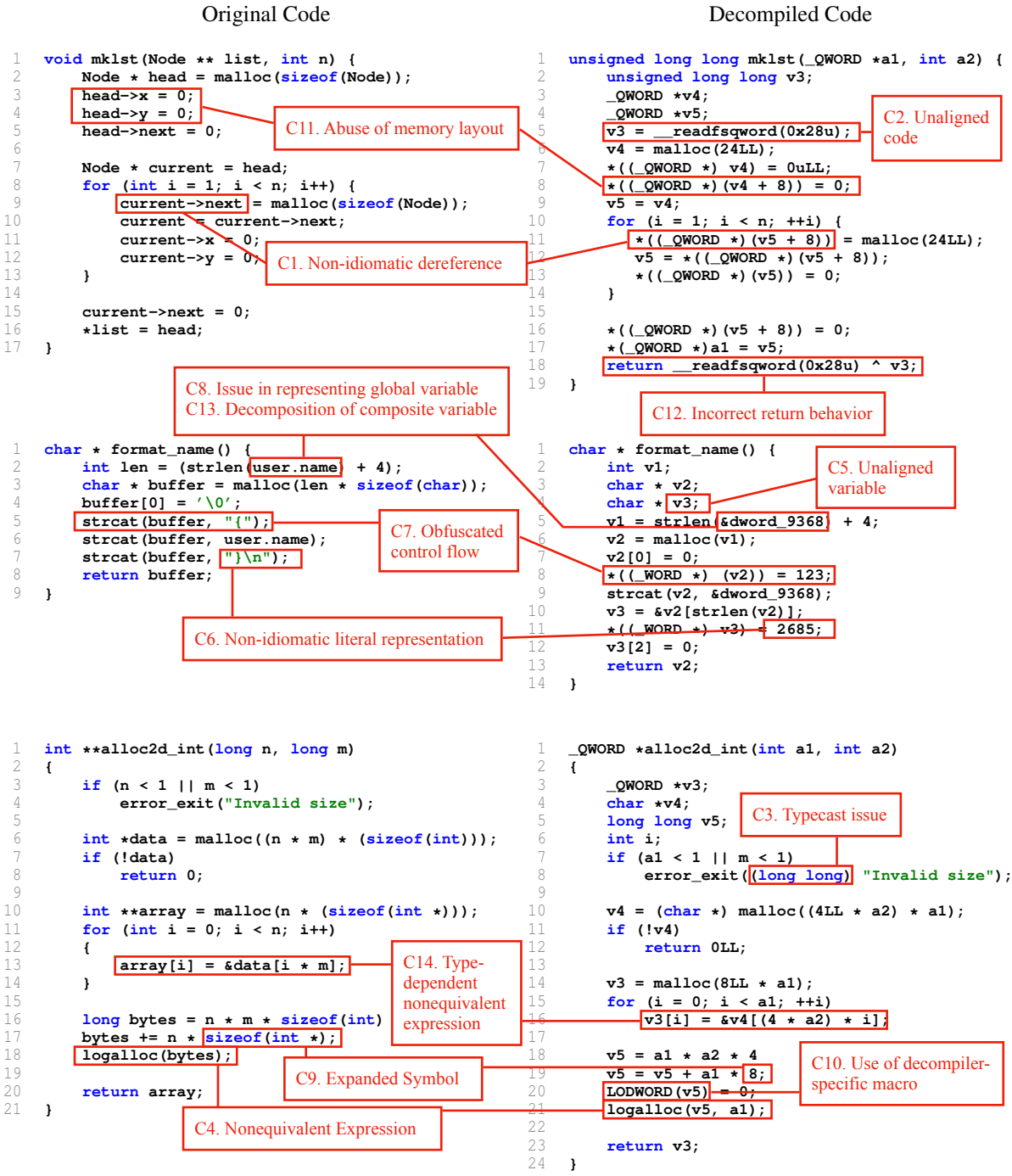

Figure 4: Examples of each of the top-level labels in our codebook. These examples have been artificially constructed by combining code fragments and common patterns from our sample in such a way that all fourteen top-level labels are represented at least once. For legibility, only one instance of each top level label is identified in the diagram, though there are multiple instances of certain issues present across each of the functions. In some cases, a subclass of the top-level label might apply; in this diagram, we identify issues only by their top-level labels for simplicity.

Measure	Round 1	Round 2
Lines researchers agreed had interpretability issues	0.67	0.78
Mean agreement across all labels	0.44	0.58
Weighted mean agreement across all labels	0.70	0.78

Table 1: Cohen's Kappa coefficient of intercoder agreement for each round of testing. After the first round of testing, we updated our codebook to make it more robust. This lead to an increase in measured agreement. Cohen's kappa is usually interpreted as follows: > 0.4 indicates moderate > 0.6 indicates good, and > 0.8 indicates very good agreement. The bottom two rows are the mean and weighted mean of cohen's kappa individually. The mean is lower because of disagreement on rare issues. However, we found that some disagreement was not true disagreement but rather the result of mismatched assumptions about information external to the example function or simple fatigue.

with broad, fundamental classes of issues at the top of the hierarchy and more specific instances at lower levels. There are 14 top-level labels, and 48 codes across all levels. In this section, we'll discuss each of the top-level labels in turn. Figure 4 contains several artificial example functions, composed of code fragments and specific issues observed in our dataset, which collectively illustrate all 14 top-level labels at least once. The complete codebook is provided in Figure 6 in the appendix.

C1. Non-idiomatic dereference: This occurs when a pointer variable in the decompiled code is used in a way that does not reflect its type in the original code. Formally, it refers to a mismatch between aligned operator(s) where the operator in the original code is used to access value at or relative to a memory location. For example, in Figure 4, C1 labels what was a struct dereference in the original code (`current->next`) decompiled as a sequence of three operations: pointer arithmetic, a typecast, and a pointer derference (`*((_QWORD *)(v5 + 8))`). The example in Figure 4 is in particular an instance of the sub-label C1.a.i. (Pointer arithmetic to access struct members) but there are others, including situations where `struct`s are accessed as if they are arrays (C1.a.ii.), and where arrays are accessed with pointer arithmetic (C1.b.ii.).

C2. Unaligned code: refers to situation where decompiled code does not align with any code in the original function; that is, the code is extra or missing relative to the original. Figure 4 illustrates an example where an extraneous variable, i.e. one that does not occur in the original source code, is itself initialized by an expression that does not occur in the original source code. In some cases, the original code will include code that is entirely irrelevant to delivering functionality. For example, some examples in our dataset include `do { ... } while(0);` loops. We do not label these with C2 because they do not contribute information about the decompiled code's functionality.

C3. Typecast issues: refers to extra or missing typecast operators relative to the original. We do not consider this to be a part of C2 because extra or missing typecasts are a consequence of decompilers' imprecise type recovery and in that sense could be considered to align with other operators to ensure that the decompiled code typechecks properly. Figure 4 illustrates an instance of an extra typecast added to a string literal, a pattern with some string literals in our dataset.

C4. Nonequivalent expression: refers to a collection of operators that align but that are not semantically equivalent to each other. Figure 4 illustrates one of several patterns found in our dataset, where a function call in the decompiled code receives an extra argument.

C5. Unaligned variable: This label refers to a situation where a variable in the decompiled code is missing or extra relative to the original code. Our alignment formalism discussed in Section 3.2.1 defines alignment as a mapping between operators. Those operators can be connected in various ways by variables while still ensuring semantic equivalence. We define an alignment of variables as a mapping between sets of dataflow connections $A \rightarrow P \cup C \cup F$ between operators. A good variable alignment minimizes the differences between sets. For example, in the `format_name` function in Figure 4, the decompiled code's variable `v1` variable fulfills the same role as the `len` variable, so these two align. Similarly, `v1` aligns with `buffer`. However, the variable `v3` in the decompiled code does not correspond to any variable in the original code; rather, it helps perform part of the functionality of an inlined function. Thus, it is extra relative to the original source code. Another example of a situation where extra variables can occur is when a multi-operator expression in the original code is broken up into two separate expressions with a variable storing the intermediate result. This occurs in Figure 3 where line 9 of the original code is split into lines 7 and 8 of the decompiled code.

C6. Non-idiomatic literal representation: This label is used to label literals used in nonstandard ways. For example, in Figure 4, the string literal `"}\n"` is replaced with the integer constant `2685`.

C7. Obfuscated control flow: This label is used when control flow is used in a way that is not idiomatic. In Figure 4, the `strcat` function is inlined. It may be harder to recognize what the decompiled code is doing relative to the original source code when a function definition is presented inline instead of the name which summarizes that functionality. Another example of C7 is a for-loop used in a non-idiomatic way, such

as the following excerpt from our dataset:

Original Code

```
while (pack->next_object != obj)
{
    pack = pack->next_object;
}
```

Decompiled

```
for (i = a2; a1 != (*((_QWORD *) (i + 64)));
 i = *((_QWORD *) (i + 64)));
```

This example also contains instances of C1 (non-idiomatic dereference), which also contribute to making it harder to read.

C8. Issues in representing global variables: We observed that decompilers sometimes struggled to represent global variables correctly. In our examples, global variable names were not explicitly stripped out. Thus, in some cases, a reference to a global variable could occur by referencing the name of the global variable, just as occurred in the original source code. However, this was not always the case, especially with composite global variables. For example, Figure 4 illustrates an example of a pattern in our dataset where a global `struct` is broken up into multiple variables. The `.name` component of the struct is represented with a reference to a decompiler-generated global variable seemingly named after a memory location.

C9. Expanded symbol: refers to the situation where a macro or similar construct like `sizeof` is represented by its value rather than by the symbol itself. For example, Figure 4 shows an example of how a `sizeof` expression is replaced with a constant.

C10. Use of decompiler-specific macros: Some decompilers define and use macros in decompiled code. Examples of this is the `LODWORD` macro shown in Figure 4 and similar macros used by the Hex-Rays decompiler, which are used in some situations involving type conversion and bitwise operators. As with any feature of decompiled code, these may become less problematic from an interpretability perspective the more a reverse engineer becomes familiar with them, but they represent more information a user must know interpret decompiled code.

C11. Abuse of memory layout: This label refers to the situation where memory is used in a non-idiomatic while being semantically equivalent to the original code. For example, Figure 4 illustrates a situation where two consecutive elements of a struct are each initialized to 0. The decompiled code treats both struct members as a single entity and assigns 0 to the entire construct.

C12. Incorrect return behavior: Sometimes, a decompiled function returns a value while the original function does not or vice versa. In these scenarios, we code C12; there is one sub-label for each of the two situations. Figure 4 shows an instance of C12.a, where a function that is originally `void` has a return value.

C13. Decomposition of a composite variable: When composite variables like `struct`s or arrays are used directly in a function (as opposed to with a pointer), the decompiler may interpret the members of those composite variables as separate variables. Figure 4 illustrates this happening with a global variable.

C14. Type-dependent nonequivalent expression: This occurs as a by-product of the decompiler choosing an incorrect type. When the decompiler chooses an incorrect type, it may cause other expressions to become incorrect relative to the original code such that changing only the type does not fix the code. In Figure 4, the decompiler interprets what should be an `int` array as a `char` array. Accordingly, to ensure the behavior of the function remains the same, the decompiler uses the expression `(4 * a2) * i` as compared with the original code's `i * m` (where `a2` aligns with `m`). If the type in the decompiled code was corrected to `int *`, the resulting code would be incorrect without further changes.

5 Discussion

Our taxonomy can be used to reason about issues in decompiled code. In particular, we are interested in improving the interpretability of decompiler output. We use our taxonomy to reason about how certain classes of issues can be fixed.

In addition, we compare the three decompilers used in the study. We use the taxonomy to identify how each decompiler performs.

5.1 Mitigating Issues

An analysis of the issues mentioned in our taxonomy suggests ways that issues might be mitigated. We caution that our discussion here is based on our subjective judgement; it is impossible to know how technological advances will change decompilation.

Decompilation is a difficult process involving sophisticated static analysis techniques. In recent years, there has been an increasing amount of work focused on decompiling with the assistance of statistical methods like machine learning [8, 9, 13, 16, 17, 19, 24]. In theory, a nondeterministic approach is attractive because a general-purpose technique like a large language model is capable of generating arbitrary strings of text or code. Thus, a nondeterministic technique is capable of producing code very much like the original without the need for sophisticated static analysis. However, these techniques can also make arbitrary mistakes, potentially misleading reverse engineers. Along with these techniques' black-box nature, this may mean that reverse engineers may not trust them, a previously-identified problem in AI [34]. In contrast, as a reverse engineer who participated in Votipka et al.'s [29] study of the reverse engineering process said, "[...] Hex-Rays can be wrong, and disassembly can't be. And this is generally true, but Hex-Rays is only wrong in specific ways."

Thus, we would prefer to suggest solutions to interpretability issues in decompiled code that use more determinism if possible. We differentiate between four different categories of determinism in our analysis:

- Deterministic: The approach we suggest can fix the defect purely through rules-based static analysis. In these cases, we identify the rules that could be used to fix the issue.
- Heuristically optimizable deterministically: These defects can be largely optimized away by following certain rules, but cannot match the original code in every case without nondeterminism.
- Deterministic given types: These defects result from the decompiler lacking complete information about the type of one or more variables. Type prediction itself can't be done deterministically. However, with this class of defects, the decompiler can use a deterministic rules-based approach after receiving the types of relevant variables.
- Nondeterministic: We can't identify a rules-based approach based purely off of the information in the executable. These issues require more nondeterminism than just type information.

Note that correctness issues may be due to decompiler bugs. We have no way of knowing which defects reflect the intended behavior of the decompiler and which are bugs; as a result, we assume that all decompiler output is a result of the decompiler functioning as intended.

In the following discussion, we make references to the relative frequencies of certain labels. We caution that our sample may not be representative of software in general, and that these relative frequencies may differ in other software.

We organize our discussion by the degree of nondeterminism our suggested fixes require, beginning with deterministic fixes.

5.1.1 Deterministic

Some defects of decompiled code are fixable with solely rules-based approaches. They are relatively rare. It appears the sophisticated static analysis techniques used by modern decompilers already take advantage of most deterministic opportunities for improving interpretability.

C10 issues (use of decompiler-specific macros) only occurred in examples decompiled by Hex-Rays. This indicates by example that it is possible to decompile complex these operations without decompiler-specific macros. Figure 5 shows an example of a C10.a issue that exists when Hex-Rays decompiles an executable but not when Ghidra does. Some users of Hex-Rays may find these macros helpful because they help explicitly specify what is happening to specific bytes in an expression. However, in general, we do not know if they are

Original Code

```
1  sreg |= 1 << 7;
```

Decompiled by Hex-Rays

```
1  LOBYTE(result) = sreg | 0x80;
2  sreg = result;
```

Decompiled by Ghidra

```
1  sreg = sreg | 0x80;
```

Figure 5: How an example bitwise oepration is decompiled by Hex-Rays and Ghidra compared to the original source code. In each instance where we identified code C10 for Hex-Rays, we did not identify it for Ghidra. Label C10 refers to decompiler-specific macros used with bitwise operators; `LOBYTE` here.

more or less preferable to reverse engineers than the original code. In any case, C10 represents a barrier to entry for those unfamiliar with this aspect of the tool.

C6.d. refers to situations where a string is replaced by a reference to another location in the binary, e.g. `fz_strlcpy(param_4,&DAT_00100cdf,param_5)` instead of `fz_strlcpy(buf, "CBZ", size)`. In this case, a rule that allows for the recovery of the string is to go to the location provided and replace the reference with the string at that location.

Finally, C5.a.i (extraneous variable duplicating another variable) can often be addressed deterministically. Variables assigned this code do not align with variables in the original program, copy the value in another variable, and could be replaced with the variable they copied from without altering function semantics. In most cases, duplicate variables are never read from after initialization. These variables can be found and replaced deterministically.

5.1.2 Heuristically optimizable deterministically

These issues can be fixed largely deterministically—in a heuristically optimal way—but cannot match the original source code exactly without nondeterminism.

Unaligned variables (C5)—that is, those that are missing (C5.b) or extra (C5.a) relative to the external code—are a common example of this. Data can flow between operators in various ways. If $a_{\texttt{bar},1} \rightarrow$ `foo`; that is, the first argument to `bar` is the result of evaluating `foo`, we can represent this dataflow in source code by either inlining the two expressions (as in `bar(foo());`) or by assigning the result of `foo` to a variable and passing this to `bar` (as in `v1 = foo(); bar(v1);`).

In our observation, all three decompilers usually opt for the second approach. This results in many extra variables (C5.a) not present in the original source code, cluttering the code and, in our experience, making it harder to follow.

Of course, sometimes it makes sense to use variables to

store intermediate values. When the result of an expression is used more than once, saving the value rather than recomputing it often makes sense. Variables are often needed when their values are updated in a loop body. And in some cases, breaking a long expression down into smaller sub-expressions in a sensible way can help make those complicated expressions easier to understand. Finally, programmers will occasionally declare extraneous variables on purpose, often for type-related reasons, as occurs in this snippet from our dataset:

```
1  void FreeTextStream(void *ios)
2  {
3    TextStream *io = ios;
4    // other code using io but not ios ...
```

The arrangement of variables that is "most readable" is somewhat subjective, and can't be found deterministically.

In general, though, for all three decompilers, extra variables greatly outnumbered missing ones, indicating that decompilers generally tend to err too much on the side of using intermediate variables, at least relative to what the authors of the original code thought was best. In many cases, extraneous variables can be eliminated by applying the following rule: if the variable is assigned to then subsequently read from once, eliminate the variable and inline the two expressions. In fact, this rule applies to `v3` in Figure 3. It is possible that this may help reduce reverse engineers' mental strain because they have to track fewer variables in the decompiled code.

C2.a.i is related to C5.a. C2.a.i refers to situations where an extraneous expression (i.e. one that does not align with anything in the original source code) is used to initialize an extraneous variable. If extraneous variables are identified and the extraneous expression has no side effects (which is usually the case), then the whole extraneous initialization statement can be removed.

Incorrect return behavior (C12) is another instance of a heuristically optimizable issue. While types (including functions' return types) are out of scope for this study, we do consider whether or not a function returns a value—this is a correctness issue. Thus, we consider two possible cases for incorrect return behavior: C12.a (return value for void function) and C12.b (no return value for non-void function). A deterministic rule that sometimes fixes C12.a is to make a decompiled function a void function if no function that calls it collects a return value from it. This rule does not work all of the time, however, in the case of functions that do have a return value that just so happens to be unused by all other functions in the program. Interestingly, it appears that Ghidra may follow this rule, while Hex-Rays and retdec seemingly do not. Accordingly, our Ghidra sample has no C12.a labels but falls afoul of C12.b.

Thus, applying this rule incurs a trade-off. We endorse its use, however, because the code to prepare a return value that is never used can be thought of as extraneous from the perspective of the program as a whole. Eliminating it may in fact enhance clarity. Meanwhile, unnecessary extra code for returning values unnecessarily can clutter up functions, sometimes significantly. Additionally, C12.b also seems to be less common than C12.a, though we caution that this observation may be due to sample size.

5.1.3 Deterministic given types

We refer to some codes as "deterministic given types." Many codes fall into this category: C1, C3, C8, C11, C13, and C14. Type prediction itself is nondeterministic in general. However, correctly predict the type of the variables involved, and each of these issues can be resolved deterministically. Some decompilers, like Hex-Rays, already feature an API to re-decompile a function when type information is given to correct these issues; that is, the deterministic piece is already built into these decompilers.

There is existing work on type prediction. Chen et al. [9] develop a probabilistic model, DIRTY, for predicting types in decompiled code, though their method is limited in two ways. First, they predictions are over a fixed set of types, leaving the model unable to generalize to types outside of this set. Second, their prediction for a given variable is based on the memory layout of that variable on the stack and how that variable is used within a single given function. This is problematic in the case of a very common class of variables: pointers, especially pointers to composite types like `struct`s. The memory layout of a pointer variable on the stack is simply a single value representing the memory location of the data.

However, it is still possible to infer the composite types being pointed to based on how these types are accessed. Accesses of different parts of the composite type are, in the general case, spread out throughout multiple functions. Any individual function might only operate on a subset of a struct's members. Therefore, it might be necessary to examine all functions in which a given composite data type is used to determine what that type is. A struct may have fields that are never used, which would be difficult to predict accurately. However, it still may be possible to predict a struct with the subset of fields that are used in the program.

We identify general type prediction as a major opportunity to significantly improve the output of decompilation.

Type prediction can also often help with better representing literals in certain situations. The decompiler often represents character literals as small, positive integers (C6.a); when these integers are used in conjunction with `char` variables, it may be reasonable to convert each integer literal to the corresponding character literals (i.e. `65` to `'A'`). C6.c is a rare label that refers to a situation where the decompiler represents a negative integer as a large, positive integer that would overflow the appropriately sized integer type. If that integer type is known, the literal can be correctly and deterministically transformed into a negative number in the decompiled code.

5.1.4 Nondeterministic

Issues identified by nondeterministic labels require arbitrary additions and deletions to the decompiled code to match the original source code.

The label C2 refers to missing or extra code in the decompiled code relative to the original. C2.b (missing code) issues are usually serious, and are discussed at the end of the section.

In some cases it is more tractable to address C2.a (extra statement) issues—that is, to eliminate the extra statement. In fact, C2.a.i (extra statement for initializing extraneous variables) can often be eliminated via a deterministic rules-based approach after extraneous variables (C5.a) are identified as discussed in Section 5.1.2. Similarly, extra code associated with extraneous return behavior (C12.a) can also be eliminated once those functions are identified. While most C2.a issues fall into one of these two categories, we were unable to find any common patterns amongst the others. Identifying extra code likely requires nondeterminism in the general case from a static analysis perspective.

The label C6 encapsulates non-idiomatic literal representation. In most cases, this issue can be fixed deterministically or with only type information, but there are corner cases which require nondeterminism.

The label C7 refers to non-idiomatic control-flow representations. This is a good use case for function-level nondeterministic modeling. In C7, the semantics of the code are presented correctly, but in a confusing manner. Because all of the correct semantics are present, the non-idiomatic code is reasonably predictive of the more idiomatic version.

C9 labeled issues (expanded macros and other symbols) generally require nondeterminism to address. This is especially true for user-defined macros, which in our sample overwhelmingly were used to "name" a constant (like `CRYPT_ERRTYPE_ATTR_ABSENT` representing `3`). These macros can be useful because they communicate information about the meaning of certain constants. Addressing C9 issues may require whole-program level information because it is often not clear from a single instance of a constant the meaning that the author assigned that constant. It may be possible, however, for these usage patterns to be teased from the program as a whole. The same is generally true for other macros, though some may be "easier" than others.

C2.b issues occur when functionality is missing, and C4 represents most types of semantic nonequivalence between two pieces of code. These are particularly difficult issues to handle. With most other issue labels, the semantics of the program as provided by the decompiler are predictive, sometimes in a roundabout way, of the target original code. However, with C2.b and C4, this is not the case. Thus, we wouldn't generally expect a predictive model to be able to correctly fix them in general. It might be possible to infer the correct behavior if a nondeterministic technique is able correctly infer the purpose of the program as a whole. However, these issues might be best addressed by refining the rules decompilers use to generate source code.

5.2 Comparison of Decompilers

To ensure the robustness of our codebook, we examined the same set of functions decompiled with the Hex-Rays Decompiler, Ghidra, and the retdec decompiler. With a common standard taxonomy of decompiler interpretability issues, we compare the three decompilers.

5.2.1 Hex-Rays Decompiler

The Hex-Rays decompiler is a popular commercial decompiler sold with IDA reverse engineering tools. Hex-Rays tended to have many extraneous variables (C5.a). As noted in Section 4, Hex-Rays sometimes uses custom macros (C10) along with bitwise operators or type conversion operations. Hex-Rays tended to struggle with global composite variables, treating them as if they were separate global variables some with names seemingly derived from memory locations (e.g. `dword_9368`). Hex-Rays also tends to assume a return value for many functions. We recorded no instances of C12.b. (no return value for non-void functions) but numerous instances of C12.a (return value for void function). In combination with the tendencies of Hex-Rays to create variables to store those extraneous return values and to favor a single return statement at the function's end rather than multiple throughout, Hex-Rays sometimes added many extra lines of unnecessary code.

5.2.2 Ghidra

Ghidra is an open-soruce decompiler developed by the National Security Administration. Like Hex-Rays, Ghidra tends to use many extraneous variables (C5.a), though we observed expression inlining in a few cases. We observed no instances of C5.a.i. (extraneous variable duplicating another variable), while we did observe this for Hex-Rays, even on the same functions. Like Hex-Rays, Ghidra tends to struggle with global variables, though interestingly sometimes on different functions. In one case, Ghidra recognized an issue with overlapping symbols at the same address and provided a comment warning about it. As discussed in Section 5.1.2, Ghidra is more conservative with return values; we observed no instances of C12.a but instead a single instance (in our sample) of C12.b (no return value for non-void function).

5.2.3 Retdec

Retdec, an open-source "retargetable decompiler", is a decompiler inspired by LLVM's retargetable nature. Like the other decompilers, retdec uses many extraneous variables (C5.a). However, retdec is also the only of the three decompilers to create extraneous variables that represent individual `struct`

members. Retdec has many issues with representing global variables (C8), and unlike the others, used generic placeholders instead of global variable names (like `g2`) even when those global names were available. Perhaps most concerningly, retdec had many more instances of C4 (nonequivalent expression) than Ghidra or Hex-Rays. Retdec has a tendency to replace some expressions, especially string literals, with the constant 0.

6 Threats to Validity

There are several threats to validity resulting from the dataset's composition and construction. In some security applications, binaries may be obfuscated, making them more difficult to reverse engineer. The dataset we drew from did not include any obfuscated binaries. It also consisted largely of C projects from GitHub, of which a very small minority may be malware. It is possible that there are different interpretability issues that we did not consider that would be exposed had we performed this research on obfuscated binaries or malware, which may be disproportionately common in security settings.

Additionally, the dataset was constructed by attempting to compile projects found on GitHub. The sample is necessarily biased towards those projects that had recognizable build scripts and further, that built.

Finally, the dataset that we drew from performed a filtering step that removed grammatically invalid examples. Unfortunately, this filtering step filtered out some grammatically valid examples as well. This problem largely affected functions where `struct` type names were declared using the `struct` keyword in the function. Struct variables whose types were declared with `typedef`ed names were unaffected, and thus our data includes many struct-typed variables. It is possible, though unlikely, that this caused a certain type of decompiler issue to be excluded from our sample, though we found no evidence of this during spot-checks.

7 Conclusion

In this work, we study the characteristics of decompiled code that make it difficult to interpret. We develop a strategy for identifying and classifying interpretability defects in decompiled code. We then perform multi-step qualitative study of interpretability defects, building a taxonomy. We analyze our taxonomy, identify patterns in our data, and suggest how different classes of defects could be addressed.

We observe that a large group of defect types can be fixed simply by identifying variable types and allowing the decompiler to re-construct the code with the new type information. We observe that many other labels require nondeterminism to be addressed in general but sometimes can be heuristically optimized with deterministic rules.

Based on our results, we identify several promising future directions for work on improving the output of decompilers. First, techniques with the ability to predict types in general, including the types of pointers, have the ability to help address a large class of issues. Second, decompilers themselves could adopt a collection of rule-based approaches which address some issue classes and minimize the impact of others. Finally, we see the opportunity for nondeterministic techniques, especially scaled to be multi-function or whole-program level, to broadly address many classes of errors including some of the most difficult issues and corner-cases that other techniques do not solve.

Acknowledgments

Acknowledgments redacted to protect anonymity.

Availability

We make available all coded examples across all three compilers and all coders. We also provide a comprehensive codebook with a definition of each code as well as numerous examples.

References

[1] Ali Al-Kaswan, Toufique Ahmed, Maliheh Izadi, Anand Ashok Sawant, Prem Devanbu, and Arie van Deursen. Extending source code pre-trained language models to summarise decompiled binaries. In *International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 2023.

[2] Bowen Alpern, Mark N Wegman, and F Kenneth Zadeck. Detecting equality of variables in programs. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 1–11, 1988.

[3] Ron Artstein and Massimo Poesio. Inter-Coder Agreement for Computational Linguistics. *Computational Linguistics*, 34(4):555–596, 12 2008. ISSN 0891-2017. doi: 10.1162/coli.07-034-R2.

[4] Pratyay Banerjee, Kuntal Kumar Pal, Fish Wang, and Chitta Baral. Variable name recovery in decompiled binary code using constrained masked language modeling. *arXiv preprint arXiv:2103.12801*, 2021.

[5] Virginia Braun and Victoria Clarke. *Thematic analysis.* American Psychological Association, 2012.

[6] Kevin Burk, Fabio Pagani, Christopher Kruegel, and Giovanni Vigna. Decomperson: How humans decompile and what we can learn from it. In *31st USENIX Security*

Symposium (USENIX Security 22), pages 2765–2782, 2022.

[7] Juan Caballero and Zhiqiang Lin. Type inference on executables. 48(4):65, 2016.

[8] Ying Cao, Ruigang Liang, Kai Chen, and Peiwei Hu. Boosting neural networks to decompile optimized binaries. ACSAC '22, page 508–518, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450397599. doi: 10.1145/3564625.3567998. URL https://doi.org/10.1145/3564625.3567998.

[9] 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 *31st USENIX Security Symposium (USENIX Security 22)*, pages 4327–4343, 2022.

[10] Yaniv David, Uri Alon, and Eran Yahav. Neural reverse engineering of stripped binaries using augmented control flow graphs. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–28, 2020.

[11] Steffen Enders, Mariia Rybalka, and Elmar Padilla. Pidarci: Using assembly instruction patterns to identify, annotate, and revert compiler idioms. In *International Conference on Privacy, Security and Trust (PST)*, pages 1–7. IEEE, 2021.

[12] Steffen Enders, Eva-Maria C Behner, Niklas Bergmann, Mariia Rybalka, Elmar Padilla, Er Xue Hui, Henry Low, and Nicholas Sim. dewolf: Improving decompilation by leveraging user surveys. *arXiv preprint arXiv:2205.06719*, 2022.

[13] Cheng Fu, Huili Chen, Haolan Liu, Xinyun Chen, Yuandong Tian, Farinaz Koushanfar, and Jishen Zhao. Coda: An end-to-end neural program decompiler. *Advances in Neural Information Processing Systems*, 32, 2019.

[14] Ilfak Guilfanov. Decompilers and beyond. In *Black Hat USA*, 2008.

[15] Jingxuan He, Pesho Ivanov, Petar Tsankov, Veselin Raychev, and Martin Vechev. Debin: Predicting debug information in stripped binaries. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, pages 1667–1680, 2018.

[16] Deborah S Katz, Jason Ruchti, and Eric Schulte. Using recurrent neural networks for decompilation. In *International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 346–356. IEEE, 2018.

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

[18] Shahedul Huq Khandkar. Open coding. *University of Calgary*, 23:2009, 2009.

[19] Jeremy Lacomis, Pengcheng Yin, Edward Schwartz, Miltiadis Allamanis, Claire Le Goues, Graham Neubig, and Bogdan Vasilescu. DIRE: A neural approach to decompiled identifier renaming. In *International Conference on Automated Software Engineering*, ASE '19, pages 628–639, San Diego, California, November 2019. IEEE.

[20] JongHyup Lee, Thanassis Avgerinos, and David Brumley. TIE: Principled reverse engineering of types in binary programs. In *Proceedings of the Network and Distributed System Security Symposium*, February 2011.

[21] Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. Automatic reverse engineering of data structures from binary execution. In *Proceedings of the Network and Distributed System Security Symposium*, 2010.

[22] Zhibo Liu and Shuai Wang. How far we have come: testing decompilation correctness of c decompilers. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 475–487, 2020.

[23] James Mattei, Madeline McLaughlin, Samantha Katcher, and Daniel Votipka. A qualitative evaluation of reverse engineering tool usability. In *Proceedings of the 38th Annual Computer Security Applications Conference*, pages 619–631, 2022.

[24] Vikram Nitin, Anthony Saieva, Baishakhi Ray, and Gail Kaiser. Direct: A transformer-based model for decompiled variable name recovery. *NLP4Prog 2021*, page 48, 2021.

[25] Kexin Pei, Jonas Guan, Matthew Broughton, Zhongtian Chen, Songchen Yao, David Williams-King, Vikas Ummadisetty, Junfeng Yang, Baishakhi Ray, and Suman Jana. Stateformer: Fine-grained type recovery from binaries using generative state modeling. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, pages 690–702, 2021.

[26] Barry K Rosen, Mark N Wegman, and F Kenneth Zadeck. Global value numbers and redundant computations. In *Proceedings of the 15th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 12–27, 1988.

[27] Edward J. Schwartz, JongHyup Lee, Maverick Woo, and David Brumley. Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring. In *Proceedings of the USENIX Security Symposium*, 2013.

[28] M. Shaw. Abstraction techniques in modern programming languages. *IEEE Software*, 1(04):10–26, oct 1984. ISSN 1937-4194. doi: 10.1109/MS.1984.229453.

[29] Daniel Votipka, Seth Rabin, Kristopher Micinski, Jeffrey S Foster, and Michelle L Mazurek. An observational investigation of reverse engineers' processes. In *29th USENIX Security Symposium (USENIX Security 20)*, pages 1875–1892, 2020.

[30] K. Yakdan, S. Dechand, E. Gerhards-Padilla, and M. Smith. Helping johnny to analyze malware: A usability-optimized decompiler and malware analysis user study. In *2016 IEEE Symposium on Security and Privacy (SP)*, pages 158–177, May 2016. doi: 10.1109/SP.2016.18.

[31] 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 *Proceedings of the Network and Distributed System Security Symposium*, 2015.

[32] 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 *Proceedings of the IEEE Symposium on Security and Privacy*, 2016.

[33] Miuyin Yong Wong, Matthew Landen, Manos Antonakakis, Douglas M Blough, Elissa M Redmiles, and Mustaque Ahamad. An inside look into the practice of malware analysis. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, pages 3053–3069, 2021.

[34] Yunfeng Zhang, Q Vera Liao, and Rachel KE Bellamy. Effect of confidence and explanation on accuracy and trust calibration in ai-assisted decision making. In *Proceedings of the 2020 conference on fairness, accountability, and transparency*, pages 295–305, 2020.

[35] Lukás Ďurfina, Jakub Křoustek, and Petr Zemek. Psybot malware: A step-by-step decompilation case study. In *2013 20th Working Conference on Reverse Engineering (WCRE)*, pages 449–456, 2013. doi: 10.1109/WCRE.2013.6671321.

Here, we list the differences between decompiled and original code that we counted as benign. These differences did not receive a code and were not counted as interpretability issues.

- Rearranged statement order that does not change function semantics
- Rearranged expression order that does not change function semantics. For example, in the decompiled code below, the expression multiplying the two arguments is inlined into the mm_malloc and memset function calls. This is semantically equivalent to the original code, and is not definitively more or less idiomatic than the original.

Original Code

```
void *mm_calloc(size_t nmemb, size_t size)
{
  size_t bytes = nmemb * size;
  void *newptr;
  newptr = mm_malloc(bytes);
  memset(newptr, 0, bytes);
  return newptr;
}
```

Decompiled Code

```
void *mm_calloc(long long a1, long long a2)
{
  void *s;
  s = (void *) mm_malloc(a2 * a1, a2);
  memset(s, 0, a2 * a1);
  return s;
}
```

- Inverted conditional statement clauses. Sometimes the decompiler will negate the conditional of an if statement and switch its body and else clauses or do something equivalent, as shown in the example below. Because it is neither ordering is particularly more idiomatic than the other, this sort of change does not receive a label.

Original Code

```
if (DaoIO_CheckMode(self, proc, DAO_STREAM_WRITABLE) ==
      0)
  return;

DaoIO_Writef0(self, proc, p + 1, N - 1);
```

Decompiled Code

```
result = DaoIO_CheckMode(*a2, a1, 8);
if ((_DWORD) result)
  result = DaoIO_Writef0(v5, a1, (long long) (a2 + 1),
       v4 - 1);

return result;
```

- Curly brackets on single-statement conditional
- < to <=, with appropriate operand adjustment
- New-style parameter type declarations instead of old-style.

Original Code

```
int my_index(S, M)
char *S;
char M;
{
// ...
```

Decompiled Code

```
long long my_index(const char *a1, char a2)
{
// ...
```

- Loop break-return pattern instead of in-loop return
- Statement blocks or extraneous loop for organization not reproduced.

Original Code

```
do
{
  if (mk_write_id(c, id) < 0)
    return -1;

}
while (0);
```

Decompiled Code

```
if (((signed int) mk_write_id(a1, a2)) < 0)
  return 0xFFFFFFFFLL;
```

- Const introduced in decompiled code where appropriate.
- Dropped extraneous assert statement
- Condensed extraneous statements or expressions
- Add explicit types to integer literals (e.g. LL)
- Missing inline keyword
- Variable declaration and initialization on different lines
- i++ vs ++i as statement.
- Missing `volatile` or `static` keyword.
- Return statement at the end of a void function.
- Different print statements than intended.

C1. Non-idiomatic dereference
C1.a. Non-idiomatic struct dereferences
C1.a.i. Pointer arithmetic to access struct members
C1.a.ii. Array access to access struct members
C1.a.iii. Pointer dereference to access first struct member
C1.b. Non-idiomatic array dereferences
C1.b.i. Pointer dereference to access array members
C1.b.ii. Pointer arithmetic to access array members

C2. Missing or extraneous statements
C2.a. Extraneous statement
C2.a.i. Extra statement to initialize extraneous variable
C2.b. Missing statement

C3. Typecast Issues
C3.a. Extraneous typecasts
C3.b. Missing typecasts

C4. Nonequivalent expression
C4.a. Incorrect arguments
C4.a.i. Extra arguments
C4.a.ii. Missing arguments
C4.a.iii. Unused missing arguments
C4.b. Equivalence depends on behavior of external code
C4.c. Extra & when accessing global variables

C5. Unaligned variable
C5.a. Extraneous variable C5.a.i. Extraneous variable duplicating another variable
C5.b. Missing variable

C6. Non-idiomatic literal representation
C6.a. Character literals as integers
C6.b. String literal as single integer
C6.c. Very large positive integers for negative integers
C6.d. String replaced with reference to undeclared or global variable

C7. Obfuscating control-flow refactorings
C7.a. While loop as non-canonical for loop
C7.b. For loop as while loop
C7.c. Inline function definition instead of function call
C7.d. Deconstructed ternary

C8. Issue in representing global variables

C9. Uses Expanded Macros
C9.a. Expanded standard macro
C9.b. Expanded user-defined macro

C10. Use of nontype decompiler-specific macro
C10.a. Bitwise operators with decompiler-specific macro

C11. Abuse of memory layout

C12. Incorrect return behavior
C12.a. Return value for void function
C12.b. No return value for non-void function

C13. Decomposition of a composite variable into multiple variables

C14. Type-dependent incorrect expression

Figure 6: The complete codebook.