A Taxonomy of C Decompiler Fidelity Issues

Luke Dramko*, Jeremy Lacomis*, Edward J. Schwartz†, Bogdan Vasilescu*,
Claire Le Goues*

*Carnegie Mellon University. {lukedram, jlacomis, bogdanv, clegoues}@cs.cmu.edu
†Carnegie Mellon University Software Engineering Institute. eschwartz@cert.org

Abstract

Decompilation is an important part of analyzing threats in computer security. Unfortunately, decompiled code contains less information than the corresponding original source code, which makes understanding it more difficult for the reverse engineers who manually perform threat analysis. Thus, the fidelity of decompiled code to the original source 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 from C-language executables 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 four different decompilers. Using thematic analysis, we 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 fidelity 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 [36]. An example of this can be seen in the study by Ďurfina et al. [43], where analysts employed a decompiler to analyze the `Psyb0t` worm, a piece of malware that infects routers to build a botnet.

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 original source definition. Decompilers cannot 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.

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 [34], 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.

Published in the Proceedings of the 2024 USENIX Security Symposium

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.

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 [18, 26, 33, 38, 39]. Despite this, decompilers are one of the most popular tools used by reverse engineers.

Figure 1 shows an example of a decompiled function and its original source code definition. Although the decompiled code is C source code,[1] it is arguably quite different from the original. 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 readability and correctness issues *fidelity* issues because they do not faithfully represent the software as intended by its authors. (See Section 3.1 for a discussion of fidelity).

Fidelity 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 high-level program design as the reverse engineer develops an understanding of what the executable binary does [36]. Code that is more faithful to the original source contains more of the abstractions designed to assist with human comprehension of code. Thus, the fidelity of decompiled code to the original source matters, as it can significantly impact reverse engineers' productivity. Evaluating the products of decompilation based on fidelity to the original source is common in existing work [9, 11, 15, 21, 22, 24].

Improving the functionality and usability of decompilers has long been an active research area, with many contemporary efforts [7, 13, 14, 30, 37]. A recent trend in this direction is using statistical methods such as deep learning-based techniques to improve the process of decompilation [12, 15, 19, 21, 32, 42], or augment the output of traditional decompilers [2, 4, 11, 24, 31]. 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 [7, 36, 40], there are many more fidelity issues in decompiled code, and there is relatively little knowledge of what they are, how they vary across decompilers, and what the implications are for learning-based approaches aiming to improve the fidelity of decompiled code to the original source.

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 fidelity issues in decompiled code. Concretely, we start by curating a sample of open-source functions decompiled with the Hex-Rays,[2] Ghidra,[3] retdec,[4] and angr[5] [35] 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 [6], to analyze the decompiled functions for fidelity 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 call *alignment*. Using this abstraction, we define fidelity defects in decompiled code, creating a taxonomy consisting of 15 top-level issue categories with 52 in total. We then 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.

In this study, we focus primarily on decompiled code from C-language binaries; that is, those that were built from C-language source. C is a common source language for malware and other binaries targeted by reverse engineering efforts. Further, most decompilers of machine-code executables generate output in terms of C pseudocode regardless of the language in which the source code for the executable was written. It is unclear what it would mean for a C decompiler to correctly decompile a Golang executable into C, for example, since Golang contains concepts and features without a direct equivalent in C.

We also examine Java decompilation. However, we find that decompiled Java code is of very high fidelity and thus there are few fidelity issues to classify.

Our results are robust both across different researchers as well as the four decompilers we considered.

In summary, we make the following contributions:

- A comprehensive, hierarchical taxonomy of fidelity issues in decompiled code beyond names and types.
- 235 coded decompiled/original function pairs, identify-

[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

[5] https://angr.io/

ing 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 four different modern decompilers.
- A thorough analysis describing classes of decompiler issues and suggestions for how 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 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 [8], but also review some notable security-oriented work developed in recent years. This includes systems that operate on dynamic runtime traces, such as REWARDS [27], and follow-on systems that statically recover the types by analyzing the executable code at rest. TIE [26] is an exemplar static recovery system used in the academic Phoenix [33] and DREAM [38, 39] decompilers. The Hex-Rays decompiler uses its own static type recovery system [18].

Control flow structuring is the process of converting an unstructured control flow graph (CFG) into the structured control flow commands that are more common in source languages, such as if-then-else and while loops. The Phoenix [33] decompiler introduced a control flow structuring algorithm 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 subsequent DREAM [38] and DREAM++ [39] decompilers introduced 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. The RevEngE decompiler [5] provides the ability to do incremental, on-demand decompilation and is designed for human-in-the-loop decompilation.

Some researchers [9, 15, 21, 22] 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. Theoretically, these approaches are capable of generating decompiler output that is identical to the original source code. However, neural decompilation often fails to match the original source code.

2.2 Improving decompilation through learning

Recent 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. [24] and Nitin et al. [31] 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. [11] 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 fidelity 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 [28] 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 characteristics of decompiled code that cause it to differ from the original, including but not limited to 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 fidelity of the decompiler output to the original source.

2.4 The Reverse Engineering Process

There is relatively little work examining how reverse engineers analyze a binary. Votipka et al. [36] conduct detailed

interview studies with 16 professional reverse engineers to develop a three-step process model for reverse engineering: overview, sub-component scanning, and focused experimentation. Burk et al. [7] corroborate these findings. Mantovani et al. [29] study how reverse engineers (both novices and experts) move through the flow graph of the binary. They find that while there is some variance in individuals' strategies, novice reverse engineers tend to move forward (from `main`) more and examine more of a binary, while experts move backwards and forwards through the flow graph as needed and more quickly dismiss irrelevant parts of the binary. Yong Wong et al. [40] investigate the workflows of malware analysts with an emphasis on dynamic analysis through interviews with 21 reverse engineering professionals. They build a taxonomy of reverse engineering processes each with its own unique interplay between static and dynamic analysis.

3 Methodology

To build our taxonomy of fidelity issues in decompiled code, we used open-coding techniques [23] followed by thematic analysis [6] to group these codes into hierarchical themes. In particular, we used pairs of function representations: a decompiled function and the corresponding original function. Open coding is typically used to systematically analyze textual data, such as interview data. 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 Fidelity

We seek to identify, characterize, and catalog the characteristics of decompiled code that make it differ from the corresponding original source code; the original source code is our oracle. Accordingly, the labels in our codebook are presented in terms of *differences relative to the original source code*. Henceforth, we use the term "difference" to refer to the difference between a piece of decompiled code and the corresponding piece of original code.

Broadly, we consider two ways in which decompiled code is not faithful to the original source code:

- *Correctness issues* occur when there is a semantic difference between the decompiled code and the corresponding original code.
- *Readability issues* occur when the code is semantically equivalent but is communicated using different language features that are difficult to interpret.

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 not readability issues, in the supplementary material.

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 here.

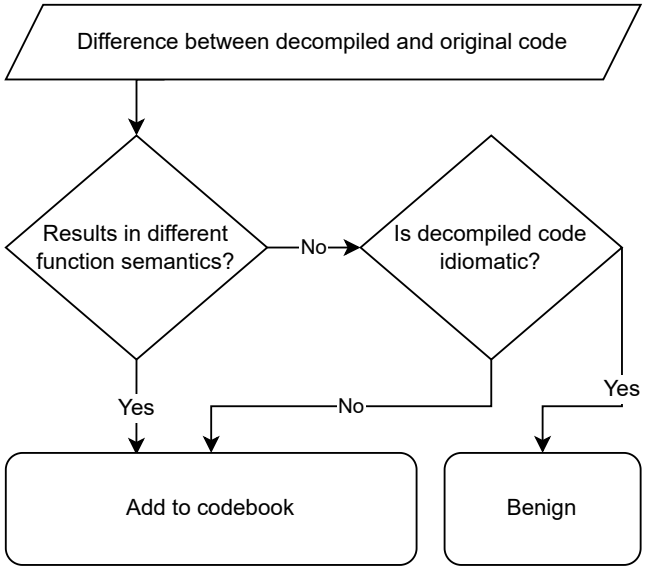

Figure 2: The process by which a difference between the decompiled code and corresponding original code was added to the codebook. Differences added to the codebook are considered meaningful defects, while others are considered benign differences in style. When in doubt, differences were added to the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

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 found that it is often easy to align code by hand, though there are are some challenges to doing so. Figure 3 demonstrates several factors that can complicate determining an alignment. 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 cannot even be thought of as a mapping between individual statements.

Despite this, in most cases, alignment is evident when coding an example. To ensure that our definitions are robust in corner cases, we provide a formal definition of alignment. The formal definition can be found in the supplementary material.

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 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 using a tool called GHCC.[6] 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 function dataset (31% of the total); those with more were sorted into the smaller function dataset (69% of the total). We mostly used functions from the small function dataset, though we also coded ten functions from the large function dataset, as we discuss in the next section.

3.4 Coding Procedure

We selected a random sample of 200 decompiled/original function pairs from the small function dataset. We split the sample into two sets, which we refer to as the small function development set and small function test set. We coded in several phases.

1. **Collecting Differences.** We examined each example in the small function development set, building a set of differences without making judgement as to which semantics-preserving differences constituted readability

[6] https://github.com/huzecong/ghcc

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.

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 classified each semantics-preserving difference 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 all 100 examples in the small function test set 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. While missing variable names and types constitute readability issues, we did not label these issues on our examples because they are trivial and significantly clutter the coded examples.

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 three other decompilers, Ghidra, retdec, and angr, and extracted the requisite functions. We then labeled those examples using the same codebook. Our codebook developed on Hex-Rays worked well, requiring one modification: differentiating between different types of incorrect return behavior. (As discussed in Sections 5.1.3 and 5.2, Hex-Rays exhibited only one type of incorrect return behavior.)

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, which increased 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.

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

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, and gave these to him to code. We then computed interrater agreement using Cohen'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, updating the codebook each time. Our evaluation of inter-coder agreement is shown in Table 1. The agreement is generally good, though worse for rare issues. Additionally, 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.

6. **Generalization Across Languages** In steps 1–5, we focused exclusively on C code. We also wanted to determine if our taxonomy was applicable to other languages. We chose Java, which as an object-oriented and managed language, is significantly different from C and may offer different challenges to decompiler writers. We used the Defects4J dataset [20], which contains seventeen real-world Java projects and their corresponding build scripts. We built each project, randomly selected 25 non-unit-test functions from the dataset, then identified and decompiled the corresponding class files using the JD-GUI decompiler [1]. Finally, we applied our codebook to our sample of the dataset.

4 Taxonomy

Our study yielded a comprehensive codebook of defects in decompiled code. Our codebook is organized hierarchically, with broad, fundamental classes of issues at the top of the hierarchy and more specific instances at lower levels. There are 15 top-level labels, and 52 codes across all levels. In this section, we 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 15 top-level labels at least once. Figure 8 in the appendix shows the complete codebook.

C0. Incorrect identifier name: refers identifier names in the decompiled code, including variable names (C0.a), types (C0.b), and function names (C0.c), which do not match the corresponding identifier names in the original code. Compilers discard essential information about these abstractions: variable and some function names are discarded entirely, while type abstractions are reduced to memory offsets. In the examples we labeled, we did not explicitly label C0 issues because they vastly outnumber the other types of issues and because they have already been studied extensively in existing work [11, 19, 24, 31, 42].

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. More precisely, 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 dereference (`*((_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.

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. We define 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 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` aligns with `len`, and `v2` 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 as illustrated in Figure 5.

C8. Issues in representing global variables: 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

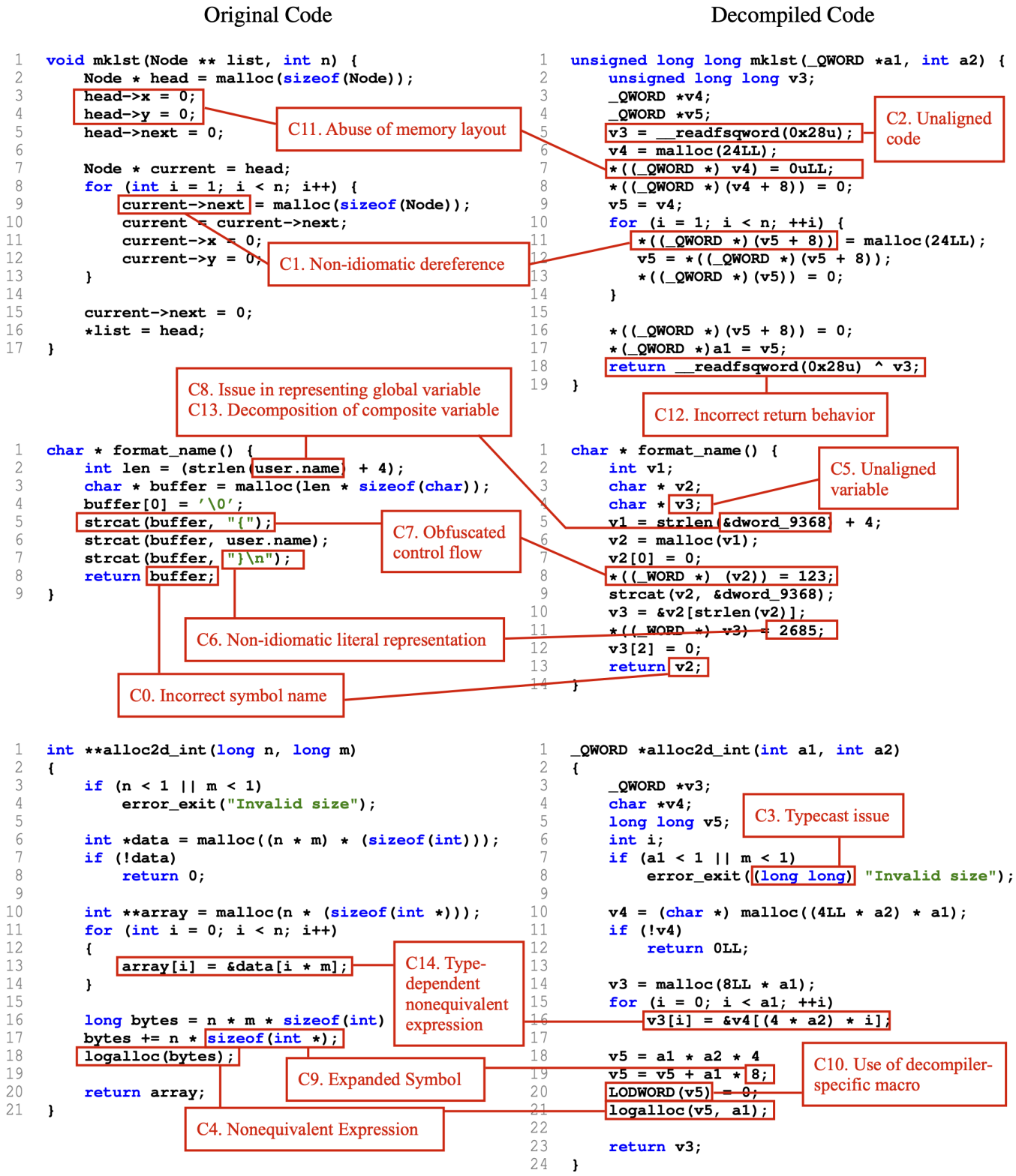

Figure 4: Examples of each of the top-level labels in our codebook. These examples have been artificially constructed by combining code fragments and common patterns from our sample in such a way that all fifteen top-level labels are represented at least once. For legibility, only one instance of each 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.

Original Code

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

Decompiled

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

Figure 5: A decompiled while loop that qualifies as an instance of label C7 (obfuscated control flow). This example also contains instances of C1 (non-idiomatic dereference), which also contribute to making it harder to read.

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. An example 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 the more a reverse engineer becomes familiar with them, but they represent information a user must know to 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 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 fidelity of decompiler output. We use our taxonomy to reason about how certain classes of issues can be fixed. Next, we compare the four decompilers used in the study. We use the taxonomy to identify how each decompiler performs. Finally, we discuss how our taxonomy applies to Java.

5.1 Automatically Mitigating Issues

Our taxonomy classifies the types of defects that can be introduced when decompiling code. A natural question is then whether these differences can be automatically corrected. In this section, we analyze how some classes of defects might be automatically mitigated.

Many aspects of decompilation are fundamentally undecidable because multiple distinct source programs can compile to the same executable. Consequently, decompilers rely on approximation and heuristics to make decisions. Traditionally, decompiler researchers have approached the problem of constructing decompilers by creating sophisticated static analysis algorithms that examine a single executable in isolation. We call such approaches *deterministic* because they will always decompile an executable to the same code. More recently, researchers have been experimenting with *probabilistic* approaches such as machine learning [9, 11, 15, 21, 22, 24, 31]. In addition to examining the executable being decompiled, these approaches incorporate knowledge about the distribution of *other* programs.

Probabilistic approaches are attractive because they can leverage information about the distribution of programs to compute an answer that is—on average—most likely to be correct, even if it is not always correct. In some cases, this can allow predictions to be made that would be impossible using a deterministic approach. As an extreme example, a probabilistic approach can predict a variable's name by examining the context in which it is used, even though a vari-

able's name can be changed without affecting the executable form of the program [24]. The downside to probabilistic approaches is that they are inherently more difficult to understand, which can lead reverse engineers to mistrust them; this is a previously-identified problem in AI [41]. In contrast, as a reverse engineer who participated in Votipka et al.'s [36] study of the reverse engineering process said, "[...] Hex-Rays can be wrong [...] but [it] is only wrong in specific ways."

Thus, we prefer deterministic solutions when possible, because they are easier to understand, and thus to trust. In this vein, we differentiate between four different categories of determinism in our analysis:

- Deterministic: A deterministic function can be applied to the original decompiled code to repair the defect.
- Deterministic given types: Some defects are caused by inaccuracy during type recovery. Type prediction itself cannot be solved deterministically because multiple source programs with distinct types can be compiled to identical executables. However, if the decompiler is provided with accurate types—either by the user, or a system such as DIRTY [11]—then the issue can be fixed by a deterministic analysis.
- Deterministic heuristic: *Most* of these defects can be addressed using a deterministic heuristic—a heuristic that only examines the program being decompiled—but not always. The general case requires probabilistic reasoning.
- Probabilistic: These defects cannot be remedied by any function that only leverages information in the executable. Consequently, the best that can be hoped for is making *likely* decisions based on an expected distribution of programs.

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.

5.1.1 Deterministic

Some types of defects can be repaired by applying a deterministic analysis. However, they are rare. It appears the sophisticated static analysis techniques used by modern decompilers already take advantage of most deterministic opportunities for improving fidelity.

One issue that can be deterministically repaired is C10.a (bitwise operators with decompiler-specific macro). Figure 6 shows an example of a C10.a issue that occurs in Hex-Rays, but not in the other three decompilers. This indicates by example that it is possible to decompile such operations without decompiler-specific macros. Some users of Hex-Rays may find these macros helpful because they help explicitly specify

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 6: How an example bitwise operation is decompiled by Hex-Rays and Ghidra compared to the original source code. Wherever 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.

what is happening to specific bytes in an expression. However, in general, we do not know if they are 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 is another issue that can be repaired deterministically, and refers to situations where a string literal 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, a subset of C5.a.i (extraneous variable duplicating another variable) issues can be addressed deterministically. This can happen when a duplicate variable:

- copies the value in another variable
- could be replaced with the variable they copied without altering function semantics. (In many cases, duplicate variables are never read from after initialization).

To fix these issues, replace the duplicate variable with the variable from which it takes its value.

Deterministic issues can be fixed such that they match the original code exactly.

5.1.2 Deterministic given types

We say some codes can be fixed by a function that is deterministic when given type information. Many codes fall into this category: C0.b, C1, C3, C8, C11, C13, and C14. Type prediction itself is undecidable in general. However, if some technique can correctly predict the type of the variables involved, then each code in this category can be resolved deterministically. Some decompilers, such as 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. [11] develop a probabilistic model, DIRTY, for predicting types in decompiled code. A key limitation of DIRTY, however, is that its 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; different types may have different access patterns. For example, an array may be more likely to have its members accessed sequentially in a loop, while a struct may be more likely to have unordered accesses to arbitrarily-sized memory offsets. Accesses of different parts of a composite type are, in general, spread out throughout multiple functions. Therefore, it might be necessary to examine all functions in which a given composite data type is used to determine what that type is. 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. For example, 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'`).

If the type prediction mechanism is correct, these issues can be fixed so they match the original code exactly. However, probabilistic type prediction may make mistakes.

5.1.3 Deterministic heuristic

Most issues in this class can be fixed by applying a deterministic heuristic. However, the heuristic is known to be incorrect in some cases.

Unaligned Variables Unaligned variables (C5)—that is, those that are extra (C5.a) or missing (C5.b) relative to the original code—are examples of unaligned variables. Data can flow between operators in various ways. If 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 four 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 or even may be necessary (in the case of functions with side-effects). 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 ...
```

In general, though, for all four 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 had written. In many cases, extraneous variables can be eliminated by applying the following rule: if the variable is assigned to once 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.

We empirically validate this heuristic by showing that the vast majority of variables written to and read from once are not present in the original source code. To do this, we use the DIRE dataset [24], a large dataset of decompiled functions wherein generic decompiled variable names are matched with the corresponding variable names in the original source code, if any. We search each function for non-parameter local variables that are written to and read from once. The percentage of these that are decompiler generated (i.e., are lacking a corresponding developer-provided name from the original source code) is 85.9%. In other words, the vast majority of variables that are written to and read a single time are fabrications of the decompiler. Cates et al. [10] evaluate how breaking down expressions into smaller expressions with intermediate variables affects comprehension. They found that breaking down expressions using extra variables impedes comprehension, which also suggests that our inlining heuristic may be reasonable. However, we caution against making strong conclusions due to the study's small number of participants ($n = 6$).

Extraneous Expressions C2.a.i is a common byproduct of C5.a (extra variable in the decompilation), which is discussed in the preceding section. 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. Identifying extraneous variables cannot be done deterministically but can be done heuristically, as discussed above. 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 Incorrect return behavior (C12) is another instance of an issue that a heuristic can remedy. 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 heuristic that often fixes C12.a is to make a decompiled function return `void` if no function that calls it uses its return value. 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, retdec and angr 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 decompiled 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.

We empirically evaluate this rule by measuring how often the return values of functions are used by other functions in the same project (i.e., GitHub repository) across all the projects and functions in the dataset from which we sampled our labeled examples. We perform this study on the original code so our results reflect the ground truth of how developers use return values. For each function in our dataset, we determine whether or not its return value is used in that project by scanning the body of each other function in the project and recording if a call to that function occurs in an expression that uses the function's return value. In addition, we record whether or not the function is void. There are thus four possible classes that a function can fall into:

1. `void` function, return value unused
2. non-`void` function, return value used
3. non-`void` function, return value unused
4. `void` function, return value used.

We count the first and second cases as rule successes, the third case as rule failures, and the fourth as undefined behavior. Undefined behavior was very rare (0.75% of the total). Excluding undefined behavior, our rule is successful 76.1% of the time across the projects in our dataset. To prevent very large projects from dominating the results, this number is calculated by computing the success rate in each project and averaging across projects.

Figure 7 shows the decompiled function from Figure 3 when both of these heuristics are applied. Applying the heuristics increases similarity to the original.

```
1  long long help_object(long long a1, unsigned int a2)
2  {
3    if (!a1)
4      return set_error("help.c", 436LL, 2LL, 0LL);
5
6    return obj_dump_template(a1, a2, get_nestlevel(a2),
         0LL);
7  }
```

Figure 7: The decompiled function from Figure 3 with the heuristics in Section 5.1.3 applied. Applying the heurisitics increases similarity to the original.

5.1.4 Probabilistic

Issues identified as probabilistic cannot be accurately remedied in general, because of the inherent undecidability in decompilation. However, these classes of issues can be improved in practice by techniques such as statistical modeling or machine learning.

Incorrect identifier names (C0) require nondeterminism because the information needed to reconstruct them is partially or completely discarded during compilation.

The label C2 refers to missing or extra code in the decompiled code relative to the original. In general, without access to the original source code, determining what is missing or extra is not possible. There are a few exceptions with C2.a (extra statement), however. As is discussed in Section 5.1.3, extra statements correlating with extraneous variables (C5.a) and extraneous return behavior (C12.a) can be eliminated. C2.b (missing code) issues are usually serious, and are discussed at the end of the section.

The label C7 refers to non-idiomatic control-flow representations. This is a good use case for function-level statistical modeling. In C7, the semantics of the code are presented correctly, but in a confusing manner, as illustrated in Figure 5. 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. Constant macros from common libraries may be guessable (e.g., `O_RDONLY` for C's `open` function).

C2.b issues occur when functionality is missing, and C4

Table 2: A comparison of the frequency at which selected labels occurred on a sample of 25 functions decompiled by four different decompilers. Note that these results may not generalize due to the relatively small sample size. The full table is given in the supplementary material (see Availability).

Label	Hex-Rays	Ghidra	Retdec	Angr
C1.a.i	20	22	20	0
C1.a.ii	8	4	0	20
C4	9	2	28	52
C4.a.i	2	0	3	21
C5.a.	14	16	24	16
C5.a.i	4	0	4	8
C12.a	11	0	12	5
C12.b	0	1	0	3

represents most types of semantic non-equivalence between two pieces of code. These are particularly difficult issues to handle. With most other issue classes, 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 would not 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.

Probabilistic techniques in the best case can fix issues so that they match the original code exactly. However, they are also capable of making arbitrary mistakes.

5.2 Comparison of Decompilers

To ensure the generality of our codebook, we randomly sampled 25 functions from the small function test set (Section 3.4) and decompiled them with the Hex-Rays, Ghidra, and retdec, and angr decompilers. We then coded the output from each decompiler using our codebook. We found that our codebook was applicable to all four decompilers, but that each decompiler tended to make different types of mistakes. In this section, we discuss the differences between the decompilers. The relative frequencies of selected issues across decompilers are show in Table 2.

5.2.1 Hex-Rays Decompiler

Hex-Rays is a popular commercial decompiler sold as an add-on to IDA, a popular, interactive disassembly tool. Compared to the original source code, Hex-Rays created many extraneous variables (C5.a). Hex-Rays also struggled with global composite variables, treating them as if they were separate global variables, with names derived from memory offsets (e.g. `dword_9368`) (C8 and C13). Hex-Rays also introduced a return value for many functions that did not have one (C12.a). However, we recorded no instances of C12.b (no return value for non-void functions). As a side effect of its tendency to create extraneous 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 code.

5.2.2 Ghidra

Ghidra is an open-source decompiler developed by the National Security Agency. Similar to Hex-Rays, Ghidra introduces many extraneous variables (C5.a). Unlike for Hex-Rays, we observed no instances of C5.a.i (extraneous variable duplicating another variable) for Ghidra. As with Hex-Rays, Ghidra struggled with global variables, though interestingly, not always on the same functions as Hex-Rays. 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 5.1.3, Ghidra is conservative with return values; we observed no instances of C12.a (return value for void function) and only a single instance 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 one of the four decompilers to create extraneous variables that represent individual `struct` members (C5.a). Retdec has many issues representing global variables (C8), and unlike the other decompilers, used generic placeholders instead of global variable names (e.g., `g2`) even when those global names were available from symbols in the binary. Perhaps most concerningly, retdec had many more instances of C4 (nonequivalent expression)—28—than Ghidra (2) or Hex-Rays (9), though fewer than angr (52). Retdec has a tendency to replace some expressions, especially string literals, with the constant 0.

5.2.4 Angr

Angr [35] is an open-source binary analysis framework with many features, including a decompiler. One notable feature of angr, unique among the decompilers we analyzed, is its generic structure suggestions, such as

```
typedef struct struct_1 {
    char padding_0[8];
    unsigned long long field_8;
} struct_1;
```

As expected, angr did not always correctly detect structs; it frequently decompiled structs as arrays, resulting in many instances of C1.a.ii (array access to access struct members).

Interestingly, unlike Hex-Rays and Ghidra, angr has instances of both C12.a (return value for void function) and C12.b (no return value for non-void function). More concerningly, angr frequently adds extra arguments to function calls that are not present in the original source (C4.a.i). Generally, angr's decompiler output is more frequently semantically incorrect than all three other decompilers (52 for angr as compared with 9, 2, and 28 for Hex-Rays, Ghidra, and retdec, respectively).

5.3 Applying the Taxonomy to Java

To determine how well our taxonomy applies to a language other than C, we sampled Java functions from the Defects4J dataset [20] and coded them as described in Section 3.4. We found that Java decompiler output has substantially fewer fidelity issues than C decompiled code. In particular, we only found instances of C3 (typecast issues) and C4 (correctness issues). The latter are mostly subtle, and we believe they are caused by decompiler bugs (rather than fundamental undecidability). For example, in one C4 issue we found, a variable in the decompiled code is declared inside an if-statement and used outside of it. This is semantically illegal in Java. In the original code, the variable is declared directly before the if statement. We believe that the additional information stored in Java bytecode makes the process of decompiling Java code substantially more tractable. We suspect this is true for decompiling other managed languages as well.

In summary, while all fidelity issues we discovered in Java decompilation fit within our taxonomy, the naturally high fidelity of Java decompilation meant that there were very few issues to classify.

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. It is unlikely that many of them are malware. It is possible that there are different fidelity 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 Future Work

Our study identifies nontrivial fidelity issues in decompiled code. It is probable, however, that not all of these issues have the same impact on the difficulty of reverse engineering. A challenge for future work is to determine which issues have the largest impact on reverse engineers' productivity.

Existing program comprehension literature, for the most part, does not cover issues in our taxonomy. This is because to our knowledge all existing program comprehension literature focuses on normal (original) source code. Decompiled code is fundamentally different from normal source and thus the issues identified by existing literature in program comprehension [16, 17, 25] differ from those in our taxonomy.

Our work may also be useful for evaluating decompilers. Our taxonomy is defined in terms of alignment. Determining alignment automatically in general is a hard problem; in our study, this was performed by human experts. However, given a suitable technique for determining alignment, our taxonomy could be used to automatically evaluate decompiler output or neural decompilation methods. In particular, numeric scores related to the frequency of each issue in the taxonomy could be reported. This would allow a detailed and nuanced quantitative measurement of the quality of decompiler output.

8 Conclusion

In this work, we build a taxonomy of fidelity issues in decompiled code. We analyze our taxonomy, identify patterns in our data, and suggest how different classes of defects could be addressed.

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

This material is based upon work supported in part by the National Science Foundation (awards DGE2140739, 1815287, and 1910067).

Availability

We make available all coded examples across all four decompilers and all coders. We also provide a comprehensive codebook with a definition of each code, a definition of alignment, and a list of benign (uncoded) issues. This material can be found at https://doi.org/10.5281/zenodo.8419614.

References

[1] Jd-gui 1.6.6, 2019. URL https://github.com/java-decompiler/jd-gui.

[2] 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.

[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] Marcus Botacin, Lucas Galante, Paulo de Geus, and André Grégio. Revenge is a dish served cold: Debug-oriented malware decompilation and reassembly. In *Proceedings of the 3rd Reversing and Offensive-Oriented Trends Symposium*, ROOTS'19, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450377751. doi: 10.1145/3375894.3375895. URL https://doi.org/10.1145/3375894.3375895.

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

[7] 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.

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

[9] 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.

[10] Roee Cates, Nadav Yunik, and Dror G Feitelson. Does code structure affect comprehension? on using and naming intermediate variables. In *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*, pages 118–126. IEEE, 2021.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] Dan Gopstein, Jake Iannacone, Yu Yan, Lois DeLong, Yanyan Zhuang, Martin K-C Yeh, and Justin Cappos. Understanding misunderstandings in source code. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, pages 129–139, 2017.

[17] Dan Gopstein, Hongwei Henry Zhou, Phyllis Frankl, and Justin Cappos. Prevalence of confusing code in software projects: Atoms of confusion in the wild. In *Proceedings of the 15th International Conference on Mining Software Repositories*, pages 281–291, 2018.

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

[19] 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.

[20] René Just, Darioush Jalali, and Michael D Ernst. Defects4j: A database of existing faults to enable controlled testing studies for java programs. In *Proceedings of the 2014 international symposium on software testing and analysis*, pages 437–440, 2014.

[21] 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.

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

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

[24] 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.

[25] Chris Langhout and Maurício Aniche. Atoms of confusion in java. In *2021 IEEE/ACM 29th International Conference on Program Comprehension (ICPC)*, pages 25–35, 2021. doi: 10.1109/ICPC52881.2021.00012.

[26] 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.

[27] 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.

[28] 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.

[29] Alessandro Mantovani, Simone Aonzo, Yanick Fratantonio, and Davide Balzarotti. {RE-Mind}: a first look inside the mind of a reverse engineer. In *31st USENIX Security Symposium (USENIX Security 22)*, pages 2727–2745, 2022.

[30] 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.

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

[32] 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.

[33] 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.

[34] 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.

[35] Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *IEEE Symposium on Security and Privacy*, 2016.

[36] 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.

[37] 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.

[38] 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.

[39] 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.

[40] 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.

[41] 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.

[42] Zhuo Zhang, Yapeng Ye, Wei You, Guanhong Tao, Wenchuan Lee, Yonghwi Kwon, Yousra Aafer, and Xiangyu Zhang. Osprey: Recovery of variable and data structure via probabilistic analysis for stripped binary. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 813–832. IEEE, 2021.

[43] 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.

C0. Incorrect identifier name
C0.a. Incorrect variable name
C0.b. Incorrect type name
C0.c. Incorrect function name

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

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

C3. Typecast Issue
C3.a. Extraneous typecast
C3.b. Missing typecast

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 variable

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 literal as integer
C6.b. String literal as single integer
C6.c. Very large positive integer for negative integer
C6.d. String replaced with reference to undeclared or global variable

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

C8. Issue in representing global variable

C9. Uses expanded symbol
C9.a. Expanded standard symbol
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 8: The complete codebook.