

Neural Decompilation with Minimal Risk

Luke Dramko

CMU-S3D-26-114

July 2026

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

Thesis Committee:

Claire Le Goues, Chair
Rohan Padhye
Edward J. Schwartz
Lin Tan (Purdue University)

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

Copyright © 2026 Luke Dramko

This document is based upon work supported in part by the National Science Foundation (awards DGE2140739, 1123717, CNS1852260, and CCF1910067), Microsoft (award 5008587), and the Software Engineering Institute and the Department of Homeland Security (award FA8702-15-D-0002).

The views and conclusions contained in this document are those of the author and should not be interpreted as representing the official policies, either expressed or implied, of any sponsoring institution, the U.S. government or any other entity.

Keywords: Machine Learning, Decompilation, Reverse Engineering

Abstract

Decompilation is the process of converting executable programs back into source code, reversing compilation. Decompilation is used in several security-related tasks for which the source code of the program of interest is typically unavailable, including malware analysis, vulnerability research, and patching legacy software. Unfortunately, a significant amount of information is discarded during compilation, which means that information is unavailable for decompilers to use when reconstructing the source code. As a result, decompiled code is missing details like variable names, user-defined types, and comments, and is often structured in substantially different ways. The end result is that decompiled code is substantially more difficult to read than traditional source code, making reverse engineering a painstaking process. The key goal of this thesis is a method to recover these missing details with machine learning while not undermining the stronger correctness properties that deterministic decompilation provides.

In the first thrust, to gain insights into the nature of decompiled code with an eye towards how it can be improved for reverse engineers, we conduct an empirical study of decompiled code. In particular, we build a comprehensive, hierarchical taxonomy of the ways in which decompiled code is unfaithful to the original source code from which it was derived (through compilation and subsequent decompilation). Two key conclusions from an analysis of our taxonomy are (1) a lack of type information, especially user-defined-types, are directly or indirectly involved in over half (8/14) of all top-level issues, and (2) arbitrary textual transformations are required to fix all issues generally.

Leveraging these insights, in the second thrust, we built a neural decompilation tool, IDIOMS, to help address the fidelity issues with decompiled code. Because type information and much of the information needed to rewrite the decompiled code is discarded during compilation, our approach is *probabilistic*, based on machine learning. IDIOMS consists of a family of fine-tuned language models of code. IDIOMS jointly rewrites decompiled code and provides definitions for all user-defined types used within the rewritten code. IDIOMS takes as input the decompiled form of a function and, because code throughout an executable program can inform the prediction of user-defined types, other nearby functions in the binary.

The final thrust addresses a fundamental limitation of neural decompilation tools like IDIOMS: incorrect predictions (mispredictions). Especially concerning are semantic mispredictions: those that do not faithfully represent the behavior of the underlying binary. Mispredictions are a critical threat to validity for neural decompilers because a misprediction could mislead a reverse engineer into drawing incorrect conclusions about the executable. We introduce a technique, FAULTLESS, that checks for the absence of semantic mispredictions by comparing a neural decompiler’s predicted code with the decompiled code, allowing the reverse engineer confidence that the rewriting can (or cannot) be trusted. FAULTLESS can also be applied to evaluate a neural decompiler by comparing the predicted and original code.

Acknowledgments

This dissertation would not have been possible without the support and advice of so many.

First, I'd like to thank my PhD advisor, Claire Le Goues, for six years of top-notch advice and a wholehearted investment in helping me succeed. I am grateful for her continuous guidance that helped me produce the work that culminated in this dissertation and perhaps more importantly grow so much as a researcher. I'd also like to thank my frequent collaborator and unofficial second advisor, Edward Schwartz, always just a slack message away. Without our numerous conversations about reverse engineering and his excellent feedback, I would not be where I am today.

I'd also like to thank the other members of my thesis committee, Rohan Padhye and Lin Tan. They were wonderful to work with and helped guide this work to be the best it could be. I'd also like to thank Bogdan Vasilescu, Graham Neubig, Jeremy Lacomis, Jeffrey Gennari, Qibin Chen, Uri Alon, and Alex Shypula for their help, advice, and all of the conversations that helped shaped my work. I'd especially like to thank Bogdan for inspiring the first chapter of this thesis.

I'm also grateful the members of the CMU community who helped give me such good times: Sam Estep, Kush Jain, Kaia Newman, Vasu Vikram, Manisha Mukherjee, Long Nguyen, Nadia Nahar, Peter Carragher and countless others. I'd also like to thank all of the members of squaresLab who helped make the lab such a welcoming place and who gave me advice on so many talks and papers.

I'd like to thank my undergraduate REU advisors, Heather Miller and Michael Dietze, for helping me start my journey in research. I would not be here today without their patient mentorship.

Finally, I'd like to thank my family, who were always there for me in good times and bad. I could not have done this without their support.

Contents

1	Introduction	1
1.1	Thesis Statement	3
1.2	Terminology	3
1.3	Outline of Contributions	4
2	Background and Related Work	7
2.1	Decompilation	7
2.2	Machine Learning: Background	8
2.3	Machine Learning for Decompilation	9
2.3.1	Neural Decompilation	9
2.3.2	Targeted Abstraction Prediction	10
2.4	Program Equivalence	11
3	A Taxonomy of C Decompiler Fidelity Issues	13
3.1	Introduction	13
3.2	Methodology	15
3.2.1	Fidelity	15
3.2.2	Coding Standards	16
3.2.3	Dataset	18
3.2.4	Coding Procedure	18
3.3	Taxonomy	20
3.4	Discussion	23
3.4.1	Automatically Mitigating Issues	24
3.4.2	Comparison of Decompilers	30
3.4.3	Applying the Taxonomy to Java	31
3.5	Threats to Validity	31
3.6	Conclusion	32
4	Idioms: Neural Decompilation With Joint Code and Type Prediction	33
4.1	Introduction	33
4.2	Approach	36
4.2.1	Modeling	37
4.2.2	Dataset	38
4.2.3	Training Details	41

4.3	Experimental Design	42
4.3.1	Datasets and Baselines	42
4.3.2	Metrics	43
4.3.3	Experimental Methodology	44
4.3.4	Dataset and Prediction Processing	46
4.4	Results	47
4.4.1	RQ1: Model Performance and Trends	47
4.4.2	RQ2: The Challenge of Real-World Code with UDTs	49
4.4.3	RQ3: The Role of Neighboring Context	49
4.5	Limitations and Threats to Validity	49
4.6	Conclusion	50
5	Robust Evaluation and Validation of Neurally Decompiled Code	51
5.1	Introduction	51
5.2	Faultless	54
5.2.1	Defining Equivalence	55
5.2.2	Execution Model	57
5.2.3	Memory Model	59
5.2.4	The Compiler	66
5.2.5	The Interpreter	68
5.2.6	The Proof Engine	71
5.3	Evaluation	76
5.3.1	Experimental Design	76
5.3.2	Results	79
5.4	Utility	81
5.4.1	Experiment Methodology	82
5.4.2	Results	83
5.5	Limitations	87
5.5.1	Soundness	87
5.5.2	Completeness	90
5.6	Conclusion	92
6	Conclusion	93
	Bibliography	95

Chapter 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 where the source code may be unavailable. By converting executables into human-readable C-like code, decompilation allows reverse engineers to more effectively understand and respond to the threats posed by malware [104]. An example of this can be seen in the study by Āurfina et al. [116], where analysts employed a decompiler to analyze the `Psybot` worm, a piece of malware that infects routers to build a botnet. 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 [104].

Analyzing and understanding the behavior of executable code is significantly more difficult than analyzing source code due to information that is discarded during the compilation process. Source code is a dual-channel medium, containing a formal channel that specifies execution semantics and a natural (language) channel that communicates information to developers who read and write the code [19]. Unfortunately, while high-level programming languages contain natural-channel abstractions and constructs such as variable names, types, comments, and control-flow structures that make it easier for humans to write and understand code [90], 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 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. Decompilers are one of the most popular tools used by reverse engineers. 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 [40, 65, 89, 107, 108].

One way to help would be to recover the natural-channel abstractions that are designed for human comprehension in the first place. Unfortunately, this cannot be done deterministically be-

<pre> 1 static void enqueue(struct queue *q, struct urb * urb) { 2 if (q->head) { 3 q->tail->next = urb; 4 } else { 5 q->head = urb; 6 } 7 q->tail = urb; 8 urb->next = ((void *)0); 9 } </pre>	<pre> 1 __int64 __fastcall func7(_QWORD *a1, __int64 a2) { 2 __int64 result; // rax 3 4 if (*a1) 5 *(_QWORD *) (a1[1] + 8LL) = a2; 6 else 7 *a1 = a2; 8 a1[1] = a2; 9 result = a2; 10 *(_QWORD *) (a2 + 8) = 0LL; 11 return result; 12 } </pre>
(a) Original Source	(b) Decompiled

Figure 1.1: A function, and the same function having been compiled, then decompiled. The decompiled function is missing abstractions like identifier names and user-defined types which make the code more difficult to read.

cause those details are discarded. However, it is usually the case that natural-channel abstractions and formal-channel semantics are linked: the former is used to describe and give structure to the latter. Moreover, software is *natural*: predictable given context [45]. In other words, programmers are often consistent about how they use programming languages to express ideas in code. For instance, a call to `malloc` is often followed by a `NULL`-check. In combination, these two ideas suggest that it may be possible to *predict* natural-channel abstractions like names, types, and code structure from the formal-channel abstractions that remain in the executable. For instance, the name of a variable that controls the maximum value of an integer used to index into an array is more likely to be called `length` or `size` than it is `filename` or `node`. Such a prediction technique must be fundamentally probabilistic, and thus may occasionally be wrong, at least relative to the original source, in the (by definition) rare case that these abstractions are *unnatural*, such as when a programmer uses an unusual or even misleading variable name. However, there is still the potential to significantly improve the decompiled code.

A rapidly growing body of work leverages these insights to improve decompilation with machine learning. Broadly, this work falls into two categories: techniques that target one or more specific abstractions (e.g. variable names, variable types, function names), updating the placeholder values in the traditional deterministic decompilation [21, 29, 54, 59, 62, 76, 79, 105, 106, 113, 114] and work that generates a complete code snippet (based on the executable, assembly, or deterministic decompilation) [5, 17, 48, 50, 53, 57, 98]. We refer to the former as *targeted-abstraction-prediction* and the latter as *neural decompilation*. Neural decompilation is attractive because it elegantly can predict all natural-channel abstractions at once. Because there are numerous differences between decompiled and original code, as we show in Chapter 3, it is impractical to maintain targeted models that handle each of them. However, it is also inherently more risky because the neural decompiler could mistakenly generate code that does not faithfully represent the semantics of the binary, thereby undermining reverse engineers’ understanding of the executable and thus undermining decompilers’ key purpose in the first place. Beyond this, existing neural decompilation techniques do not handle user-defined composite types, like `structs` and `unions`, well. In particular, they will generate code with these types but not the definition of those types, meaning that their solutions are underspecified. This is a key limitation, because user-defined composite types are very common in real code.

In this dissertation, I present work to help address this gap. First, to gain insight into the nature of decompiled code, I conduct an empirical study of decompiled code, building a taxonomy of the ways in which decompiled code differs from the original. I leverage insights derived from analyzing our taxonomy to inform a new neural decompilation approach. In particular, it jointly generates type definitions along with the code, and, based on the observation that type recovery is a fundamentally interprocedural task, I provide other functions in the binary as context. Neural decompilers are probabilistic and may predict code that does not faithfully represent the binary; they may make semantic mispredictions. To address this, I introduce a new technique that helps detect mispredictions by checking that the code predicted by the neural decompiler is equivalent to a reference version. In a controlled evaluation setting, the reference can be original source code, while in an online reverse engineering scenario the reference code is the decompiled code. Program equivalence is an undecidable problem with no general-purpose solution; different tasks may be best addressed by different techniques. Here, I take advantage of the fact that the original, decompiled, and predicted code should all be, if correct, implementations of the same algorithm and thus have structural similarities.

In this thesis, I focus on decompiler’s targeting C code, which is the language targeted by most commercial decompilers of machine code.

1.1 Thesis Statement

Deterministically decompiled code differs substantially from the corresponding original source code, especially in terms of abstractions designed to aid in program comprehension. Neural decompilation, leveraging interprocedural information to help jointly predict types and code, can help address the deficiencies of deterministic decompilation, while equivalence checking that leverages the structural parallelism between the predicted code and the reference original or decompiled code can measure and mitigate the risk of semantic mispredictions at an acceptable runtime cost.

1.2 Terminology

For clarity, we define terminology that we use throughout this dissertation.

- **Decompiler/Deterministic Decompiler:** a deterministic tool that converts machine code into a higher-level language using program analysis algorithms. Examples include IDA Pro’s decompiler and Ghidra’s decompiler.
- **Neural Decompiler:** a machine learning model that converts a deterministically derived representation of the machine code as input, including the raw machine code, the assembly, or deterministic decompilation, into a higher-level language.
- **Original code/Original source code:** the developer-written source code for a software application.
- **Decompiled code:** code produced when a traditional deterministic decompiler is applied to machine code.

- **Predicted code/Neurally Decompiled Code:** code produced by a neural decompiler. In keeping with machine-learning convention, the code is the model’s prediction.
- **Decompilation Improvement:** adding additional detail to or clarifying information in decompiled code.
- **Idiomatic:** an adjective used to describe code that follows conventions typical of what human developers use. Decompiled code is generally not idiomatic.
- **User-Defined Types (UDTs):** types that are not part of the language definition but which are instead defined in a library or software application. While aliases to primitive types can be thought of as user-defined types, we do not do so here. In Chapter 4, this term refers to `structs` and `unions`.

1.3 Outline of Contributions

This thesis contains three main pieces of work.

A Taxonomy of C Decompiler Fidelity Issues. It is well-known that decompiled code is not idiomatic and significantly differs from the original code. Certain other details are eminently notable: variable names are missing, and type reconstruction, especially for user-defined types, is incomplete. However, beyond this, the nature of decompiled code has not been systematically studied in existing literature. With the aim of better understanding decompiled code to better develop decompilation improvement solutions for it, we conduct an empirical study of decompiled code in which we build a hierarchical taxonomy of the (consequential) ways in which decompiled and the corresponding original code differ. Two important takeaways for neural decompilation improvement are: (1) a lack of type information, especially user-defined-types, is the indirect cause of a variety of issues, and (2) arbitrary textual transformations are required to fix all issues generally. This latter point motivates the need for neural decompilation rather than a more targeted approach focusing on one particular type of decompilation improvement.

This work was originally published in USENIX Security 2024:

Dramko, Luke, Jeremy Lacomis, Edward J. Schwartz, Bogdan Vasilescu, and Claire Le Goues. “A Taxonomy of C Decompiler Fidelity Issues.” In *33rd USENIX Security Symposium (USENIX Security 24)*, pp. 379-396. 2024.

IDIOMS: A Simple and Effective Framework for Turbo-Charging Local Neural Decompilation with Well-Defined Types. Using the insights we gained from our empirical study, we designed IDIOMS, a state-of-the-art neural decompiler. IDIOMS differs from prior work in two key ways. First, we observed that recovering user-defined types is critical to the ultimate syntactic form of the code. Existing tools can predict code that uses UDTs, but they do not define the types they are using. For example, they might predict code where a field of a structure is accessed, but the layout of that structure is not provided. This means that a reverse engineer using these tools does would not know either the type of the field nor its position in the larger data structure, both of which are very useful pieces of information. Second, all existing neural decompilation work focuses on one function at a time. The primary features of the decompiled

code that suggest the layout of a given UDT are the relative offsets in a decompiled field access operation (see lines 5 and 10 of Figure 1.1b for examples) and the context in which they occur. In general, field accesses to a given structure may be spread out throughout the binary. Therefore, to correctly predict a UDT, it may be necessary to look at multiple functions. IDIOMS uses additional functions as context to better predict UDT definitions. IDIOMS achieves state-of-the-art results both on the most challenging existing benchmark. The work also introduces a dataset that features a more realistic concentration of UDTs than existing benchmarks.

This work was originally published in NDSS 2026:

Dramko, Luke, Claire Le Goues, and Edward J. Schwartz. “Idioms: A Simple and Effective Framework for Turbo-Charging Local Neural Decompilation with Well-Defined Types.” In *34th Network and Distributed Security Symposium (NDSS)*, 2025.

FAULTLESS: *Equivalence Checking for Neural Decompilers*. Neural decompilers can produce code that is idiomatic and includes meaningful identifier names and types. However, as probabilistic machine-learning models, neural decompilers may also make mistakes, misrepresenting the semantics of the program under analysis. This incurs a risk of misleading reverse engineers, potentially wasting their time or causing them to make incorrect conclusions about the functionality of the program. As a solution, we introduce, FAULTLESS, a technique check the correctness of neural decompilers’ predictions. When evaluating neural decompilers, the correctness can be checked by comparing predictions with the gold-standard original source code. While comparing with the original source is standard when evaluating a neural decompiler in a controlled experimental setting, a reverse engineer using a neural decompiler in practice does not have access to the original source code. However, deterministically decompiled code is readily available, and has stronger correctness properties than neurally decompiled code, making it a feasible oracle against which correctness can be checked in active reverse engineering scenarios. Program equivalence is a very difficult problem that is undecidable in general. It is made even more difficult here because it is difficult to even compile decompiled or predicted code, and compilation is a prerequisite step to many existing program equivalence techniques. However, the task of neural decompilation affords opportunities as well: the decompiled code and predicted code by construction should follow the same algorithm as the original code. FAULTLESS takes advantage of this fact, ascribing dynamic semantics to single-static assignment [3, 84] ϕ instructions to decompose the equivalence checking into tractable overlapping subproblems. FAULTLESS is effective as both a correctness evaluation technique and for verifying the absence of mispredictions in online reverse engineering scenarios.

Chapter 2

Background and Related Work

Decompilation is a very difficult task, made so by the lossy nature of compilation and the sophistication of the code transformations performed during compilation. In recent years, there has been increasing interest in applying machine learning to decompilation, either replacing the decompilation process entirely or operating on the output of a decompiler in an effort to improve it. Program equivalence has a longer history, but many existing techniques make assumptions that do not hold in the task of neural decompilation. In this section we first provide some background on decompilation and machine learning, then outline related work in machine learning for decompilation and program equivalence.

2.1 Decompilation

Decompilers reverse the process of compilation, converting a lower-level language into a higher-level language. In this work, we are primarily concerned with decompiling machine code into C pseudo-code, though there exist decompilers for other pairs of source and target languages as well, such as those that target transform Java bytecode into Java code. We say C pseudo-code because many decompilers may produce code that resembles C but is not syntactically or semantically valid; decompilers often prioritize readability over correctness. A key challenge of decompilation—and focus of this thesis—is that compilation is lossy; some source-level information is discarded during compilation. Compilers are not injective [88]; that is, textually distinct source code programs can compile into the same machine code. This means that decompilers can at best produce a representative version of the source.

Decompilers work by performing a sequence of steps that is roughly the inverse of a compiler [24, 25]. The first step is disassembly: converting binary machine code into assembly, an equivalent format that is easier to work with. In theory this is a relatively simple translation, but in practice it is challenging to differentiate between code and data, which can be muddled together in modern binary formats. The assembly code is lifted to a higher-level intermediate representation that abstracts away machine-specific details like concrete registers and flags. Some source level concepts like division can be represented by multiple assembly instructions; these patterns are recognized and reversed. The higher-level intermediate representation provides a suitable format for performing analyses, such as those that reconstruct yet higher-level constructs like

control flow structures and variables. In control flow recovery, unstructured control flow (e.g. jump and branch instructions) are converted to structured control flow (e.g. `if`-statements and loops) by analyzing and partitioning the control flow graph. Variables and their types, ubiquitous in source code, don't exist at the machine level; only typeless registers and memory. A given register can represent multiple conceptual variables, even one of different types, in the same function. In type recovery [14, 73], constraints on types are inferred from registers' interaction with instructions (e.g. the argument to a floating-point addition must be floating point) and these constraints are solved. In variable recovery, each register or accessed memory location at each program location could represent a unique variable but these can be merged together to represent conceptual variables. When all analyses are complete, code in a high-level language like C is generated from the representation inferred in the previous passes. Despite the complexity of this process, decompiled code often looks very little like the original. We systematically analyze the differences between decompiled and original code in Chapter 3.

Notably, deterministic decompilers are generally unsound. An exception is byte-equivalent decompilation [87], in which the decompiled code can be re-compiled to produce exactly the same machine code as the original binary, byte-for-byte. While better than regular decompilation, it is also generally very difficult to achieve by following traditional decompilation methods described above. Instead, a guess-and-check approach is used where the source code is iteratively modified and compiled to draw it closer to the goal; Schulte et al. [87] use an evolutionary search algorithm. This approach is not guaranteed to find a byte-equivalent solution.

2.2 Machine Learning: Background

Many state-of-the-art neural decompilation techniques, including IDIOMS (Chapter 4) are based on *causal language models*. In this section, we provide a brief machine learning background, discussing some machine-learning basics, pretraining/finetuning, self-supervision, causal language modeling, adapters, and data leakage.

State-of-the-art language models are neural networks, machine learning models that encode information in terms of real numbers called parameters, which are usually organized in tensors. Inputs are transformed into output predictions by multiplying, adding, or otherwise combining them with the parameters in a sequence of steps dictated by the model architecture. Models learn via training, where the error of the prediction relative to some ground-truth oracle, the label, is quantified, and the parameters are adjusted a small amount to minimize the error via a process called backpropagation [86]. Over time, the model learns to map the inputs to the outputs. Neural networks are universal function approximators—if given enough data and parameters, they can model any function, mapping arbitrary inputs to arbitrary outputs [47].

A key challenge in machine learning is finding a suitable oracle for generating enough input/output pairs to train a model sufficiently well. A popular solution to this challenge is the *pretrain/finetune* paradigm. If a lot of data is available for a task related to the target mapping, the model can be trained first on the related task. The model will generally do much better on the target task with limited quantities of data.

Another popular solution is *self-supervision*: creating an artificial task for which an oracle can be automatically constructed for otherwise unlabeled data. Usually this happens by hiding part

of the input and training the model to predict the hidden part. This is the case in causal language modeling, where the beginning of a text sequence is used to predict the next word/token. (Due to how text is input to the model, text is broken up into a sequence of words or subwords called tokens.)

The aggregation of these effects has led to the creation of large, pretrained, general-purpose causal language models trained on a variety of different types of text, which have shown to produce state-of-the-art results on a variety of different downstream tasks. Some of these models train on source code text in particular; we focus on these in this work.

As state-of-the-art pretrained models have increased in size, finetuning them has become increasingly computationally expensive. In particular, a key challenge is fitting the model and associated training information in the VRAM of the GPUs used to train models efficiently. LoRA adapters [49] help alleviate this issue by storing a high-fidelity, small-memory-size approximation of the collective updates to the parameters that occur during fine-tuning. For each $n \times m$ matrix in the original model, where n and m are large, LoRA stores an $n \times r$ and a $r \times m$ matrix that together, when multiplied, approximate the sum of the changes to the parameters that occur during finetuning. If r is small, the memory savings are substantial. QLoRA [32] increases the savings even further by quantizing—that is, approximating with a representation that uses less memory—the weights of the original pretrained model.

A key risk in machine learning is that of *data leakage*. Machine learning models tend to “memorize” examples that they have been trained on, and as a result, evaluating a model on examples in the training set leads to inflated results that are unrealistic in practice. An important dataset preparation step is to check that the training and evaluation sets are disjoint. As we discuss in Section 4.5 the risk of data leakage via pre-training is relatively low for neural decompilation in particular, but the risk of data leakage during finetuning is higher. In constructing the dataset for IDIOMS, we take aggressive measures to avoid data leakage, outlined in Section 4.2.2.

2.3 Machine Learning for Decompilation

A large body of prior work has applied machine learning to the problem of decompilation in an attempt to predict information lost during compilation. We categorize ML-based approaches into two broad categories: *neural decompilation* and *targeted abstraction prediction*. First, we discuss the former, of which IDIOMS is a member, before turning to the latter.

ML-for-decompilation work has shown improvements over time, but the problem is far from solved. No existing technique in either category is accurate enough to reach widespread industrial adoption yet. Additionally, some older techniques are impacted by methodological issues resulting in data leakage, limiting the validity of the results.

2.3.1 Neural Decompilation

Neural decompilation approaches attempt to reconstruct the original source code for a particular binary. Some neural decompilation approaches take assembly as input; others use the output of a deterministic decompiler. This latter approach is sometimes called decompilation refinement [98].

Early work in neural decompilation focused on adopting more powerful architectures and tailoring them to the task of neural decompilation. Katz et al. [57] use a recurrent neural network. Coda [36] uses a multi-stage model with tree-based encoder and decoder to generate an approximate solution followed by an ensemble of other models to edit the first stage’s output. Cao et al. [17] use a graph neural network.

Since the transformer model has become dominant, work has shifted to scaling these models in size and leveraging them in novel ways. BtC [48] uses the powerful transformer architecture [102] in its encoder-decoder (sequence to sequence) configuration. SLaDe [5] is 50% larger than BtC [48] and uses numerous small adjustments. LLM4Decomp [98] introduce a family of large causal transformer models with sizes in the billions of parameters. LLM4Decomp models come in two flavors: one that takes assembly as input and one that takes deterministic decompilation from Ghidra as input. Nova [53] is designed to handle assembly using a custom hierarchical self-attention mechanism designed to handle the low information density and long sequence lengths of assembly code.

IDIOMS, described in Chapter 4, is also a neural decompilation model. Unlike all prior work in neural decompilation, IDIOMS produces the definitions for user-defined types used within the code it generates. This addresses a key gap in prior work, which is underspecified with respect to user-defined types. To help predict the UDT definitions, IDIOMS leverages interprocedural context, an approach not seen in prior neural decompilation work.

2.3.2 Targeted Abstraction Prediction

Other work has focused on targeting a particular issue with decompilation, or a fixed set of a few issues. For instance, one of the most obvious deficiencies with decompiled code is the lack of variable names; some techniques target only variable names. As we discuss in Chapter 3, there are numerous issues with decompiled code. Accordingly, there are techniques targeting numerous different issues. Following is a discussion of recent work in each of several popular target issues.

Variable names In DIRE [62], Lacomis et al. develop a novel technique to generate supervised training data for predicting variable names in decompiled code by compiling and decompiling binaries with debug information then stripping away debug information and decompiling again. They also contribute an ensemble neural model consisting of an LSTM lexical model and a GGNN graphical model. DIRECT [76] follows up this work with a more modern transformer [102] neural architecture to improve performance on the DIRE dataset. VarBERT [79] is a transformer-encoder variable name prediction model using a newer corpus, VarCorpus, and which predicts names from a fixed set. In contrast, GenNm [97] uses a generative decoder-only model to predict variable names. Unlike VarBERT, it can generate arbitrary names, and unlike all of the above work, uses information from a function’s calling context to inform variable name predictions.

Variable Types Several techniques have attempted to predict the types of variables present in an executable. Stateformer [80] uses an attention [8]-based architecture to predict types in de-

compiled binaries, pretraining their model on a generative state modeling task (predicting the complete set of execution states in assembly snippets) before training on a type prediction step. Their approach does not predict the composition of composite types like `structs`. Osprey [113] uses binary analysis to identify and eliminate candidate types, using probabilistic reasoning to resolve ambiguities. TyGr [114] uses a graph neural network predict variable types for C. Osprey and TyGr only predict the *composition* of composite or user-defined types, not the names of the types or their fields. Lehmann and Pradel [66] use an LSTM model to predict types in WebAssembly code in a custom, expressive type language. Of course, computer security researchers have also studied non-learning techniques for type recovery for decades [15, 65, 68, 93].

Function Names David et al. [29] introduce the Nero tool, which generates a call graph and predicts function names based on the structure of the call sites and the position of nodes in the call graph. Likewise, SymLM [54] uses program analysis techniques to analyze an executable, identifying the calling contexts of each function. This information, as well as information about the function itself, is fed into a heterogeneous collection of neural models to predict each functions’ name. Kim et al. [59] use a single transformer model for the same purpose.

Multitask Methods Some techniques provide the ability to perform multiple of the above tasks in a single model. DEBIN [43] is a tool that uses conditional random fields on a dependency graph between elements of a lifted IR representation of an executable to predict DWARF debug information, including variable names, variable types and function names in an executable. DIRTY [21] predicts variable names and types and is based on the transformer [102] architecture. Based on the intuition that variable names and types are not independent, it contains a custom interleaving variable name and type decoder. A key drawback of DIRTY, however, is that it predicts types out of a fixed set. ReSym [105] is also a name-and-type prediction method. It has two language-model-based decoders, one which generates the names and types for local variables, and the other which generates field access information, if needed. The results from each model are aggregated using a Prolog-based inference system. HexT5 [106] is general-purpose language model for security, evaluated on four different tasks: summarization, function name prediction, variable name prediction, and code similarity.

2.4 Program Equivalence

The proposed work verifies that a neural decompiler’s prediction preserves the semantics of the decompiled code. In other words, it is a program equivalence task. Program equivalence is undecidable; iteration (via loops or recursion) is the concept which makes it so. Approaches to handling loops can be grouped into four main categories: (1) assuming an iteration in one loop matches an iteration in another [10, 30, 35, 63, 75], (2) using bounding to limit the number of iterations to a user-specified maximum value [7, 63, 69, 115], (3) relying on an external source of additional information, like a human engineer, execution trace, or compiler (during optimization), to help infer loop-related properties [23, 56, 74], and (4) using heuristics designed to take advantage of certain properties of the input and output programs [23, 27, 41]. Early work is

primarily based on the syntactic form of the input programs [75, 109], while much more recent work has taken advantage of advances in SMT solvers to decide equivalence [7, 23, 27, 69].

One of the most well-studied applications of program equivalence is *translation validation* [82]: ensuring that the input and output of a program that changes the syntactic form of a program are semantically equivalent. The primary use case for translation validation is checking that compiler optimizations do not affect program semantics [75]. Alive2 [69] uses loop unrolling to handle loops, which is unsound (a validated translation may actually be not semantics-preserving), but can still detect many interesting mis-optimizations in LLVM. It is designed to ensure that no new undefined behavior is added during optimization. Some compiler validation techniques use information provided by the compiler to help perform optimization [56, 74, 75]. Neural decompilers are black-box machine learning models and thus cannot inform the equivalence-checking process.

Other equivalence techniques are more amenable to black-box code transformations. Churchill et al. [23] use test cases to create traces for the input programs, then compares them to suggest relationships between states in the input programs. JFTGs [27, 41] bundle control flow with nonbranching code and attempt to match branches in the input programs with each other with a heuristic guess-and-check strategy; Gupta et al. [42] use a similar technique guided by counterexamples. These approaches either use execution information which is difficult and possibly dangerous to obtain in neural decompilation scenarios or are very computationally expensive. Another approach is to summarize the entire program with constrained Horn clauses and using an appropriate solver to show equivalence [30, 35]. These approaches require the synchronization of one loop with another. A very different paradigm for translation validation is that of equality saturation [96, 99], which involves repeatedly applying transformations known to be semantics preserving. This approach is limited by the manually-specified transformation rules.

A related task is *semantic differencing*: finding a *functional* difference between two programs, such as a counterexample generated by an SMT solver. Differential symbolic execution [81] identifies textual or AST-based differences in code, symbolically executes them, and queries the SMT solver to check their equivalence. ArrDiff [7] builds on this work by extracting information from unchanged code blocks that may nonetheless be useful for showing the diffs equivalent. Symbolic-execution-based approaches are useful, but are unsound with respect to loops, are expensive to run, and must usually compile the input programs, which renders them useless for tasks like evaluating neural decompilers. In contrast, SymDiff [63] translates input functions into Boogie [9], a verification language, and creates logical formulas summarizing the effects of the functions, then uses an SMT solver to check their equivalence. This approach cannot fully handle loops; they must be unrolled to a specified depth or translated to tail-recursive functions, the latter of which are checked for equivalence separately.

Chapter 3

A Taxonomy of C Decompiler Fidelity Issues

To motivate and guide our efforts to improve decompiled code, we begin with a study of the problem itself: what makes decompiled code difficult to work with? Or, more concretely and less subjectively, how does decompiled code differ from the corresponding original code?

3.1 Introduction

Figure 3.1 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 3.1 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 3.2.1 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 [104]. 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 [17, 21, 36, 57, 62].

However, while variable names and types are certainly important for program comprehension, including in a reverse engineering context [13, 104, 110], 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.

¹Decompiled code is not always syntactically correct C 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 3.1: 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.

We argue that before designing more advanced solutions, we first need a deeper understanding of the problem. Consequently, in this chapter 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 angr⁵ [92] decompilers. Next, we use thematic analysis, a qualitative research method for systematically identifying, organizing, and offering insights into patterns of meaning (themes) across a dataset [11], 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.

²<https://hex-rays.com/decompiler/>

³<https://ghidra-sre.org/>

⁴<https://github.com/avast/retdec>

⁵<https://angr.io/>

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, in this chapter, 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, 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 four modern decompilers.
- An analysis describing classes of decompiler issues and suggestions for how to fix them.

3.2 Methodology

To build our taxonomy of fidelity issues in decompiled code, we used open-coding techniques [58] followed by thematic analysis [11] 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.”

3.2.1 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 3.2.

3.2.2 Coding Standards

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

A critical assumption made in Section 3.2.1 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 some challenges to

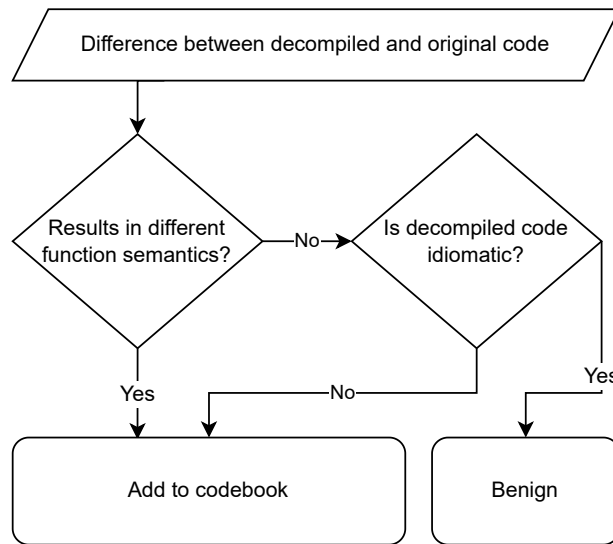


Figure 3.2: The process by which a difference between the decompiled code and corresponding original code was added to the codebook. Differences added to the codebook are considered meaningful defects, while others are considered benign differences in style. When in doubt, differences were added to the codebook.

Original	Decompiled
<pre> 1 void help_object(obj_template_t *obj, help_mode_t mode) 2 { 3 if (obj == 0) 4 { 5 set_error("help.c", 436, ERR_INTERNAL_PTR, 0); 6 return; 7 } 8 obj_dump_template(obj, mode, get_nestlevel(mode) 9 , 0); 10 }</pre>	<pre> 1 long long help_object(long long a1, unsigned int a2) 2 { 3 unsigned int v3; 4 if (!a1) 5 return set_error("help.c", 436LL, 2LL, 0LL); 6 7 v3 = get_nestlevel(a2); 8 return obj_dump_template(a1, a2, v3, 0LL); 9 }</pre>

Figure 3.3: 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.

doing so. Figure 3.3 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 3.3). 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 3.3’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 for this chapter.

3.2.2.a 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.

3.2.3 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 `Makefiles` and executing those to build executable binaries using a tool called GHCC [51]. 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 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.

3.2.4 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.

Table 3.1: 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

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 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 3.4.1.c and 3.4.2, Hex-Rays exhibited only one type of incorrect return behavior.)
5. **Generalization Across Coders** To ensure that our codebook was robust, we performed several rounds of inter-coder reliability testing [6]. 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 inter-rater 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 3.1. 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 [55], 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.

3.3 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 3.4 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. The full codebook is in the supplementary material.

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 [21, 43, 62, 76, 113].

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 3.4, 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 3.4 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 `structs` 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 3.4 illustrates an example where an extraneous variable, i.e., one that does not occur in the original

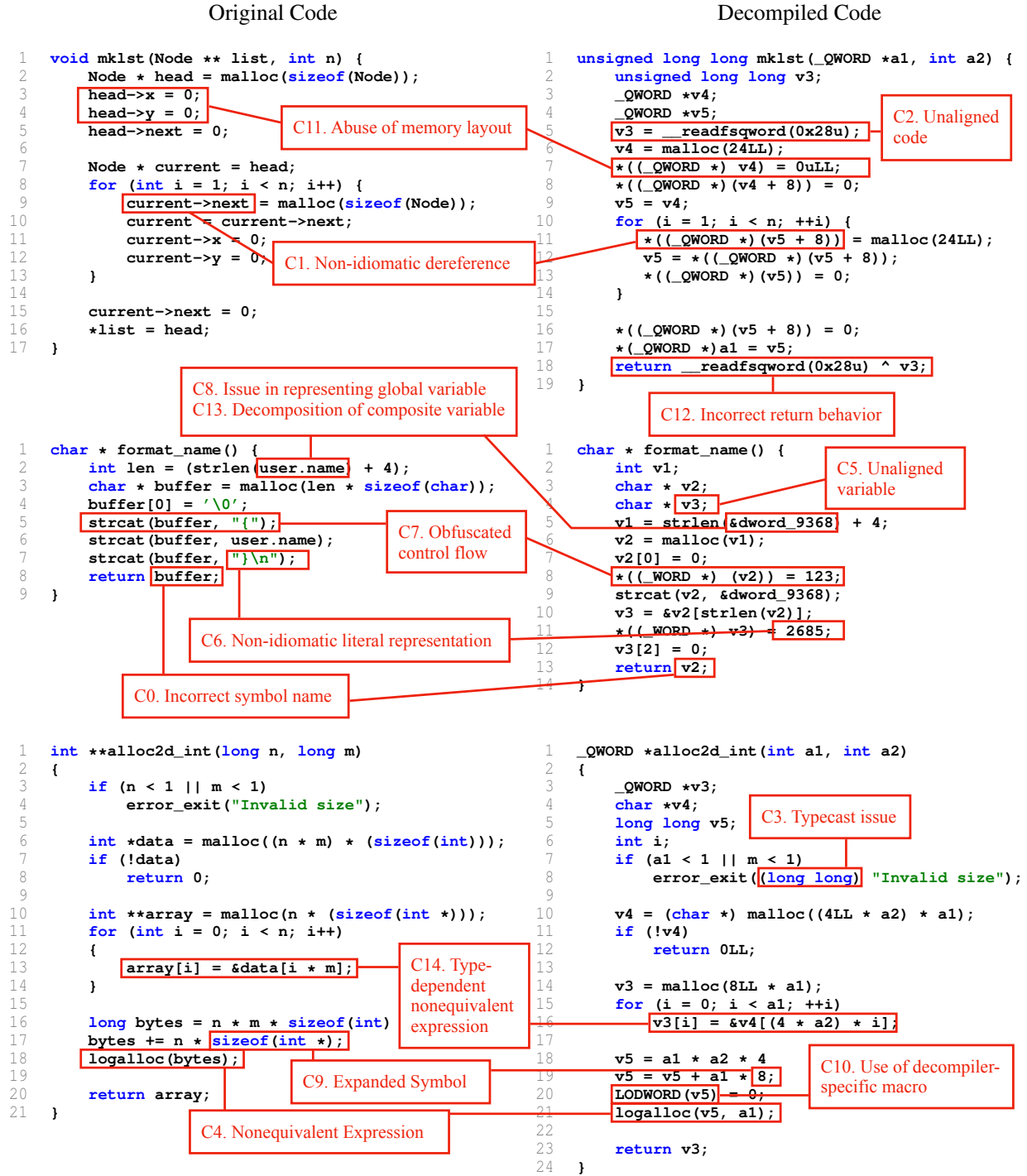


Figure 3.4: Examples of each of the top-level labels in our codebook. These examples have been artificially constructed by combining code fragments and common patterns from our sample in such a way that all fifteen top-level labels are represented at least once. For legibility, only one instance of each 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

```
1 while (pack->next_object != obj)
2 {
3     pack = pack->next_object;
4 }
```

Decompiled

```
1 for (i = a2;
2 a1 != (*((__QWORD *) (i + 64)));
3 i = (*((__QWORD *) (i + 64)));
```

Figure 3.5: 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.

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 3.4 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 3.4 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 3.4, 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 3.3 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 non-standard ways. For example, in Figure 3.4, 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 3.4, 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 3.5.

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 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 3.4 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 3.4 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 3.4 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 3.4 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 3.4 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 `structs` 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 3.4 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 3.4, 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.

3.4 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.

3.4.1 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 [17, 21, 36, 57, 62, 76]. 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 variable’s name can be changed without affecting the executable form of the program [62]. 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 [112]. In contrast, as a reverse engineer who participated in Votipka et al.’s [104] 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 [21]—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

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 3.6: 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.

relative frequencies may differ in other software.

3.4.1.a 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 3.6 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 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_strncpy(param_4, &DAT_00100cdf, param_5)` instead of `fz_strncpy(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.

3.4.1.b 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. [21] 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 `structs`. 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.

3.4.1.c 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 3.3. 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 [62], 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. [20] 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.

```

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 3.7: The decompiled function from Figure 3.3 with the heuristics in Section 3.4.1.c applied. Applying the heuristics increases similarity to the original.

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 3.7 shows the decompiled function from Figure 3.3 when both of these heuristics are applied. Applying the heuristics increases similarity to the original.

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 3.4.1.c, extra statements correlating with extraneous variables (C5.a) and extraneous

Table 3.2: 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

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 3.5. 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 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.

3.4.2 Comparison of Decompilers

To ensure the generality of our codebook, we randomly sampled 25 functions from the small function test set (Section 3.2.4) 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 shown in Table 3.2.

3.4.2.a 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.

3.4.2.b 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 3.4.1.c, 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).

3.4.2.c 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.

3.4.2.d Angr

Angr [92] 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

```
1 typedef struct struct_1 {  
2     char padding_0[8];  
3     unsigned long long field_8;  
4 } 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).

3.4.3 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 [55] and coded them as described in Section 3.2.4. 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.

3.5 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` 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.

3.6 Conclusion

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 [38, 39, 64] differ from those in our taxonomy.

Our analysis of our taxonomy yields several key takeaways. 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.

While the second insight applies to deterministic decompilers, we leverage the first and third to create IDIOMS, a novel neural decompilation technique, in the next chapter.

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>.

Chapter 4

Idioms: Neural Decompilation With Joint Code and Type Prediction

4.1 Introduction

Neural decompilation is the most general form of ML-based decompilation improvement, allowing arbitrary transformations and rewrites of the input to statistically recover any type of abstraction that is missing or distorted. As we find in Chapter 3, such ability to make broad transformations is necessary to be able to fix all known fidelity issues in decompiled code. In some cases, neural decompilers can greatly outperform traditional deterministic decompilers.

For example, Figure 4.1a shows a function that initializes a time structure. Figure 4.1b shows the same function, but after having been compiled and decompiled with the industry-standard Hex-Rays decompiler. Notice that the decompiled code is missing meaningful names, and the time structure is misrepresented as an array of integers. In contrast, Figure 4.1c shows the prediction of LLM4Decompile [98], a leading state-of-the-art neural decompiler. Although LLM4Decompile did not recover meaningful names in this instance, it did predict that the function’s last argument was likely a structure, and the decompiled code is easier to read.

Although it may be an improvement over deterministic decompilers, the output of neural decompilers is still far from ideal. A security practitioner can glance at the original source code (Figure 4.1a) and immediately know that the function is time-related by looking at the names (`real_time_2_tm`) or output type (`struct rtc_time`). In contrast, the neurally decompiled code (Figure 4.1c) contains no obvious clues that the function manipulates times. More problematically, studies have shown that reverse engineers often work across multiple levels of abstraction [104] such as assembly code and decompiled code. To map between abstraction levels, it is necessary to know the memory layout of the data structures used in decompiled code. Although we can tell from the figure that `struct_t` is a structure, a reverse engineer does not know the memory layout *because there is no type definition at all*. The lack of type definitions causes problems for code analysis as well. As is, the code in Figure 4.1c is not well-defined without a definition of `struct_t`; it cannot be analyzed by static analysis tools, and cannot even be compiled.

This limitation is significant. As we discuss in Section 4.2.2 (Table 4.1), user-defined types

```

1  struct rtc_time {
2      int tm_year;
3      int tm_mon;
4      int tm_mday;
5      int tm_hour;
6      int tm_min;
7      int tm_sec;
8  };
9
10 void real_time_2_tm(int year, int mon, int
    mday, int hour, int min, int sec,
    struct rtc_time *tm) {
11     tm->tm_year = year - (1900);
12     tm->tm_mon = mon;
13     tm->tm_mday = mday;
14     tm->tm_hour = hour;
15     tm->tm_min = min;
16     tm->tm_sec = sec;
17 }

```

(a) A function which initializes a structure describing a date and time.

```

1  void func0(int a, int b, int c, int d, int
    e, int f, struct_t *s) {
2      s->a = a - 1900;
3      s->b = b;
4      s->c = c;
5      s->d = d;
6      s->e = e;
7      s->f = f;
8  }

```

(c) Figure 4.1a, having been compiled, decompiled and run through LLM4Decompile-Ref [98]. While the predicted code represents the same sequence of operations as 4.1a, it is difficult to know if the prediction is correct without knowing the types of the fields used and how they’re arranged within the type `struct_t`. LLM4Decompile’s solution is *under-specified*; the definition of the type `struct_t` is necessary to fully define the semantics of the function.

```

1  _DWORD * func0(int a1, int a2, int a3, int
    a4, int a5, int a6, _DWORD *a7) {
2      _DWORD *result; // rax
3
4      a7[5] = a1 - 1900;
5      a7[4] = a2;
6      a7[3] = a3;
7      a7[2] = a4;
8      a7[1] = a5;
9      result = a7;
10     *a7 = a6;
11     return result;
12 }

```

(b) Figure 4.1a, which as been compiled, then decompiled. The decompiled version is missing details like names and types, which makes it harder to read.

```

1  struct date_t {
2      int year;
3      int month;
4      int day;
5      int hour;
6      int min;
7      int sec;
8  };
9
10 void set_date(int year, int month, int day
    , int hour, int min, int sec, struct
    date_t *date) {
11     date->year = year - 1900;
12     date->month = month;
13     date->day = day;
14     date->hour = hour;
15     date->min = min;
16     date->sec = sec;
17 }

```

(d) Figure 4.1a, having been compiled, decompiled, and run through the IDIOMS version of LLM4Decompile we introduce in this work using our training process. Unlike Figure 4.1c, the IDIOMS model jointly predicts the necessary type definitions along with the function.

Figure 4.1: A function with a user-defined type and different decompilations of it.

(UDTs), such as structs, are widespread in real code. However, existing neural decompilers are not designed to predict the definitions of UDTs. This problem is largely masked because the benchmarks that existing neural decompilers are evaluated on feature very few UDTs, if any at all.

In this chapter, we address these shortcomings by proposing a novel method for training neural decompilers that more effectively reconstructs user-defined type definitions alongside re-

constructed code. Figure 4.1d shows the output of our approach, which we call IDIOMS, since it recovers idiomatic code. Notice that IDIOMS predicts the type definition of the output structure, `struct date_t`, as well as the names of the fields within it.

There are two key insights behind our approach:

Insight 1: Code and types should be predicted jointly. The code and function definitions must be predicted *jointly*—in coordination with one another—so that the applications of the type in the code (e.g., the field names) are consistent with the definition. The alternative approach—predicting code and types independently—is problematic because an independent type recovery tool is unaware of the neural decompiler’s prediction, and so it may produce field names that do not match those in the prediction. For example, one might wonder if it is possible to use a neural decompiler like LLM4Decompile alongside a separate existing type recovery tool [21, 105, 114], but this will not help solve the issue in Figure 4.1c. Suppose a type prediction tool was used to generate the following type definition:

```
struct s {int u; int v; int w; int x; int y; int z;};
```

This does not help someone interpret Figure 4.1c because the field names do not match those in Figure 4.1c, despite having the correct memory layout. Because type definitions must be consistently applied throughout code, we argue that the code and types must be predicted jointly, in a coordinated fashion. Field names relate a type definition to its use and application in its associated code.

Insight 2: Scattered evidence for UDT reconstruction necessitates consideration of broad context. A key reason it is difficult to predict UDT definitions is the *scattered evidence* problem: in general, only a subset of a UDT’s fields are accessed within any given function (Section 3.4.1). This motivates us to consider the broader context of a function’s call graph to provide additional evidence for UDT reconstruction in model training and prediction. We train a series of models where we provide additional context in the form of functions that are close in the call graph, which we term *neighboring* functions. By nature, a function’s callees and callers process the input, output, and some internal values associated with a function, and may offer additional clues as to the types of those values. In turn, the types of values within those callees and callers may be informed by their neighbors.

Based on these insights, we introduce a new training strategy and associated family of IDIOMS models, that (a) jointly predict code and type definitions, enabling consistent type application, and (b) leverage neighboring functions in the call graph to provide necessary additional context for UDT reconstruction. Like LLM4Decompile-Ref [98], we use the output of deterministic decompilers as input to our model(s), allowing us to take advantage of the decades of work in deterministic decompilation and reduce the difficulty of the model’s task. Unlike LLM4Decompile, we train our IDIOMS family of models to predict the complete definitions of all UDTs used by a function along with the definitions of those functions. Decomplied code is the only input required to our trained models; unlike other existing work [5], for example, our approach does not require test cases, which are typically unavailable in reverse engineering scenarios.

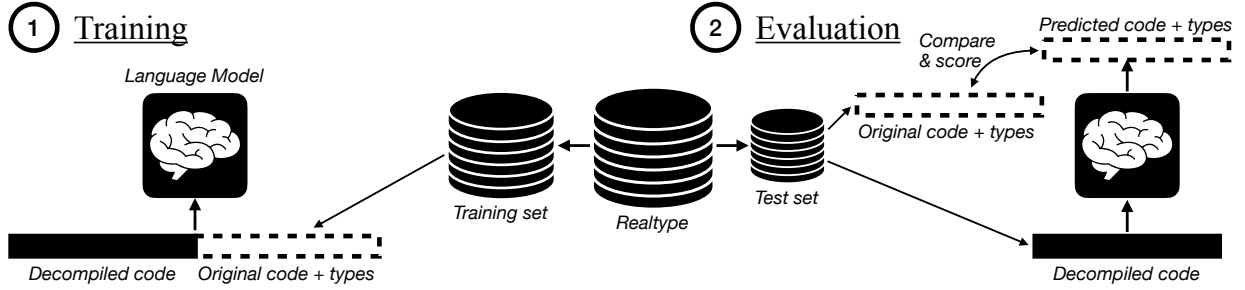


Figure 4.2: Our high-level approach. We use our dataset, REALTYPE, to finetune causal language models into IDIOMS models then evaluate them.

Our experiments show that neurally decompiling functions containing UDTs is more challenging than decompiling those without, even when controlling for function size. These findings generalize across a variety of model sizes and architectures. We show that our approach and family of models achieve state-of-the-art neural decompilation results (even on code without UDTs in the most challenging existing benchmark). We also demonstrate that neighboring context substantially improves UDT composition prediction—the types and order of all fields, recursively—by up to 63%, with larger models benefiting more.

In summary, in this chapter, we contribute

- A new dataset, REALTYPE, containing 154,301 training functions and 2,862 evaluation functions with real UDTs. We extract complete definitions of all UDTs for all functions in REALTYPE by parsing preprocessed original source code.
- A training procedure to jointly predict user-defined types and code in neural decompilation.
- A state-of-the-art family of neural decompilation models, which we call the IDIOMS family of models.
- Experimental results showing the difficulty of UDT type prediction, the ability of our neighboring-context training strategy to improve the prediction of UDTs and the generalizability of both across a wide variety of model architectures and sizes.

In the interest of open science, we release the code used to build the datasets, train and evaluate the models, as well as our dataset, REALTYPE, all IDIOMS models, and other models we train in our experiments for comparison.¹

4.2 Approach

Figure 4.2 overviews our approach. Neural decompilation generally entails *training* machine learning models to predict the original source code for some target function of interest from some deterministically derived representation of the binary (left-hand-side of Figure 4.2). Our IDIOMS models are trained to operate on the output of a deterministic decompiler, and to predict the original source code definition of a target function, along with a list of user-defined types in

¹Code can be found at <https://github.com/squaresLab/idioms>; models and datasets can be found at <https://zenodo.org/records/14797017>.

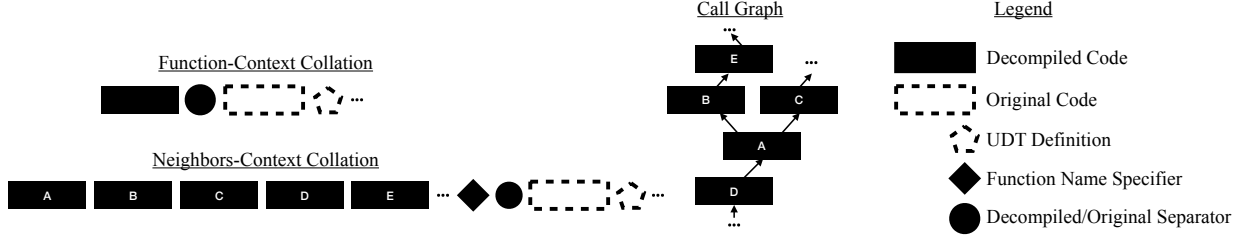


Figure 4.3: Organization of the sequences used to train the models. IDIOMS models use neighboring context, which places decompiled functions starting from a target function in the context window following a breadth-first search of the call graph. In the figure, A is the target function. Function context is used in some experiments to compare with IDIOMS.

that function. Section 4.2.1 details the modeling approach, which entails finetuning pretrained causal language models, using call graphs to select context to inform predictions.

Training models in this way requires suitable example input and output data. Existing datasets for neural decompilation have two key limitations: (1) a lack of variables with UDTs and their definitions, and (2) they are organized at the function level, meaning there is insufficient information available to build a call graph. We build a new dataset, REALTYPE, that includes these features, that we describe in Section 4.2.2.

Using this data, we train a family of causal language models to predict the original code associated definitions of user-defined types. Section 4.2.3 describes training details.

Once trained, models are used to predict the original code and UDT definitions for a given decompiled function (right-hand-side of Figure 4.2). For evaluation purposes, this must be performed on data not included in the training dataset. We discuss evaluation in more detail in Section 4.3.

4.2.1 Modeling

Model architecture and finetuning. State-of-the-art language models are neural networks, machine learning models that encode information in terms of real numbers called parameters. Models learn via training on typically large quantities of data; the goal of training is to learn parameter values that minimize the error entailed in the model’s prediction (using a process called backpropagation). To partially overcome the challenges of sufficient training data for a particular task, a model can be *pretrained* on data for a related task, and then *finetuned* on a much smaller quantity of data relevant to the actual target task. This has been widely shown to be more effective than training a model for the target task alone on smaller quantities of data. The approach we take for neural decompilation follows this paradigm by finetuning causal language models pretrained on code. Causal language models process token sequences. Figure 4.3 illustrates the overall organization of the sequence provided to our model.

In a supervised context like this, where the task is to learn to predict an output (original code + UDT definitions) from an input sequence (decompiled code), the input and output are concatenated into a single sequence, where the input precedes the output. We delimit the two with a *separator token* which indicates that the input is complete. At evaluation time, or during

use in practice, the input and the separator token are used to prompt the model to generate the output (the rest of the sequence).

Jointly predicting code and types On the “output” side of the context is the definition of the function and user defined types used within it from the original source code; in Figure 4.3, this is everything to the right of the ●. Importantly, code and type definitions are part of the same sequence. Causal language models use the entire preceding sequence to predict the next token in the sequence. This means the names of the fields as used in the prediction, as well as how they are used, are available when the model is generating the type definitions. This allows type definitions to be generated jointly with the decompiled code.

Neighboring context IDIOMS models use neighboring decompiled context to help address the scattered evidence problem for UDT composition. In Figure 4.3, the decompiled context is to the left of the ●. It is arranged such that the target decompiled code is placed first, and the callees and callers of the target function are placed after it. Then their callees and callers are placed if they haven’t been already added to the context, and so forth. This breadth-first order is not arbitrary. Intuitively, the model must relate the information it learns from the neighboring context to inform the prediction of variable types in the target function. That information can only be related if there’s an unbroken trace from the source of the information to the target function. Additionally, due to practical resource constraints, models can only handle sequences up to a certain length, meaning that choices must be made on what context to put within the available space. The BFS order means the context can be cut off arbitrarily on the right side to fit context-length constraints while maintaining the property of having an unbroken trace back to the target function. All of this context is followed by a special name-indicator token and tokens of the decompiled target function’s name in the decompiled code.

Figure 4.3 also shows function-level context, which is used by all existing prior work on neural decompilation [5, 17, 36, 48, 53, 57, 98], though sometimes due to architectural differences on some older models the decompiled code and original code are actually separate sequences. Function-level context consists of only the target decompiled function. We demonstrate the value of neighboring context experimentally (Section 4.4) by comparing with models built using only function-level context.

4.2.2 Dataset

Machine learning requires training data, and our model architecture and overall goals impose two requirements on a training dataset. First, it should contain code with user-defined types representative of those found in the real world. Second, we must be able to construct/use interprocedural call graphs of the compiled binaries. Existing neural decompilation datasets like EXEBENCH [4] and HUMANEVAL-DECOMPILE [98] do not meet these requirements; they provide only function context, and have fewer and simpler UDTs than real-world code, if any. We examine this in detail in Section 4.4.2. We therefore construct a novel dataset, REALTYPE, that meets these requirements.

Mining functions with UDTs. To build REALTYPE, we cloned and compiled majority-C-language repositories from GitHub using the GitHub Cloner and Compiler (GHCC) tool [51]. The tool executes standard build configuration scripts, executes standard build scripts like Makefiles, and extracts any resulting ELF-format binary files, then archives the repository if it is under 100 MB in size. Among the ELF files are object (.o) files. Including object files increases the amount of data available because not all projects build completely. However, it also means that for projects that do build completely, the same function is present at least twice: once in the object file, and once in the complete binary (executable or library). We filter out all object files for which the functions in that object file appear in another ELF-format binary from that repository. We adapt dataset preprocessing scripts from the DIRTY [21] replication package to interface with the Hex-Rays decompiler to decompile each binary. We filter out PLT stubs, then canonicalize the function names for functions in the binary to `funcX`, where X is an integer, a standard canonicalization scheme [53, 98, 106].

To collect the original code and the types used in it (our output data), we modified GHCC to run `gcc`’s preprocessor, inhibiting the generation of linemarkers (`gcc -E -P`), and collecting any resulting generated text files. To ensure that the original code and binary code represent exactly the same version, we use the repository archives created in the previous step. We parsed the preprocessed code, tracking `typedef` aliases and recording the definitions of UDTs. We store all types in a python object model representing the C type system built on top of DIRTY’s [21] type system, allowing fine-grained programmatic analysis of types. The types we collect reflect the expressive power of C types, including UDTs nested arbitrarily deep and typed function pointers.

Next, we extract and textually canonicalize each function. To do this, we traverse the function’s AST and record all type descriptors (e.g. at variable declarations, typecasts, and the function’s return type). We de-alias each name by searching the previously created `typedef` alias chains to locate the original form of each name, then replace the names in the function’s text with the standard name. For instance, `int32` and `_int32_t` each becomes simply `int`.² We do this to provide a standard form for each type. This makes results easier to evaluate and reduces the ambiguity in the conceptual mapping between the memory representation of each type and its syntactic form that the neural decompiler must learn. We store the canonicalized form of each function along with the type of each variable in the function.

With the decompiled and original code prepared, we matched preprocessed functions with decompiled functions and organize them by the binary in which they occur. We do this so as not to leak information during training; a reverse engineer will attempt to analyze one or more binaries at once, without access to the original source. Within each binary, we compute and store the call graph between functions in that binary.

Deduplication. *Data leakage* is a key risk in machine learning. Machine learning models tend to “memorize” examples in the training dataset; evaluating a model on examples in the training set leads to inflated, unrealistic results. Thus, an important dataset preparation step is

²The C standard specifies the minimum size of each type, but not the maximum, so technically `int` is ambiguous—hence the reason for the alternative integer type names in the first place. We performed all experiments on the same platform, x86_64 linux with `gcc`, so this was not an issue in our case. We chose C keywords for ease of use, but an alternative convention that more explicitly describes the size of each type would also work, so long as it is consistent. It is the myriad of equivalent but different naming conventions that is the key issue.

Table 4.1: Complexity in Evaluation Datasets

	HUMANEVAL-DECOMPILE [98]	EXEBENCH [4] (<i>test_real</i>)	REALTYPE (ours)
Lines of code (mean)	15.2	13.9	14.2
Variables with a UDT (%)	0	0.5	26.4
Functions with a UDT (%)	0	1.9	53.4
Recursive UDT field count (mean)	N/A	2.2	17.6
Type-tree complexity (mean)	1.4	1.5	16.2
UDT Type-tree complexity (mean)	N/A	4.2	57.3

In this context, a UDT is a user-defined type, specifically a `struct` or `union`. Our dataset is extracted from real code on GitHub, while HUMANEVAL-DECOMPILE is based on programming challenge problems. EXEBENCH is also extracted from real code on GitHub, but the dataset construction process biases it against UDTs. Each primitive type is a leaf node in a type-tree and each other modifier is an internal node. The complexity measure is the total number of nodes: `int` has complexity 1, `int *` has complexity 2, and `struct s{int x; int y;};` has complexity 3.

to check that training and evaluation sets are disjoint. As we discuss in Section 4.5 the risk of data leakage via pretraining is relatively low for neural decompilation. However, the risk of data leakage during finetuning on this constructed dataset remains, because duplicates on GitHub are common [95].

To mitigate this risk, we deduplicate the dataset aggressively, so that we can confidently split the dataset into training and testing splits that we expect to be disjoint. Our approach is two-fold:

- Minhashing [12]: This deduplication technique, originally developed for deduplicating webpages, clusters similar text files, and has recently become popular in deduplicating code datasets [37, 61] due to its robustness. Exact-match deduplication is insufficient because many copies of popular projects represent different past versions of that project or contain small modifications. We treat all C files in a repository as a “document” (the order is arbitrary due to shingling) and treat the resulting clusters of repositories as duplicates. We select the repository from each which produced the most data. Sometimes some duplicates will fail to build, perhaps due to modifications made to build scripts.
- By-project splitting: we ensure that all data from a given repository ends up in entirely either the train, validation, or test set and is not spread across those sets. This is because the model can memorize project-specific details and conventions, including UDTs, and apply them at test time, greatly inflating results. Indeed, Xiong et al. [106] find that not performing by-project splitting in variable name prediction for decompiled code inflates results by 200-300%.

Using both strategies substantially increases their efficacy. Using by-project splitting without minhash deduplication means that a duplicate of a project in the training set could end up in the test set. By-project splitting is required to ensure that project-specific details aren’t leaked across set boundaries even when duplicates are not present.

REALTYPE characteristics. After deduplication, our dataset contained 154,301 training functions, 540 validation functions, and 2,322 test functions. We refer to this dataset as the REALTYPE dataset. REALTYPE has substantially more user-defined types than existing benchmarks. Table 4.1 compares REALTYPE with two popular benchmarks used to evaluate the state-of-the-art models LLM4Decompile [98] and Nova+ [53]. They contain very few, if any, user-defined types, and those that are present are unrealistically simple.

4.2.3 Training Details

To show the generalizability of our training approach, we finetune IDIOMS models from five pretrained model architectures and sizes:

- CodeQwen2.5 [52], 0.5 billion parameter version
- LLM4Decompile [98], 1.3 billion parameter version
- CodeGemma [100], 2 billion parameter version
- CodeGemma [100], 7 billion parameter version
- CodeLlama [85], 7 billion parameter version.

We use the common convention in machine learning to attach `-xb` to the name of each model, where `x` is the number of parameters in that model, in billions, e.g. `codellama-7b`.

For models under 1 billion parameters in size (CodeQwen2.5-0.5b), we perform traditional finetuning. For models above 1 billion parameters in size, finetuning becomes computationally prohibitive; we therefore leverage recent research using adapters and quantization (QLoRA [32] adapters), allowing for a high-fidelity approximation of full finetuning at a fraction of the computational cost and VRAM consumption.

When training on the full compilable partition of the EXEBENCH dataset (for comparison with REALTYPE), we trained CodeQwen for 8 epochs, the 1-2 billion parameter models for two epochs, and the largest models for 1 epoch. We find that the larger models need less finetuning before they converge; they are inherently more capable. Simultaneously, larger models are more expensive to finetune. Uniquely among the models we use, LLM4Decompile has been pretrained on decompiled code, though it was trained on code decompiled with Ghidra rather than Hex-Rays. We train all models with a cosine learning rate scheduler [70] starting from a learning rate of 5×10^{-5} . We use a batch size of 64.

To mitigate overfitting concerns, on the 8-epoch experiments, we use standard machine learning procedure and evaluate after each epoch on the validation set. The best performing version of the model is used for the final evaluation on the test set.

For our function-context models, we configure the context window—the maximum amount of tokens the model will process at once—to be up to 2048 tokens, 1024 of which is reserved for the original code tokens and UDT definitions. We configure the context window size for IDIOMS models with neighboring context to be up to 4096 tokens, 3072 of which is the decompiled function and neighboring context and 1024 of which is reserved for the original code and UDTs.

4.3 Experimental Design

Our experiments are designed to answer three research questions:

RQ1: *How effectively do neural decompilation models, including IDIOMS, predict code and UDTs, and how does performance relate to model and dataset size?* We examine the overall performance of IDIOMS and neural decompilation models in general and how scaling up the process impacts that performance.

RQ2: *To what extent, if any, do UDTs increase the complexity of neural decompilation?* Existing work is evaluated on benchmarks that contain limited UDTs. But as we have discussed previously, predicting UDTs is valuable, and UDTs are commonplace in real code. Do UDTs impose additional challenge for neural decompilers, and if so, how much?

RQ3: *How does adding neighboring context affect the quality of neural decompilers?* The scattered evidence problem tells us that the evidence needed to understand UDTs may be spread throughout different functions of the program. Does providing neighboring context help the model predict UDTs more effectively, and if so, by how much?

4.3.1 Datasets and Baselines

The experimental configurations entail two datasets: REALTYPE (our new dataset), and EXEBENCH [4]. EXEBENCH is the most challenging existing benchmark used to evaluate existing work. It features code extracted from GitHub, along with definitions for external symbols associated with some functions. Most of the definitions are synthetic and automatically generated by the authors. They do not feature the full definitions of UDTs, however. Many have a single field named `dummy`, marking it as a placeholder value. A minority of the examples, about 15.4% of the overall dataset, have unit test cases that the authors automatically generated. The test and validation sets are selected out of those 15.4% of the functions for which tests could be generated (hence the bias illustrated in Table 4.1).

Our primary baseline is an unmodified existing state-of-the-art model, LLM4Decompile [98], which we evaluate on EXEBENCH [4]. Note that originally, LLM4Decompile was trained on code decompiled by Ghidra rather than Hex-Rays. For the purposes of baseline results, we re-decompile the EXEBENCH test set using Ghidra and use this as input to the LLM4Decompile model; evaluating it directly on Hex-Rays data would represent a covariate shift that would hurt LLM4Decompile’s performance. Later, we also finetune LLM4Decompile on Hex-Rays data as part of our main experiments.

We do not compare against two other pieces of notable recent work in neural decompilation: SLaDe [5] and Nova+ [53]. SLaDe achieves very strong results for a very small model size. However, by design, SLaDe uses the test cases bundled with EXEBENCH for each function during prediction to decide which is correct. Unfortunately, this is unrealistic in practice, because the source code was used to create the test cases, which of course defeats the purpose of a decompiler. Even with the source code available, the authors of EXEBENCH [4] were only able to generate tests for 15.4% of the dataset.

Nova+ [53] is a neural decompiler that uses assembly code as input rather than decompiled C. Nova expects the assembly to be highly processed and regularized before being fed into the model. Because the work is under review, their preprocessing script is unavailable. Attempting

to decompile without the preprocessing script introduces a covariate shift for the model, and it produced very poor results. We look forward to testing Nova+ when it is released.

General purpose LLM chatbots like ChatGPT have been shown to be poor neural decompilers [5, 53, 98], especially for their immense size, likely because they are trained on vastly more “normal” source code and little decompiled code.

4.3.2 Metrics

Ideally, a perfect neural decompilation exactly matches the original source code. In practice, we want a neural decompiler to (1) preserve the execution semantics, and (2) restructure the names, types, and code to be more readable and idiomatic, or written in a style that mimics a human developer’s. In comparison with deterministic decompilation, the former means that a good neural decompiler should be no worse than a deterministic one, and the latter means it must add value beyond what a deterministic decompiler can provide. We use a suite of metrics to measure a neural decompiler’s efficacy on both fronts, comparing to the original source code as the gold standard.

Semantic preservation. There is no easy way to check the semantic equivalence of the neural decompiler’s prediction because program equivalence is undecidable. Instead, we use two complementary proxies: one complete, but unsound; the other sound, but incomplete:

- **Unit test accuracy:** whether the neural decompiler’s prediction passes all available unit tests. EXEBENCH includes unit tests with each example in the test set, intended to evaluate semantic fidelity of decompiled code. Tests are complete but unsound, but still a useful (if optimistic) proxy for correctness. (*Metric name: passes exebench tests*)
- **Static, dependency-based equivalence.** To address the limitations of unit testing and assess correctness for REALTYPE (which lacks unit tests), we use static, dependency-based equivalence checking, comparing against the original source code as ground truth. We use features of the CODEALIGN analysis framework [34] to extract control and dataflow dependencies between operations and determine if the resulting dependency graphs are isomorphic. Intuitively, isomorphic dependency graphs mean that the programs execute the same sequence of operations in the same partial order, while ignoring variable names.³ Doing so is a sound approximation of equivalence [109], subject to two limitations: type compatibility and side effects. We also report the fraction of the neural decompiler’s predictions for which the prediction is isomorphic (matches the ground truth), and for which all variable types match. (*Metric names: dependency-based equivalence, dependency-based equivalence + typechecks.*)

Code improvements. It is also important to quantify how the neural decompiler has improved the code by adding abstractions like accurate variable names and types. A challenge in evaluating variable name accuracy is determining which variables in the prediction correspond to those in the original source. The prediction may break up expressions into smaller subexpressions,

³We say partial order because the order of some statements can be swapped without affecting semantics. For instance, `x++; y++;` and `y++; x++;` are functionally equivalent.

with results stored in intermediate variables, or vice versa. This means that, except for function parameters, it can be ambiguous how to programmatically map between variables in the original and the decompiled versions of a function to compare their names and types. To address this challenge, we identify the operations that map to one another in the previously-computed isomorphic maps computed for dependency-based equivalence. With this mapping, we compare variables’ names and types. (*Metric names: variable name accuracy, variable type accuracy*)

If code has the same dependency structure as the original, that means it represents the algorithm expressed in the binary in terms of the same operations in the same partial order. In other words, the structure of the predicted code mimics that of the original code. Dependency-based equivalence thus also serves as a measure of how often the neural decompiler recovers the structure of the original code.

UDT accuracy. One of the key challenges with decompiling real C code is reconstructing user-defined types. One novelty of the IDIOMS approach is that it predicts UDTs alongside the code, and thus we measure accuracy on UDTs separately. UDTs are often named, as are their fields. Perfectly predicting the name of a UDT and each of its constituent fields is a challenging task. We therefore also look at partial metrics that decompose accuracy on UDT predictions. Namely, knowing type composition, and accurately predicting it — that is, the type of fields it has and the order in which those fields occur — can still be helpful to a reverse engineer. We therefore report the fraction of UDTs for which the composition matches (which ignores field names). When computing our sound equivalence metric “dependency-based equivalence + typechecks” we use UDT composition to determine if all UDT variables typecheck rather than requiring that UDT names match as well. (*Metric names: variable UDT names + composition accuracy, variable UDT composition accuracy*)

In summary, unit test accuracy is an upper bound on the rate at which neural decompilers preserve semantics, while dependency-based equivalence + typechecking forms an approximate lower bound. Variable name accuracy, the three type accuracy measures (variable type accuracy, variable UDT names + composition accuracy, variable UDT composition accuracy), and the fraction of the decompiled examples that have the same dependency structure as the original, quantify the improvements that neural decompilers make to the code.

4.3.3 Experimental Methodology

To answer our research questions, we train and evaluate each of the five base models described in Section 4.2.3 under four conditions. The primary evaluation of our proposed IDIOMS approach is **neighbors-realtypes (idioms)**: Models are trained and evaluated on the REALTYPE dataset, using neighboring functions identified by call graph traversal as context.

The other three conditions are designed to provide causal comparisons by differing one factor at a time, enabling assessment of the role of dataset composition and size, and IDIOMS design features:

- **exebench**: Training and evaluating on EXEBENCH
- **parity-exebench**: Training and evaluating on a subsample of EXEBENCH that is the same size as REALTYPE.

- **functions-realtype**: Training and evaluating on REALTYPE, but using only the decompiled function as context, as existing work does.
- and finally, **neighbors-realtype** represents the full IDIOMS setup.

The first set of conditions mirrors using existing state-of-the-art training practices on the most challenging existing benchmark used in neural decompilation research, EXEBENCH [4]. Experimental conditions **exebench** and **parity-exebench** differ in dataset size; **parity-exebench** and **functions-realtype**, the dataset used; and **functions-realtype** and **neighbors-realtype**, the type of context provided. We train/evaluate all models under all conditions to assess generalizability to model sizes. Notably, only **neighbors-realtype** uses neighbors-context; the other three experimental settings all use function-only context.

For the configurations that use EXEBENCH dataset, we used the compilable partition of EXEBENCH for which deterministic decompilation succeeded. For consistency, we process the dataset in the same way as we did ours in Section 4.2.2, with a few exceptions dictated by the information available. In particular, we compiled each function individually rather than each project/repository together, since the dataset is not organized by project and there is no project-specific build information. We also made use of the function-specific dependencies (`typedefs`, function declarations, and type definitions) provided in the dataset rather than running the pre-processor on an entire project at once.

Thus, for the **exebench** setting, we trained on the full compilable train set for which deterministic decompilation succeeded, which consists of 2,383,839 functions. EXEBENCH is substantially larger than REALTYPE, which consists of 154,301 functions, and machine learning model performance generally increases with training set size, so to control for this, we randomly subsampled EXEBENCH to create a smaller training set of the same size. Then, we trained each model on this subset for the **parity-exebench** setting.

For **exebench** and **parity-exebench**, we evaluate only on the “real” subpartition of EXEBENCH’s test set. As described in Section 4.3.1, the “synth” partition contains unrealistic synthetic dependencies, including user-defined types. Further, the authors of SLaDe find that the “synth” subpartition is substantially easier to predict than the “real”, with all techniques scoring at least 20% higher (in absolute terms) on “synth” alone as compared to “synth” and “real” together. (Real alone is not evaluated.) We also exclude from the EXEBENCH test case evaluations any examples where the provided oracle solution and dependencies don’t compile, or fail at least one test. This makes up about 16% of the overall `test_real` partition. The authors of SLaDe [5] find that about 44% of oracle solutions across both real and synth partitions fail.

To evaluate the role of dataset type and quality on UDT and neural decompilation effectiveness, we compare models trained only on REALTYPE, but without additional context. This is the **functions-realtype** experiment, which uses only the decompiled form of a function to predict the original form of a function and its user-defined types. Comparing this setting to the **neighbors-realtype** experiment allows an assessment of the value of neighboring context in neural decompilation.

4.3.4 Dataset and Prediction Processing

We perform a small amount of necessary filtering on the evaluation datasets. DIRTY’s [21] decompilation scripts fail on about 16% of the EXEBENCH `test_real` set; we are forced to filter these out because otherwise we have no input to provide to the model. Following LLM4Decompile [98], we do not evaluate on any examples for which the decompiled function does not fit within the model’s context window. This filters out about 3% of each test set. The “neighbors” experiments have expanded context windows; for a fair evaluation, we ensure that they are evaluated on the same subset of the test set as the corresponding “functions” model.

In our experiments, all models are trained to predict the decompiled code first, followed by any user-defined types used in the function or in other user-defined types. Especially with the **realtype** experiments, the model’s predictions often include degenerate text [46], usually type definitions following some kind of repeating pattern. Degenerate text is a common issue in even large and powerful language models, and its causes are not well understood. We find that degenerate text can occur even when the prediction of the function’s text is correct, and also even when the definitions of the UDTs of the functions are present and correct. (The correct UDTs, without observed exception, immediately follow the predicted function definition; degenerate ones follow the correct ones.) We hypothesize that the degenerate text is at least partially a result of the extreme difficulty of the task: in general, as we discuss in Section 4.1, it is difficult if not impossible to fully predict a user-defined type from a single function that does not use all of its fields. Because the decompiled code, especially in the **realtype-functions** experiments, is *not* predictive of all UDTs, the model learns simply to generate some arbitrary UDT definitions after generating the function text and any function definitions it can figure out. The fact that the effect is less pronounced in the **exebench** experiments supports this view.

Fortunately, the degenerate text affects the evaluation very little. Type accuracy is computed for each variable; if no variable uses a degenerate UDT, it is simply ignored.

Table 4.2: LLM4Decompile [98] Baseline

	LLM4Decompile
dependency-based equivalence	20.85
dependency-based equivalence + typechecks	14.15
variable name accuracy	13.44
variable type accuracy	43.39
variable UDT names + composition acc	0.00
variable UDT composition accuracy	0.00
passes EXEBENCH tests	39.23

All values are percentages. UDT accuracy is 0% by construction because LLM4Decompile does not generate UDTs.

4.4 Results

To calibrate expectations about model performance, we provide baseline results of LLM4Decompile [98] on EXEBENCH in Table 4.2. UDT accuracy is 0% because LLM4Decompile does not natively generate UDT definitions. The authors of LLM4Decompile only evaluate their assembly-based models, LLM4Decompile-End on EXEBENCH, not their decompilation-based models, LLM4Decompile-Ref. They report a unit test passing rate of 17.86%, but it’s important to note that LLM4Decompile-Ref is more powerful, and there are several experimental conditions that differ, so this number is not directly comparable with the last row of Table 4.2. This is why we provide a baseline.

The results of our four main experiments—**exebench**, **parity-exebench**, **functions-realtypes** and **neighbors-realtypes**—are shown in Figure 4.4. Each column represents an experiment, and the columns are arranged such that each experiment differs by exactly one experimental setting from the one next to it. Between the first and second it is dataset size (2,383,839 vs. 154,301 training functions), between the second and third it is the dataset type (EXEBENCH vs. REALTYPE), and the third and fourth differ in the amount of context provided (decompiled function only vs. decompiled function and neighboring functions). The last column thus corresponds to the primary experiment, **neighbors-realtypes**, which uses the full IDIOMS approach. Note that there is high variance in UDT metrics for the two EXEBENCH-based experiments because there very few UDTs in the `test_real` subset of EXEBENCH.

4.4.1 RQ1: Model Performance and Trends

In general, in machine learning, model performance scales with dataset size and parameter counts, but performance gains are usually logarithmic in training set size; doubling either does not lead to a doubling in scores. We see this as well across all of our data. For instance, CodeQwen2.5-0.5b, our smallest model, produces dependency-equivalent code 30.65% on EXEBENCH. Meanwhile, CodeGemma-7b scores 33.70, a percent increase of about 10% despite being 14 times larger. We also see this in terms of scaling the dataset. Columns 1 and 2 illustrate this. Column 1 represents a size increase of over 15 times but the gains in the same metric are only 6-11%. The gains on both counts are relatively modest, which may be attributable in the latter case to the fact that the models are pretrained on C code, which means they possess some inherent reasoning ability about C (or, more precisely, they model a good distribution of C code).

IDIOMS (column 4) follows a similar trend albeit on a more difficult dataset (see RQ2). CodeQwen2.5’s dependency-based equivalence score is 15.55%, though this drops to 7.17% when type correctness is also factored in. Meanwhile, the two 7-billion parameter models, CodeGemma-7b and CodeLlama-7b score only 18.25, 8.31 and 18.40, 7.99, respectively. However, the scores UDT composition matches increase much more rapidly with model size. (We discuss this further in RQ3.) The best IDIOMS model recovers UDT compositions over 15.06% of the time. Considering the mean size of UDTs (Table 4.1), and the fact that all UDTs must recursively have identical compositions, this is an impressive feat.

(a) CodeQwen2.5-0.5b				
	exebench	parity-exebench	functions-realttype	neighbors-realttype (IDIOMS)
dependency-based equivalence	30.62	25.96	16.21	15.55
dependency-based equiv+typechecks	20.37	16.96	7.11	7.17
variable name accuracy	18.76	15.23	13.85	13.63
variable type accuracy	54.42	45.64	32.94	30.40
variable UDT names+composition acc	11.11	6.90	3.16	3.69
variable UDT composition accuracy	37.04	10.34	6.03	6.36
passes EXEBENCH tests	44.43	32.09	–	–

(b) LLM4Decompile-1.3b-v2				
	exebench	parity-exebench	functions-realttype	neighbors-realttype (IDIOMS)
dependency-based equivalence	31.87	29.08	17.95	17.28
dependency-based equiv+typechecks	23.29	19.79	8.45	8.30
variable name accuracy	19.16	17.22	16.14	16.88
variable type accuracy	54.90	49.59	36.08	35.34
variable UDT names+composition acc	13.79	12.00	3.78	4.60
variable UDT composition accuracy	41.38	28.00	9.96	11.09
passes EXEBENCH tests	49.11	41.33	–	–

(c) Codegemma-2b				
	exebench	parity-exebench	functions-realttype	neighbors-realttype (IDIOMS)
dependency-based equivalence	30.89	29.00	16.97	18.12
dependency-based equiv+typechecks	21.54	21.09	8.49	8.61
variable name accuracy	19.45	16.95	15.73	17.56
variable type accuracy	55.60	50.62	36.08	35.42
variable UDT names+composition acc	24.14	24.14	3.77	4.86
variable UDT composition accuracy	55.17	41.38	7.40	11.53
passes EXEBENCH tests	48.32	41.29	–	–

(d) Codegemma-7b				
	exebench	parity-exebench	functions-realttype	neighbors-realttype (IDIOMS)
dependency-based equivalence	33.70	31.35	18.06	18.25
dependency-based equiv+typechecks	23.93	22.09	7.09	8.31
variable name accuracy	20.61	18.20	16.28	19.09
variable type accuracy	58.18	53.19	36.36	37.92
variable UDT names+composition acc	20.69	17.24	4.04	5.72
variable UDT composition accuracy	34.48	44.83	9.23	15.06
passes EXEBENCH tests	54.43	47.42	–	–

(e) Codellama-7b				
	exebench	parity-exebench	functions-realttype	neighbors-realttype (IDIOMS)
dependency-based equivalence	32.72	29.15	17.76	18.40
dependency-based equiv+typechecks	22.22	19.99	9.06	7.99
variable name accuracy	19.31	18.04	17.20	19.26
variable type accuracy	55.81	53.86	36.45	37.73
variable UDT names+composition acc	20.69	25.93	4.07	5.52
variable UDT composition accuracy	41.38	48.15	10.03	14.10
passes EXEBENCH tests	48.35	42.01	–	–

Figure 4.4: Main results. All values are percentages; higher is better. Adjacent columns differ in one experimental condition.

Answer to **RQ1**: Larger models and training sets increase performance modestly across all model types, including IDIOMS. The best IDIOMS models are able to achieve dependency-based equivalence scores of over 18% and UDT composition scores of over 15% on real code and types.

4.4.2 RQ2: The Challenge of Real-World Code with UDTs

The second and third columns compare datasets of the same size and with similarly sized functions (Table 4.1) but the third column represents our dataset, `REALTYPE`, which has far more UDTs. This change causes a substantial decrease in performance on all metrics except variable name accuracy. For dependency-based equivalence (row 1 on each of Tables 4.4a-4.4e), the drop is 38-42%, relative to performance on `EXEBENCH`. The drop is even steeper when type correctness is factored in (row 2): around 55%-68%. These data highlight the challenge that UDTs provide for real code.

The absence of UDTs in existing benchmarks masks the need for predicting UDT definitions alongside the code. `REALTYPE` helps close this gap.

Answer to **RQ2**: Code with UDTs is substantially more challenging to neurally decompile than code without UDTs, with dependency based-equivalence + type equivalence dropping by 55%-68% following the introduction of data with real UDTs.

4.4.3 RQ3: The Role of Neighboring Context

Providing additional context in the form of neighboring functions' code can lead to an increase in scores, especially with larger models. Interestingly, correctness is largely unchanged, though there is typically a slight increase. On the other hand, there is typically notable increase in UDT accuracy, especially in terms of composition where type and field names are ignored. Larger models are better able to take advantage of the additional context: UDT composition accuracy increases 63% for `codegemma-7b`, 41% for `codellama-7b`, and 56% for `codegemma-2b`, with small increases in most or all other metrics. Smaller models struggle to handle the additional context, at least without tradeoffs. `LLM4Decompile` gains 11% UDT composition accuracy, but suffers small losses in dependency-based equivalence and type accuracy. The smallest, `QwenCoder2.5-5b` gains only 5% but likewise suffers small losses in structural equivalence, type accuracy and additionally the combination of the two: overall typechecked dependency-based equivalence.

Answer to **RQ3**: Neighboring context help increase UDT accuracy with little to no downside in terms of other metrics, though larger models benefit more.

4.5 Limitations and Threats to Validity

A key threat to validity with most work involving large language models is data leakage through pretraining. The training sets for most LLMs are not public, but they usually consists of large

quantities of text scraped from the internet, including code from GitHub; our dataset is also derived from GitHub. Despite this, we think the risk of data leakage is small. Relatively little decompiled code is found on GitHub or on the internet in general; models will likely be unfamiliar with it. Further, the key function that a neural decompiler must learn is the *mapping* between decompiled and original code; there is likely even less of this data used in pretraining.

In this work, we only consider unoptimized (−OO) code. Several existing pieces of work consider different levels of optimization in neural decompilation or related tasks [5, 21, 53, 79, 98]. They generally find that optimizations cause a small-to-moderate decrease in efficacy. Considering optimizations would have lead to a combinatorial explosion in the number of models to train and evaluate (4 experiments $\times$ 5 architectures/sizes $\times$ 4 optimization levels), exceeding our computation budget. Rather than reconfirm a generally well-understood phenomenon, we opted to show the generalizability of our results across models of different architectures and sizes. However, it is possible that optimizations make recovering UDTs specifically particularly difficult. We leave this to future work.

Some targets of decompilation, especially malware, are often *obfuscated*: that is, transformed in a way that makes them more difficult to understand, usually by making them more convoluted. We view deobfuscation as an orthogonal problem as there are a variety of deobfuscation techniques [26, 28, 33, 67, 101, 111] that can be applied to de-obfuscate the code before feeding it to the neural decompiler. However, in cases where these fail or a novel obfuscation is encountered it may be necessary to input the obfuscated code to the neural decompiler directly. We leave studying the impacts of this to future work.

Finally, our dataset is necessarily biased towards open source projects on GitHub that built. It is possible that some decompilation targets, especially malware, may have systematic differences from our data and thus affect performance.

4.6 Conclusion

In this chapter, we introduce the IDIOMS family of neural decompilers, which perform joint code and type recovery. We build a dataset, REALTYPE, which contains 157,163 total functions and the definitions of their user-defined types. We train models on EXEBENCH and REALTYPE, and show that REALTYPE is a more difficult dataset despite their functions being of a similar size. Additionally, we show that additional neighboring context helps address the scattered evidence problem and thus improves performance on UDTs.

IDIOMS achieves state-of-the-art results, but neural decompilation is an extremely difficult problem. Of particular concern is semantic correctness. Even on EXEBENCH, using the most optimistic correctness metric, passing all EXEBENCH tests, the best model achieves only 54.43% accuracy. In other words, the model is wrong about half the time, making it much difficult to trust. To address this, in the next chapter, we propose a technique to check whether a neural decompiler has failed to preserve semantics by comparing the deterministically decompiled code with the neurally decompiled code.

Chapter 5

Robust Evaluation and Validation of Neurally Decompiled Code

5.1 Introduction

Neural decompilers are powerful tools that can help address the many fidelity issues with decompiled code, which are a key barrier to reverse engineers’ productivity. In other words, neural decompilers can add substantial value beyond what traditional deterministic decompilers can provide. Yet at the same time, neural decompilers are not strictly better than deterministic decompilers. This is because neural decompilers are probabilistic and may sometimes produce code that does not faithfully represent the executable. Here, we are especially concerned about semantic misrepresentations.

An analysis of the results in Table 4.4 reveals several limitations with neural decompilation in practice. First, semantic misrepresentations in neural decompilation are fairly common. Even on the easiest dataset in the evaluation, EXEBENCH, on the most permissive metric, unit tests, the best-performing model scores only 54.43% accuracy, meaning it contains semantic mispredictions about half the time. The presence of mispredictions may make it difficult for reverse engineers to *trust* the tool output, which in turn may make them less likely to actually use the tool in practice. In contrast, while deterministic decompilers can also incorrectly represent the semantics of the binary [18, 103, 115], they tend to be wrong in specific and well-understood ways [104], such as by including a return statement where none should exist. However, because neural decompilers probabilistically rewrite the code, they lose this reliability.

In recent years, large-scale commercial LLMs have become widely prevalent and have anecdotally been reported to be increasingly proficient at reverse-engineering-related subtasks, including neural decompilation. It is possible that frontier models have become sufficiently adept at neural decompilation that rates of semantic nonequivalence are negligible, so reverse engineers *can* trust them. However, it is difficult to determine whether or not this is the case using existing techniques: in other words, existing *evaluation* techniques are insufficient. In the preceding chapter, we use two primary complementary metrics: codealign [34], which is theoretically sound but incomplete, and unit tests, which are complete but unsound. In other words, if codealign returns

that the techniques are equivalent, then they are indeed equivalent,¹ while if it returns nonequivalent the code may or may not be. Likewise, if unit tests determine that the prediction is not equivalent, it is not, but if all tests pass, the prediction may or may not be equivalent. We note that the gap between codealign’s dependency-based equivalence scores and unit test accuracy is quite large, around 40% (20% in absolute terms), indicating insufficient precision to conclude much about model’s trustworthiness. A manual analysis indicates that unit tests are much closer to the “true” equivalence rate (though it is still an overapproximation) than dependency-based equivalence, but unit tests are unsound, and, as we discuss further below, it is extremely difficult to generate meaningful unit tests in practice for real-world functions.

Thus, it is evident that existing evaluation techniques are insufficient for neural decompilation. Further, as we show in Section 5.4, while modern frontier LLMs perform well, they still may make mistakes on even unoptimized code—neural decompilation is an extremely difficult problem. To increase the trustworthiness and reduce the risk of using neurally decompiled code, we propose performing *validation*: comparing the neurally decompiled code with deterministically decompiled code. If the two are equivalent, then the prediction is no worse than the deterministically decompiled code, which reverse engineers already find useful.

These two tasks are highly related. Both are code equivalence problems comparing two code representations that occur in the context of neural decompilation. In evaluation, the neurally decompiled code is compared with the reference original code. In validation, the neurally decompiled code is compared with the deterministically decompiled code. Figure 5.1 illustrates these two tasks in relation to the neural decompilation process as a whole.

In this chapter, we introduce a new technique, which we name FAULTLESS, for the evaluation and validation of neurally decompiled code. The nature of these tasks offers both challenges and opportunities.

Perhaps the most far-reaching challenge—that which has the greatest impact on FAULTLESS’ design, and which also makes most alternative prior work unsuitable for the tasks of validating and evaluating neural decompilers—is the context-independent nature of the problem. By this, I mean that the functions under analysis are arbitrary functions from inside the program and that most dependency information is not available. This means, for instance, that when attempting to show these functions equivalent, we don’t have access to the calling context, the definitions or declarations of callees, nor the declarations of global variables. This constraint is realistic in the context of neural decompilation. Reverse engineers will often do a high-level sub-component scanning step when beginning to work on a binary [104]; they may choose any

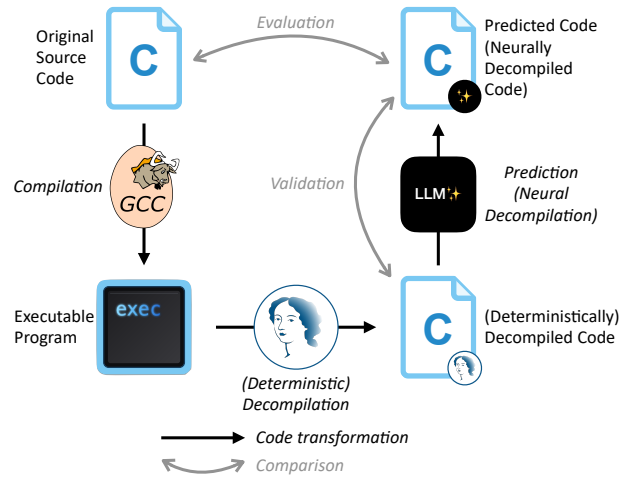


Figure 5.1: The four code representations in the neural decompilation problem, the relationships between them, and the two comparison tasks that FAULTLESS performs.

¹That is, subject to some limitations around side effects, which codealign does not model.

part of the binary that looks most promising to being their investigation. Further, binaries may be quite large and it may not be cost effective, practical or even possible with current techniques to neurally decompile the entire binary, especially in a way that renames symbols consistently across the entire neurally decompiled binary so that these dependencies can be reliably found.

A key implication of this challenge is that the code can't be compiled by typical C compilers which require full definitions of all dependencies. This aspect of the problem renders many existing program equivalence checking techniques unusable for this problem. This includes unit tests, because proper compilation is a prerequisite to execution.

One dependency we can rely on is the definitions of structs and unions: in the terminology of Chapter 4, user-defined types. As we show in the prior chapter, it may be feasible to obtain useful definitions of UDTs from a partial snapshot of the program and its call graph.

The validation task also introduces some additional challenges. As discussed in Chapter 3, deterministically decompiled code may be semantically nonequivalent to the original, though usually in ways that are well-understood. For instance, a decompiled function may return a value where the original does not because the compiler uses the return register as a general-purpose scratch register and the decompiler conservatively assumes that the temporary value within should be returned. Alternatively, a compiler may widen a 32-bit integer variable to a 64-bit one, causing a deterministic decompiler to assume that variable is the larger type. Neural decompilers, on the other hand, are trained to predict the original code. Accordingly, they may not—and are indeed trained to not—make the same conservative assumptions as a deterministic decompiler. Thus, there may be subtle semantic differences between a deterministic decompilation and a neural decompilation, even if the prediction of the neural decompiler is ideal: that is, identical to the original code. Because of this, a naïve application of existing program equivalence techniques to this problem would result in spurious false negatives where the deterministic and neural decompilers' predictions are “essentially” the same are flagged as nonequivalent. Effectively, to be useful, a carefully-calibrated degree of flexibility is required for equivalence checking between deterministically and neurally decompiled code. To this end, wherever possible, FAULTLESS ignores compiler-specific details and instead tries to capture the *algorithm* that the functions under comparison implement.

It is this property of the problem—that the functions under analysis are by construction implementations of the same algorithm—that offers a key opportunity that help inform one of the main design decisions of FAULTLESS. Iteration is the property of programs that make program equivalence especially difficult. At a high level, two implementations of the same algorithm should have the same *loop arrangement*. That is, for each loop in the original source, there is a corresponding loop in the decompiled code. Following prior work [34, 75], we say that two loops are equivalent iff (1) the state of memory at the start of the loops is the the same (2) if the i th loop iterations modify memory in the same way and incur the same side effects.

For the validation task, this may not always hold due to optimizations; some optimizations do restructure loops. (This is not a problem for the evaluation task because neural decompilers are trained to undo optimizations). In this context, if using an LLM, it is possible to simply prompt the model to preserve the loop structure of the deterministically decompiled code while otherwise improving the code as much as possible. We explore this possibility in RQ6 (Section 5.4.2.c).

Our experiments show that FAULTLESS is substantially more capable of performing evaluation and validation of neural decompilers than prior techniques. We also show that FAULTLESS

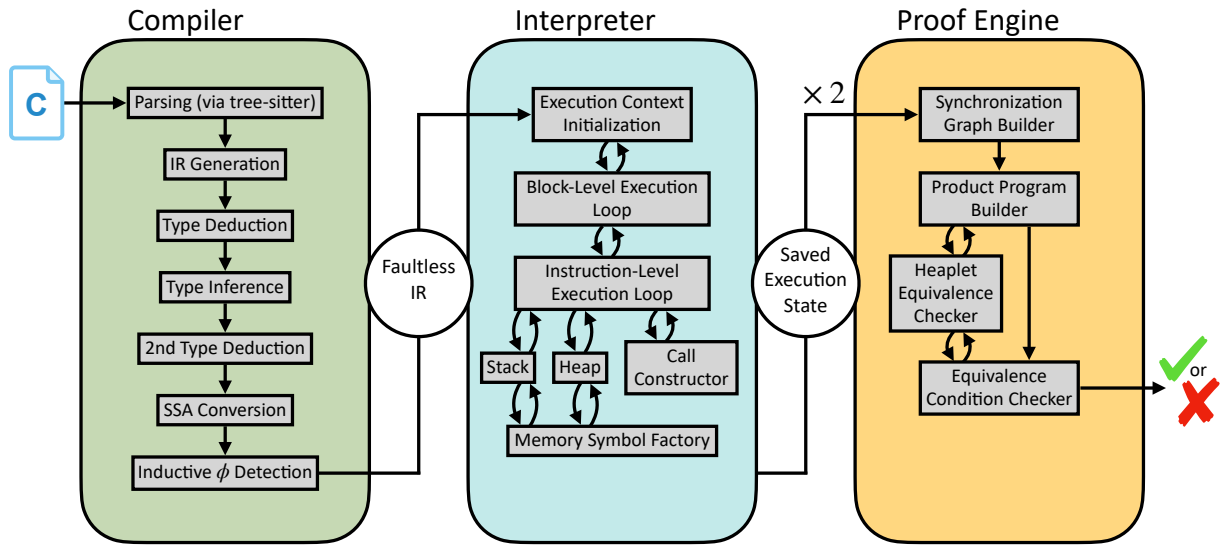


Figure 5.2: An overview of FAULTLESS. It takes as input two C functions and outputs a proof result (equivalent or nonequivalent with a diagnostic reason). It contains three main stages: a compiler, which converts raw C code into a custom intermediate representation (IR), an interpreter, which symbolically executes a function in this IR and records information about the execution, and a proof engine, which uses this information to construct a proof to, if possible, show the programs are equivalent. The first two stages are run twice, once for each of the two functions being compared.

is very efficient, performing 90.1% of equivalence checks on functions in REALTYPE in under one second and 96.7% in under two seconds.

In short, we contribute:

- FAULTLESS, an equivalence-checking technique for evaluating and validating neural decompilers.
- A python implementation of FAULTLESS totaling more than 11,500 lines of code. FAULTLESS has only two dependencies: `tree_sitter`, a parsing library, and `z3`, an SMT solver.
- A comprehensive evaluation measuring completeness and runtime performance, providing empirical evidence for soundness, and showing how FAULTLESS can be directly applied to neural decompilation with state-of-the-art models.

5.2 Faultless

The FAULTLESS proof system contains three main components: a compiler, a symbolic interpreter, and a proof engine, as shown in Figure 5.2. The technique takes two functions as input—in evaluation, the predicted (neurally decompiled) code and the reference original code, and in validation, the deterministically decompiled code and the predicted code—and returns a proof result, which can be “equivalent” or “unable to show equivalent.” (Because FAULTLESS is sound, there may be equivalent functions that it is unable to prove equivalent; these cannot always be

distinguished automatically from nonequivalent functions. For convenience, we may refer to an “unable to show equivalent” judgement as a “nonequivalent” one.) The first two components are run in sequence for each function. Then the resulting execution contexts are compared in the proof engine.

In the following section, I’ll begin by defining the objective: that is, what I mean by “equivalence.” I’ll then introduce two core abstractions, the execution model and the memory model, that influence nearly every other component. Finally, I’ll step through each component of faultless in order.

5.2.1 Defining Equivalence

The way FAULTLESS defines equivalence is carefully calibrated to balance utility and a robust notion of equivalent observable behavior. A C function can return a value, modify program state that persists beyond the deallocation of its stack frame (i.e., the state of the heap or global variables), or read input or output; FAULTLESS captures all of these. In this section, we discuss the equivalence criteria in an informal way to provide intuition for what FAULTLESS is trying to achieve and to motivate design decisions. We will formalize these conditions once the execution model, memory model, and other abstractions have been defined.

FAULTLESS uses a broad notion of functional equivalence. Two programs are functionally equivalent if they output the same values given the same inputs. FAULTLESS’ notion of inputs and outputs are not limited to the values of parameters and return values. FAULTLESS also considers the values reachable from pointers; that is, the values in memory found at the memory address stored in the pointer. Global variables and function calls can serve as inputs and/or outputs, and FAULTLESS models these as well.

The minimum necessary conditions for FAULTLESS to declare equivalence (hereafter referred to as the “necessary equivalence conditions”) are, when given the same input:

- the two functions return the same values.
- if the return values represent valid memory addresses, then memory on the heap reachable from the return value in each function is equivalent (contains the same values at the same offsets).
- at the function’s exit, memory on the heap reachable from each argument to the function is equivalent to the memory reachable from the corresponding argument in the other function. This captures pass-by-reference behavior. Note that by context independence, FAULTLESS doesn’t know if the pointee referenced a pass-by-reference argument is on the heap or the stack (a fact which may differ by callsite), so without loss of generality FAULTLESS assumes that memory addresses in arguments are locations on the heap.
- function callees, functions that are called inside the functions whose equivalence FAULTLESS is trying to determine, may also modify program state, read input or write output. Therefore, FAULTLESS requires each callee in one function is equivalent to one in the other. To prove one callee equivalent to another, faultless requires:
 - the arguments to function callees are the same.
 - to capture pass-by-reference behavior to function callees, memory on the heap reach-

Table 5.1: Options FAULTLESS provides to loosen equivalence checking to accommodate common decompiler patterns that result in trivially nonequivalent behavior.

Code ID	Code Name	FAULTLESS Option Name	Description
C0.b.	Incorrect type	Math-integer modeling	Model integers as mathematical integers ($\mathbb{Z}$) instead of bit-vectors
C0.b.	Incorrect type	Decompose compound values in parameter lists	In decompiled code, structs in argument lists are often represented as multiple independent arguments in sequence. Perform the same flattening operation on the original code's structs in argument lists.
C4.a.i.	Extra arguments	Ignore extra arguments	When extra arguments are provided to the function or its callee, assume that they are decompiler artifacts and ignore them.
C12.	Incorrect return behavior	Ignore mixed return behavior	If one function is void and the other is not void, do not compare return values.
C0.c.	Incorrect function name	Require exact function names	Require equivalent functions to have equivalent names. Off by default.

able from each argument is equivalent to memory reachable from the corresponding argument in the other callee.

- function callees occur under equivalent control-flow conditions.
- function names and global variable names are consistent between the two functions. Neural decompilation may rename global variables and function callee names, so they may not exactly match those in the decompiled code or those in the original code. Instead of requiring them to match exactly, which is often too strict in the context of both the evaluation and validation tasks, we require a bijective mapping to exist between the function names in each input function and likewise for the global variables. We discuss consistency in more detail in Section 5.2.6.e.
- mapped global variables store the same values.

It is also instructive to consider what FAULTLESS does *not* require to show equivalence. FAULTLESS intends to capture algorithmic behavior while abstracting away compiler-specific details wherever possible. Therefore, FAULTLESS may declare two functions equivalent even if they have observably different side channels or may be vulnerable to different specific exploits (e.g. those that depend on a specific stack layout). There are two other noteworthy areas in which FAULTLESS does not require equivalent behavior or equivalent state: in handling decompiler artifacts during validation, and with regards to side effects.

As previously noted, decompiled code is often not semantically equivalent to the original source code it was compiled from, though often in very well-understood ways. To prevent trivial “nonequivalent” results, FAULTLESS is equipped with a range of options each of which narrowly loosens the necessary conditions for equivalence, each of which is derived from a code from the codebook in Chapter 3. These options are summarized in Table 5.1.

Additionally, FAULTLESS does not require that side-effect-inducing expressions (such as calls to other functions) be in any particular order. For instance, FAULTLESS judges the following functions equivalent:

```
1 void foo() {
2     printf("a\n");
3     printf("b\n");
4 }
```

```
1 void foo() {
2     printf("b\n");
3     printf("a\n");
4 }
```

The alternative—requiring that side effects do occur in a particular order—would be very limiting. Due to the context-independent nature of the problem, FAULTLESS does not know which callees and callers have side effects, nor does it know which global variables are read or written from within each callee. This means that to properly model function calls, FAULTLESS would have to conservatively assume that every function has side effects and that it reads from and writes to every global variable². Such assumptions would make the equivalence checking very brittle, limiting the utility of FAULTLESS.

Finally, the current implementation of FAULTLESS has several soundness limitations which mean that functions which produce different outputs for the same inputs may be judged equivalent. These are discussed further in Section 5.5.1.

5.2.2 Execution Model

To prove two functions equivalent, FAULTLESS must show that for any input, the two functions return the same value, read the same inputs, produce the same outputs, and modify program state in the same way. To do this, FAULTLESS uses a symbolic-execution-based approach, providing each function with a set of arbitrary inputs in the form of symbolic variables. It then incrementally builds up mathematical expressions in terms of these symbolic variables by executing versions of each operation (i.e. `+`, `-`, `sizeof`, etc.) which have been redefined to operate on symbolic expressions instead of the usual concrete values.

Unlike many contemporary techniques, FAULTLESS uses static symbolic execution instead of dynamic symbolic execution. In symbolic execution, when a branch instruction is encountered, if the value of the conditional is a symbolic expression containing a symbolic variable, execution may fork to *both* the true and false branch. In dynamic symbolic execution, the two paths are maintained separately in the symbolic execution interpreter, and are both executed to completion. Usually there is just one copy of the interpreter which alternates between different paths, executing a fragment of one, then another. Unfortunately there may be a number of paths exponential in the number of branch instructions encountered, a problem known as the path explosion problem. And because loop branch conditions may be defined in terms of symbolic variables, there may in fact be an unbounded number of possible paths, meaning symbolic execution on any function with such a loop will never terminate.

Static symbolic execution is different in that it features path *merging*. If two paths would execute the same code, the interpreter halts execution and combines the state of the two paths before proceeding. This means combining the modifications made to the stack and heap from each path, as well as disjoining the path conditions from each path, signifying that that code fragment could be reached by following one path or the other.

More concretely, FAULTLESS' execution model operates on basic blocks in a control flow graph. Basic blocks are sequences of instructions with no intervening control flow which serve as nodes in the control flow graph. A basic block is ready to execute when all of its predecessors in the control flow graph have been executed. The entry basic block has no predecessors and so is ready to be executed immediately when the interpreter starts up. Each basic block is executed in sequence with no interruption. When the basic block is finished executing, its final state—the

²Effectively, this means that FAULTLESS assumes that callees don't read or write to global variables.

state of the stack, heap, and path condition to this point—are saved. Then, for each of that block’s successors, the interpreter checks to see if all of the successors’ predecessors have been executed and thus have final states saved. If so, then it merges the incoming states, and adds the merged state and the corresponding basic block to a queue of “ready” basic blocks.

5.2.2.a Loops

The above algorithm is only guaranteed to progress through the control flow graph if the control flow graph is acyclic. However, this is not the case for many control flow graphs due to the presence of loops.

To address this problem, FAULTLESS identifies which basic blocks are impacted by identifying loops from the control flow graph using a standard natural loop analysis pass [2, 44]. (Unnatural loops—which in C can only be created with `gotos`—are not supported.) FAULTLESS chooses the loop head as the location to break the cycle, because a loop head is always the first basic block in the loop that is reachable from the entry block of the function. When a loop head is encountered, only the predecessor from the outside of the loop will have a completed end state. To make up for the lack of other predecessors, FAULTLESS abstracts over all possible values that the variables could take on during the loop, assigning each variable whose value changes in the loop a unique symbolic variable. FAULTLESS identifies such variables using SSA (Single Static Assignment) form [3, 84] ϕ instructions, so these variables are referred to as ϕ -variables. In SSA-form code, each variable in the program is mapped to a collection of variables, each of which is written to exactly once. ϕ instructions are flow-sensitive copy operations placed at the start of basic blocks that receive a different value for a program variable from different successors. A ϕ instruction at a loop head, then, indicates precisely the program variables that are mutated by the loop and thus must be abstracted over by FAULTLESS with a symbolic variable. In essence, ϕ variables parameterize an arbitrary loop iteration, just as symbolic arguments to the function’s parameters represent an arbitrary call to the function.

The introduction of ϕ variables mean that symbolic expressions that occur during the execution of the function may be defined in terms of not only the function inputs, but also one or more ϕ variables. This is problematic for some tasks that symbolic execution is traditionally used for, such as finding example inputs that lead to a crash or vulnerable state. However, it is actually advantageous here, because of the way FAULTLESS performs inductive proofs of loops in the proof engine (Section 5.2.6.b).

Because loop bodies can be executed multiple times, a given ϕ variable can represent multiple dynamic values. This stands in contrast to a regular symbolic variable, which represents a singular dynamic value. A variable that is modified during a loop may be used after that loop terminates, but at that point, the variable also represents a singular dynamic value. This distinction is important with regard to the memory model. Therefore, for each ϕ variable, we introduce a corresponding post-loop ϕ variable, which we denote $\hat{\phi}$. We refer to “regular” ϕ variables that model an arbitrary loop iteration as *mediloop* ϕ variables, from the Latin word for “middle.”

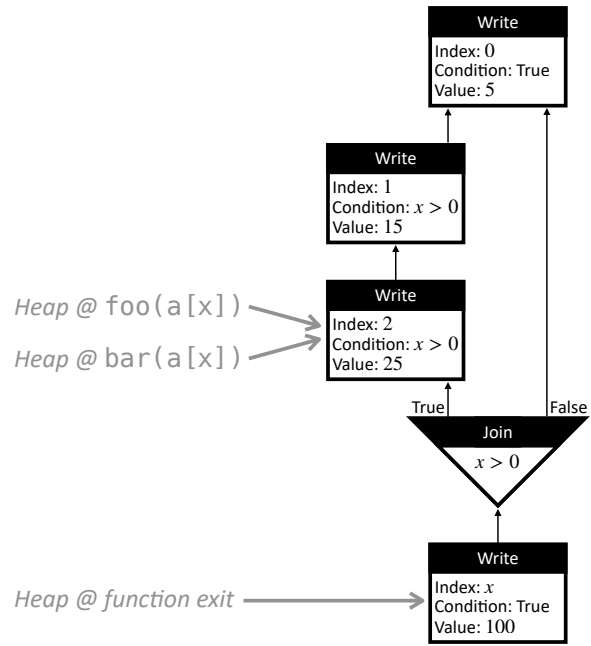
```

1 void func(int *a, int x) {
2     a[0] = 5;
3     if (x > 0) {
4         a[1] = 15;
5         a[2] = 25;
6         if (x < 3)
7             foo(a[x]);
8         else
9             bar(a[x]);
10    }
11    a[x] = 100;
12 }

```

(a) A function which writes to *a* at different points and offsets in the control flow graph.

Figure 5.3: A function and the address space graph for the heap variable *a*. If there were more variables written to in `func`, there would be an address space graph for each one. Address Mapping objects like the Heap and Stack map from the base address to a node in the AddressSpace graph, representing the state of memory at that point in the code.



(b) Heap Address Space Graph for *a*

5.2.3 Memory Model

FAULTLESS features a 3-layer memory model consisting of virtual registers, the stack, and the heap. Registers are the simplest part of the memory model. They store intermediate values in multi-operation expressions which are not assigned to any particular program variable, like the value $x * 2$ in the expression $y = x * 2 + 1$. The FAULTLESS interpreter has an unbounded number of virtual registers.

Much of the power of FAULTLESS’ memory model, however, comes from the stack and heap. The stack and heap are each instances of an abstract Address Mapping data type. An Address Mapping is a map from a base address to a segment of memory, similar to a “heaplet” in separation logic [83]. Each memory segment is its own unbounded address space. The stack is an Address Mapping indexed by program variables (i.e. variables declared in a program that represent memory locations where data is stored), and the heap is an Address Mapping indexed by symbolic-execution-style symbolic variables. We will write program variables in monospaced font (*p*), and symbolic variables in italics (*p*).

The values of an Address Mapping’s key-value store are themselves an abstract data type, an Address Space. An Address Space is conceptually a flow-sensitive unbounded symbolic array implementing McCarthy’s array axioms. Each Address Space is implemented as an immutable directed acyclic graph (DAG); Figure 5.3 shows an example. Writing to an address space means adding a node to the graph, with a singular outgoing edge pointing to the most recent prior value. In this way, the graph represents the complete “history” of all writes to a given base address. The Address Mapping stores the pointer to the most recent node in the history graph. In this way, an Address Mapping can represent a “snapshot” of the state of memory at a given point: because the

graph is immutable and the pointers in the graph point only to the past, pointing to a given node represents the state of memory immediately after the memory write pointed to was executed. For instance, in Figure 5.3, the state of the Heap saved for checking the necessary equivalence conditions of the calls `foo` and `bar` is at the write on line 5, while the state of the Heap at the function’s exit is at the write on line 11.

An Address Space is a graph rather than just a linked list because of how control flow interacts with memory. Different values may be written down different paths. When a branch instruction is encountered each resulting path receives a copy of the stack and heap. The mappings can be mutated independently, adding new nodes which point to the most recent prior node and creating a “fork” in the memory graph.

Path merges create an additional challenge because there are multiple possible write histories. To handle this, Address Space graphs include another type of node, a Join node. Join nodes are parameterized by a boolean condition and point to two prior history nodes, one for the true branch and one for the false branch. The boolean condition used is the boolean condition found in the branch which originally caused the divergence in control flow.

5.2.3.a Reading

While a write simply involves appending the value written to the Address Space graph, it is not immediately clear how to *read* a value from this graph. To do this, we first need to examine what is stored in each Write (non-Join) node of the graph. Each Write node contains three attributes:

- The *index*, relative to the base address, at which the value is stored. For a scalar variable this will simply be 0, but it could be any symbolic expression for an array variable (e.g. for the expression `a[x + 1]`, the index would be $x + 1$, or, more precisely, the symbolic value stored on the stack in the program variable `x`, which we can’t know without knowing the preceding code, plus one).
- The path condition at which this write occurred. We call this the *write condition*.
- The value written.

An address space represents a conceptually unbounded chunk of memory, with each storage location represented by a unique ID, or address, as is the case in virtual memory implemented in modern operating systems. Unlike in virtual memory, the unique IDs are represented as offsets relative to the base address, starting from zero.

A read query is parameterized by an index, which may be symbolic, and the path condition at which the read occurs, the *read condition*. To read a value, the memory graph is searched, starting from the node pointed to by the Address Mapping for that base address. If the indices are equivalent under the read and write conditions, then the value stored in that write node should be returned.

That is the ideal, anyway, but unfortunately things do not always work out this cleanly. The property of reads that makes reading difficult is that indices may be arbitrary symbolic expressions. While a concrete value like 0 or 8 represents a fixed offset with respect to the base address, a symbolic offset like i may be any offset, so, depending on the value that i takes on, this value may or may not be read. For instance, if we previously executed `arr[0] = 7` before executing the read `arr[i]`, (under empty read and write conditions) the index i could take on the value 0,

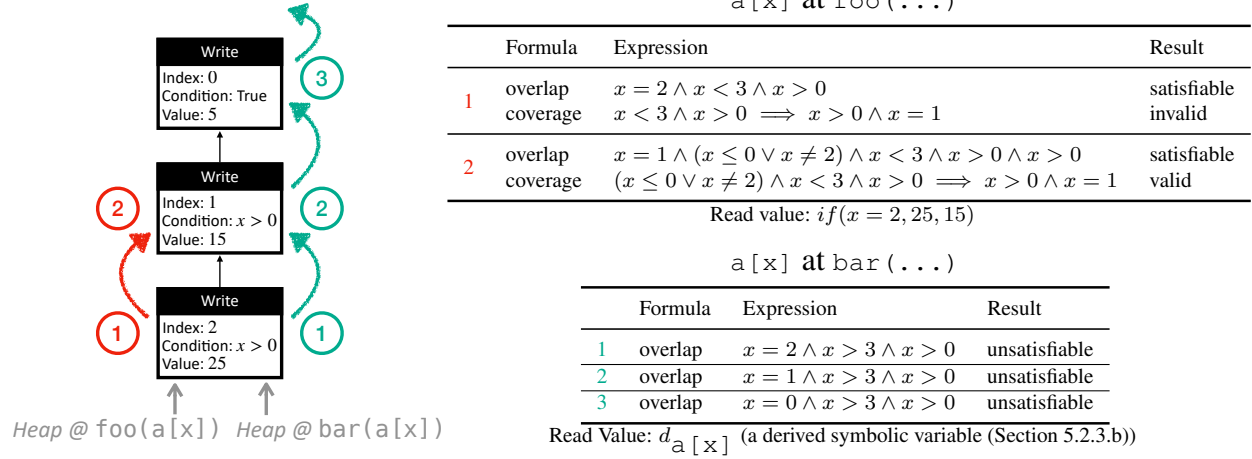


Figure 5.4: A representation of the reads that occur in the argument lists of `foo` and `bar` in Figure 5.3a. Both read $a[x]$, from the same heap, but the differing read conditions cause the two reads to return drastically different values. To perform a read, FAULTLESS traverses the Address Space graph, evaluating coverage and overlap formulas for satisfiability and validity, respectively. A valid formula is one that is always true. In practice, the symbolic value p_1 , representing the value to the first parameter, would be stored in x , but we use x here for clarity.

or it could not, so the read value could be 7, but may not be. However, under the read condition $i > 0$, it is not possible for i to be 0, and it is therefore impossible for the read value to be 7 (at least from this Write node). And of course if the read index was also 0 instead of i (reading back `arr[0]`) then the read value would be guaranteed to be 7.

To handle this complexity, FAULTLESS introduces two concepts: overlap and coverage. If a given read query *overlaps* the write, that means there is at least one index where the read and write condition are equivalent, subject to the read and write conditions. If a given write query *covers* the read, that means that all indices in the read query have a matching index in the write. There are three possibilities:

- no overlap: this write cannot be the read value. Search deeper through the write history. It is not necessary to check coverage.
- overlap but no coverage: this write may contribute to the read value, but we also need to find other possibilities in the write history.
- coverage: the read value is the value stored at this Write node.

Figure 5.4 contains examples of all three situations; steps 1, 2, and 3 for `bar(a[x])` are no overlap read-write interactions, step 1 for `foo(a[x])` represents an overlap but no coverage interaction, and 2 for `foo(a[x])` represents a coverage interaction.

When there is overlap but no coverage, multiple different read values are possible. FAULTLESS uses symbolic “if-then-else” expressions (`z3.If`) to represent them as a single value. The condition of the if-expression is a symbolic formula which captures the indices read from in this read/write interaction, the consequence is the value of the current Write node, and the alternative is the result from recursively calling read on the next node in the Address Space write history graph. The recursive call includes an additional constraint, the negation of the if-expression’s

condition, that prevents recursive read calls from matching over the same indices that overlap the current (and thus more recent) write, progressively refining the query to a smaller set of indices.

Let κ_r be the read query index, and let κ_w be the write query index. Let π be the read condition, and let ω be the write condition. All are symbolic formulas. We discharge queries over symbolic variables to the z3 SMT solver [31]. The formulas for overlap, coverage, and the recursive constraint are as follows:

Overlap. For a write to overlap the read, subject to the read and write conditions, there must be at least one satisfying assignment of values to variables that makes the read and write indices equivalent and for which the read and write conditions are both true. Thus, the overlap formula is a simple conjunction of the equality of the indices and the read and write conditions. Formally, there is overlap if the following formula is satisfiable:

$$(\kappa_r = \kappa_w) \wedge \pi \wedge \omega \quad (5.1)$$

Coverage. For a write to cover a read, every possible index specified in the read query must also exist in the write. Read indices are subject to the read condition, so we need only consider the cases when the read condition is true. But otherwise, for every possible value of the variables in the read and write index formulas, the read indices must be equivalent, and the write condition must allow for them to be. Formally, the write covers the read if and only if the following formula is valid, that is, always true for any assignment of values to symbolic variables:

$$\pi \implies (\omega \wedge (\kappa_r = \kappa_w)) \quad (5.2)$$

This simple form of the equation works in loop-free code. However, mediloop ϕ variables represent multiple dynamic values at once, as opposed to regular symbolic variables, which only represent one. This means we must quantify over mediloop ϕ variables. Let $\phi_{r_0} \dots \phi_{r_n}$ be the mediloop ϕ variables that occur in the read index or read condition symbolic formulas, and let $\phi_{w_0} \dots \phi_{w_m}$ be the mediloop ϕ variables that occur in the write index or write condition symbolic formulas. (We do not have to be concerned about this for overlap because SMT expressions are implicitly existentially quantified and overlap is by its nature purely existential). The formula says: for all possible mediloop read ϕ values that satisfy the read condition, there exists a combination of corresponding write mediloop ϕ values subject to the write condition such that the read and write indices are equivalent. Formally, this is:

$$\forall \phi_{r_0} \dots \phi_{r_n}. \pi \implies (\exists \phi_{w_0} \dots \phi_{w_m}. \omega \wedge (\kappa_r = \kappa_w)) \quad (5.3)$$

Because validity is the dual of satisfiability and SMT solvers find satisfying assignments of values to variables, we ask the solver if the negation of Equation 5.3 is unsatisfiable.

If-Condition. The consequence (the “true” branch of the symbolic “if” expression) of an overlapping but uncovered read is the value stored in the write node. The condition of the if-expression, then, must be true for all values for which the read and write indices are equivalent, and false when they are not. This is similar to what the coverage condition, which Equation 5.3 expresses. However, we do not include the read condition. By construction, the read condition holds over the entire value read. Including the read condition in the if-expression condition would provide

an interpretation of the read value where the read condition is not met, which is nonsensical. Formally, the if-condition is:

$$\exists \phi_{w_0} \dots \phi_{w_m} \cdot \omega \wedge (\kappa_r = \kappa_w) \quad (5.4)$$

The negation of this formula helps refine recursive queries. Intuitively, new writes to the same address in memory should overwrite older ones at that same address. Equation 5.4 formula describes all addresses that are read from the current Write node. We want to ensure that subsequent reads do *not* read from those indices, so we assert the *negation* Equation 5.4 in conjunction with the read condition. In step 2 for `foo(a[x])` in Figure 5.4, the refinement is the negation of $x > 0 \wedge x = 2$, which is $x \leq 0 \vee x \neq 2$.

Join-Nodes. Join nodes represent places in the program where write history to memory differs depending on the path taken. To read from a join node, FAULTLESS also uses an if-expression. FAULTLESS recursively reads down both the true and false branch, asserting the branch condition and its negation, respectively, as additional refinements. The if-expression is parameterized by the branch condition, with the result of the recursive call on the true successor of the Join node assigned as the if-expression’s true argument and the result of the recursive call on the false successor of the Join node assigned as the if-expression’s false argument. If the Join-node’s branch condition is unsatisfiable under the read condition then instead only the value from the false branch’s recursive call is returned. Likewise, if the negation of the Join-node’s branch condition is unsatisfiable under the read condition, then only the value from the true branch’s recursive call is returned.

5.2.3.b Context Independence, Progenitor Variables, and Derived Variables

Suppose that there is no write in the write memory graph which completely covers the read for a given query. In the simplest case, this occurs when a read happens before any write to a given address, such as in the snippet

```
1 void foo(int* ptr) {
2     int first = ptr[0];
3     // ...
4 }
```

FAULTLESS’ answer to this problem lies in its context-independent nature. By being context-independent, FAULTLESS assumes nothing about the calling context (nor the definitions of callees or global variables). Thus, FAULTLESS knows nothing about the values in the chunk of memory that the argument `ptr` is pointing to, nor does it even know the size of that chunk of memory. Instead, FAULTLESS *records the assumptions* that the function makes about the calling context—in this particular instance, about the calling context memory. It allocates a fresh, unique symbolic variable for each combination of base address and accessing index, symbolizing that the function assumes that there is a value at that location, or those locations if the index is a symbolic expression capable of representing multiple values.

We call symbolic variables create this way *derived variables*. Derived variables are derived from other derived variables (as in `ptr[0][0]`) or *progenitor variables*. Progenitor variables serve as the roots of derivation trees. They come from five different sources:

- symbolic parameters. As mentioned in Section 5.2.2, FAULTLESS provides a set of symbolic parameters to each function to begin execution. The variable derived from `ptr` above is derived from a symbolic parameter.
- ϕ variables, both mediloop and postloop, as described in Section 5.2.2.a.
- function calls. Calls to other functions are treated as uninterpreted, in keeping with the context-independent assumptions of FAULTLESS, and each function call is associated with a unique symbolic value. For more information about function calls, see Section 5.2.5.b.
- global variables. By context independence, FAULTLESS know nothing about the values in global variables when the function is called so each global is initialized with a unique symbolic variable.
- uninitialized local variables. FAULTLESS initializes all parameters with symbolic values, and most other local variables are initialized with symbolic expressions defined in terms of those variables or with constants. But variables which are read before they are initialized get their own unique symbolic values. Unlike with the other three types of derived variables, we know what is present in uninitialized locals, or, rather, that they contain no particular value.

The majority of the work FAULTLESS' proof engine (Section 5.2.6) does involves mapping progenitor variables to each other so that symbolic variables in expressions across different functions are defined in terms of the same variables. Once progenitor variables are mapped together, their derived values can be mapped together as well based on the corresponding index expressions.

Derived symbols are created and stored by the Memory Symbol Factory in the interpreter, as shown in Figure 5.2.

5.2.3.c The Stack and Compiler Independence

The stack is simply an Address Mapping indexed by program variable. Each program variable is mapped to its own Address Space. As noted above, each Address Space represents an unbounded set of addresses, with the base address mapping to index 0. This choice may seem odd for modeling a frame on the call stack, since we know the types of all local variables and thus the amount of space each local variable takes up. This stands in contrast to the heap, for which, by context independence, we know nothing of the size of the memory allocated at each base address so we cannot bound heap Address Spaces by any particular concrete value.

However, using an Address Space model is helpful for the Stack for different reasons. In addition to being context-independent, FAULTLESS is, wherever possible, compiler-independent as well. That means FAULTLESS makes no assumptions about the layout of the stack frame, such as the order of variables in the frame or the amount of padding placed between them. The nature of Address Mappings and Address Spaces allows for this compiler-independent behavior.

For the vast majority of regular local variables, though, all reads and writes just occur at index 0, so each write covers each read and the value read is simply the last value written.

5.2.3.d Addressability

Both reads and writes to memory happen relative to some base address. Unlike in normal program execution, these base addresses are not merely integers but are instead symbolic values (for the heap) or program variables (for the stack). FAULTLESS determines what the base address is for a given expression through its *addressability model*.

A key challenge in modeling addressability comes from the validation task, in which FAULTLESS must compile and execute decompiled code, which features frequent integer-to-pointer casts. For instance, a frequent occurrence in code produced by the Hex-Rays decompiler is a parameter typed as a `long long` which is then cast to a pointer type and dereferenced. The variable `a1` in Figure 3.1 is an example of this. A memory address, is, after all, just an integer identifying a particular virtual memory cell; pointers wrap a layer of abstraction around memory addresses, modeling how much memory is being referred to starting from that cell and how to interpret the bit pattern found in those memory cells. Decompilers simply sometimes strip variables of that semantic wrapping and then manually apply it later. Thus, we must allow integers to serve as legitimate base addresses as well. In fact, *any* integer or pointer symbolic value is a valid base address: by context independence, FAULTLESS doesn't know whether a given symbolic variable is a memory address or not based on the calling context, so if the function uses the symbolic value as if it is a memory address, FAULTLESS records the assumption by treating it as one.

The addressability model is most directly derived from the properties of memory addresses in position-independent code. FAULTLESS determines the base address not only from the type but from *how each value is used*. Because memory addresses in position-independent code may vary from execution to execution, only certain operations on valid memory addresses preserve the property that the result may be a valid memory address on an arbitrary run. For instance, multiplying an address by two will result in a value that is likely outside of the address space to or from which the process can read or write. Instead, in position-independent code, valid memory addresses can only be obtained by adding or subtracting offsets relative to an existing address.

Addressability is thus a recursive property. Plain symbolic values form the base case. Then if a symbolic expression is addressable and it is added or subtracted from another symbolic expression, the resulting expression is also addressable. There is a caveat here: if *both* of the operands to an addition are addressable expressions, what is the resulting base address? After all, it could be that both operands represent memory addresses, and summing or differencing two memory addresses also leads to an invalid address. In this case, FAULTLESS takes a hint from the operands' types: if one is a pointer and the other an integer, then the pointer is the base address. If both are integers (both pointers is a compilation error) then the resulting expression is not addressable because the base address cannot be determined. Attempting to store a value at a non-addressable lvalue is an execution error.

Supporting integer-to-pointer casts as a first-class feature is both extremely important for supporting decompiled code but also relatively unusual in symbolic execution and equivalence checking techniques; for instance, Alive2 [69] explicitly disallows integer-to-pointer casts, and Klee [16] has very limited support for them.

5.2.4 The Compiler

FAULTLESS' compiler converts each C function into a high-level three-address-form intermediate representation (IR). To preserve compiler independence, the level of abstraction is kept as high as possible, preserving source-level details. In FAULTLESS, compilation, then, serves less as a method of conforming the software to the hardware and more of a way of providing a distinct partial ordering of operations for execution, breaking down large expressions into smaller ones with intermediate results being stored in temporary variables. The control flow graph is built directly as a first-class feature of IR generation instead of inferred later from branch instructions and their targets in the code.

The FAULTLESS compiler module is based upon the compiler model from codealign [34], though it features numerous improvements, including the ability to parse and model types. (Codealign is type agnostic.) We modified the type-parsing code from the IDIOMS project and incorporated it into the compiler to help accomplish this.

Context Independence. Because FAULTLESS does not require the definitions of callees or globals, it must determine which identifiers correspond to these constructs, which it does through a collection of heuristics, inherited from codealign [34]. If an undeclared identifier is used as the name of a function in a call expression, we assume it is a function. If an undeclared identifier is used as a value, it's assumed to be a global variable. If the definition of a global variable or a function is given, FAULTLESS will parse and use it, making it unnecessary to infer that particular identifier's identity. FAULTLESS does require the definitions of *types* used in the function, though as shown in Chapter 4, neural decompilers like IDIOMS can provide them.

5.2.4.a Type Deduction

FAULTLESS applies C's rules around integer promotions, arithmetic type conversions, implicit casts, and assignment semantics to derive the types of intermediate temporary variables and to typecheck the code. We refer to this process as *type deduction* to distinguish it from the process of *type inference*, described in the next section.

Deterministic decompilers may not be strictly compliant with the C standard's typing rules, which can lead to trivial typing errors when compiling decompiled code for the validation task. In particular, they often feature size-based placeholder types, like Hex-Rays' `_QWORD` and Ghidra's `undefined8`. The FAULTLESS API features a way to specify which identifiers correspond to these types and their sizes. They're treated as supertypes of all primitive types of the same size.

5.2.4.b Type Inference

When FAULTLESS infers that a given identifier is a global variable or function, it does not inherently know that variable's type, and it is initially given a special `UnknownType`. But types are necessary to define the functions' semantics. As a result, FAULTLESS features a module that infers the types of these symbols based on the context in which they are found in the function.

The C programming language's typing rules only provide unique types when applied in the forward direction. For instance, if `g` is a global variable, we can conclude from the snippet

`short x; g = x + 1;` that `g` has type `int` by C’s integer promotion rules.³ In contrast, there are multiple possible types when applying the rules backwards: in `int x = g + 1;`, `g` could be a `char`, `unsigned char`, `short`, `unsigned short`, `int`, or even `_Bool`.

The challenges associated with inferring types for C code are well studied, including for use in decompilers themselves [15, 65, 77, 94] and for C code more generally [71, 72]. FAULTLESS’ type inference module was designed in collaboration with and synthesized by AI based upon common patterns in related work.

The type inference module assigns a type variable to each unknown type, which is associated with a domain, a set of possible types that the variable could be. The inference module generates constraints over those type variables based on their usage. For instance, if a variable with an unknown type is used in a bitwise `&` operation, it must be an integer type. If a variable with an unknown type is dereferenced, it must be a pointer type (though more evidence is required to determine the pointee type). Applying a constraint to one or more type variables narrows the type domain. The constraint solver uses a worklist algorithm to propagate constraints, using information in a given constraint to narrow the domains of type variables used in the constraint. Whenever a variable’s domain reduces in size, constraints mentioning that variable are added to the worklist. The process proceeds to a fixed point.

If a type variable’s domain is empty, then type inference has detected an inconsistency in the typing of the input function. However, it is entirely possible and indeed common for a type variable’s domain to contain more than one type at the end of type inference; these represent the set of types that are consistent with the typing evidence available. This is the case for the `int x = g + 1;` example from earlier (assuming that `g` is not used elsewhere in the function). To handle this, the type inference module maintains a set of preferences associated with each type choice. Among integer types, for instance, the smallest type that can hold the value without loss of information is preferred. If there is not enough information to determine this, then `int` is chosen.

After type inference completes, type deduction is run again to propagate the inferred types of global variables and function signatures through all of the program’s temporary variables.

5.2.4.c Preparing Loops

To further prepare the FAULTLESS IR, it is converted into SSA form. The interpreter directly executes the SSA form IR; the ϕ instructions at loop heads are important for FAULTLESS’ abstracted iteration summary execution (described in Section 5.2.2). We refer to these ϕ instructions as *loop- ϕ instructions*. To differentiate loop- ϕ instructions from other ϕ instructions, FAULTLESS performs a standard loop-analysis pass, then merges together all natural loops with the same head. (Natural loops are uniquely identified by back edges; two natural loops can have the same head if a `continue` statement is used.) The operand of each loop- ϕ which does not come from within the merged loop body is recorded as the base-case operand.

³Actually, `g` could still be any integer type or even a float type because C will implicitly cast this integer-typed `x + 1` expression to the declared type of the variable. But from a type inference perspective, the typing rules do provide an obvious unique type in this case.

5.2.5 The Interpreter

The interpreter implements the execution model (Section 5.2.2) for FAULTLESS IR. It builds expressions in terms of symbolic variables, and logs information around the return state and function calls for use in the proof engine.

The interpreter initializes the stack by writing position-dependent symbolic variables to each local variable on the stack. Here we'll denote them $p_0, \dots, p_n$. Each global variable is also initialized with a unique symbolic variable based on its name.

5.2.5.a Interactions with Memory

The interpreter directly executes the SSA-form IR. Each instruction which produces a result is assigned a unique variable; each variable is assigned a register. Registers are implemented as a simple dictionary mapping an instruction to the result of executing that instruction.

If an instruction's result corresponds to a declared program variable instead of a compiler-generated temporary variable, then the result of executing that instruction is also written to the stack. Likewise, when an instruction's SSA variable is used as an operand to another instruction, if that instruction wrote to a variable on the stack, that value is read from the stack instead of from the registers. This is for two reasons. First, unlike temporary variables, declared program variables can have their addresses taken and can be read from and written to by reference. Second, the stack already features logic for managing path merges, so ensuring that values are stored there obviates the need for a separate path-merging implementation for registers. An implication of this second point is that ϕ instructions which are not loop- ϕ instructions are ignored.

Struct values are written to memory by writing each field to the stack or heap, at the offset given by the ABI for that struct's layout. This partially breaks compiler independence, or at least platform independence (currently it is implemented for x86_64) and is the reason FAULTLESS requires UDT definitions. However, this approach is important for processing decompiled code, where struct-field access operations are often decomposed into arithmetic with a hardcoded offset, a typecast to a pointer, and a dereference, as in Figure 3.1. Mapping back to numeric offsets helps ensure that values written to the same fields end up in the same offsets in memory across both the decompiled and original versions of the code in the validation task. An alternative approach would be to treat field names as symbolic variables representing an unknown offset, but these would have to be carefully constrained to take on consistent values across various memory interactions and to prevent two different fields from being assigned the same offset.

5.2.5.b Executing Function Calls

Consistent with context independence, FAULTLESS treats function calls as uninterpreted functions. Each semantically distinct set of function arguments is assigned a unique symbolic variable. Because function calls may return pointers to memory, these symbolic variables serve as progenitor variables (Section 5.2.3.b).

Modification-By-Reference Symbolic Variables. C functions may also have pass-by-reference behavior, meaning they may return results by modifying memory accessible from the input arguments. To model this, FAULTLESS features the option to write to the base address of each

addressable argument a unique modification-by-reference symbolic variable. This is a fairly disruptive and invasive option, so it is disabled by default; most functions do not modify any parameters by reference and those that do often only change one or two. Due to context independence, FAULTLESS cannot know which function arguments are modified by reference.

Stack Initializer Inference. There is one instance where modification-by-reference is especially common: when initializing local variables through functions like `fscanf` and `memcpy`. FAULTLESS also features an option to enable the stack-initializer-inference heuristic: if a local variable is uninitialized and its address is passed to a function call as an argument, that function argument is inferred to be an initializer argument. Under this option, only initializer arguments are assigned their own unique modification-by-reference symbolic variables. This option is on by default.

5.2.5.c Entering and Exiting Loops

Under the execution model (Section 5.2.2), loops are executed once, with values changed during the loop abstracted to mediloop ϕ variables to represent an arbitrary loop iteration. Mediloop ϕ symbolic variables have special semantics in that they represent multiple possible dynamic values instead of a single, arbitrary dynamic value like a regular symbolic variable, so FAULTLESS must convert these to distinct postloop $\hat{\phi}$ symbolic variables after the loop: if the corresponding program variable is read from after the loop, we would expect the value stored there to be in terms of its postloop symbolic variable.

To accomplish this, FAULTLESS ascribes special semantics to loop- ϕ instructions. Each loop head is executed twice, once before and once after the execution of the loop's body; different semantics are given to the instruction on the first and second executions. Conveniently, all program variables whose values change during the loop have a ϕ instruction at the loop's head, and ϕ instructions are placed at the start of the basic block, before all other instructions. Therefore, on the first execution, ϕ instructions simply write the corresponding mediloop ϕ variable to the stack location for the corresponding program variable. On the second execution, the ϕ instruction writes the postloop $\hat{\phi}$ symbolic variable to the same stack location. (Loop ϕ instructions have additional semantics, which will be covered in the next two sections).

FAULTLESS' compiler emits two types of branch instructions: `Loop` and `If` branch instructions. `Loop` branch instructions are critical to facilitating this process. The first time a `Loop` branch is executed, only the true-branch successor is marked as ready-to-execute. Thus, the stack initialized with mediloop ϕ variables is used for the loop body. On its second execution, a `Loop` branch instruction's false branch will be marked as ready-to-execute. By this point, the ϕ instructions will have written postloop $\hat{\phi}$ instructions to the stack, and so these are the exposed values that can be read by code after the loop for those program variables.

Loops may not be exited by just the regular looping condition, however; `break` statements can also be used to exit the loop, as can `return` statements, implicitly. At branches that exit the loop created by these statements, FAULTLESS implements a change-of-variables, reading the current value of each loop-impacted program variable, substituting each instance of the mediloop ϕ variable for the postloop one, and writing the resulting value back out to memory. This change-of-variables is also performed on the path condition.

5.2.5.d Pre-Execution and Loop Invariants

To better characterize loop iterations, where possible, FAULTLESS infers loop invariants for what we refer to as *generalized affine loops*. Let c be the change to the value stored in a given program variable v during the execution of an arbitrary loop iteration. When a value does not change during the loop, c is trivially nothing, so we concern ourselves only with program variables that are modified during the loop, which are precisely those that we initialize with mediloop ϕ variables. Then $c = \text{stack.read}(v) - \phi_v$, where the stack in question represents the state of memory after the arbitrary loop iteration. FAULTLESS' inference code supports generalized affine loops: those for which c is not a function of ϕ_v . We call these generalized affine loops because the i th iteration ϕ_{v_i} is of the form $ci + b$, where b is the base case given by the initial value of the variable before the loop. If c is strictly increasing, FAULTLESS also infers that $c \geq b$, and if c is strictly decreasing, then FAULTLESS infers that $c \leq b$.

The necessary conditions for a change to a variable to be a generalized affine update are fairly strict, but the majority of modifications to stack variables in loops are generalized affine: on REALTYPE, 71.5% of loop- ϕ variables meet the criteria and at least one invariant is inferred.

To infer these invariants, FAULTLESS performs an additional execution of the code before the main execution. The base cases and changes made during the loop are logged and used to compute the invariants. Then during the main execution, the invariants are injected into the path condition. If there are no loop- ϕ instructions in a function, the pre-execution is skipped.

5.2.5.e The Heap Replay Buffer

When a linked data structure is traversed in a loop, a loop- ϕ instruction captures the changing value of the pointer to the current part of the data structure during iteration. Intuitively, these modifications should impact the initial pointer as well; the subsequent read of `head->val` should logically return 0 in Figure 5.5. But because the stack slot for the modified variable (`current`) is initialized with a mediloop ϕ symbolic variable in the loop's head, the modifications are made relative to the ϕ symbolic variable, not the value in `head` (which here would be p_0 , the value of the first parameter). So by default, the memory model would return a fresh derived variable for the `head->val` memory read.

To handle this situation, FAULTLESS applies a *replay buffer*. We define the replay buffer as a function, $r(s)$, where s is a symbolic variable. The replay buffer captures reads and writes made to the heap relative to the mediloop ϕ symbolic variable ($r(\phi)$) through the duration of the loop.

At the beginning of the loop, the ϕ instruction records the base address b of the ϕ instruction's base case as corresponding to that variable's mediloop ϕ symbolic variable. In Figure 5.5, it records that ϕ_{current} maps to p_0 . Then if the base case's base address is read from, the replay buffer first applies replayed writes to the heap, writing $r(b)$, or for Figure 5.5 specifically, $r(p_0)$, which is just `head->val = 0`. FAULTLESS uses reads in the buffer to apply the replay buffer recursively. Thus `head->next->next->val` would also return 0.

```

1 struct node { int val; struct node * next; };
2 void foo(struct node * head) {
3     struct node * current = head;
4     while (current) {
5         current->val = 0;
6         current = current->next;
7     }
8     // ...
9     int value = head->val;
10    // ...
11 }

```

(a) Initializing a linked list in C.

```

1 %0 = head
2 %1 = phi(%0, %5)
3 loop %1
4     %3 = %1->val
5     %4 = store %3 0
6     %5 = %1->next
7 // ...
8 %2 = head->val
9 // ...

```

(b) Figure 5.5a in FAULTLESS IR. SSA variables are assigned names in the format %X, following the style of LLVM IR.

Figure 5.5: A function which initializes a linked list, then access a value written later from the head pointer.

5.2.6 The Proof Engine

The proof engine uses information recorded during the execution to attempt to show that the necessary equivalence conditions (Section 5.2.1) hold. We refer to the two functions being compared as the *left* and *right* functions.

In checking that the necessary equivalence conditions are met, FAULTLESS often checks if two symbolic expressions, one from the left function and the other from the right (e.g. return values or callee arguments), are equivalent to each other. Any expression can be viewed as a function of its *free variables*; $x + y + 2$ is a function of x and y , for instance. It only makes sense to consider the equivalence of two expressions if they are defined in terms of the *same* free variables. Unfortunately, this is not the case for many pairs of expressions in FAULTLESS because loop- ϕ instructions and function calls introduce fresh progenitor variables. Thus, a prerequisite to checking the necessary equivalence conditions is building a *coupling relation*, denoted C , between the two programs, populating it with pairs of symbolic variables shown to be equivalent to each other. These pairs of variables can be considered “the same” for the purposes of determining if two expressions are equivalent.

The process of building a coupling relation often itself requires a (smaller) coupling relation. For instance, in order to show two function calls equivalent (and thus add their progenitor variables to the coupling relation), it is necessary to show that each pair of arguments is positionally equivalent. Those arguments are symbolic expressions themselves defined in terms of symbolic variables, whose coupling relationship must be known prior to proving equivalence. Thus, building a coupling relation is an incremental affair that goes hand-in-hand with showing the necessary equivalence conditions.

5.2.6.a Synchronization Graphs

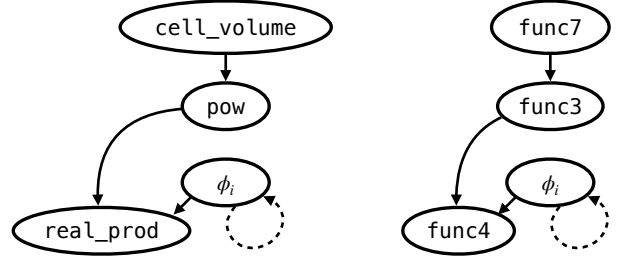
The key data structure which structures the equivalence checking process is the *synchronization graph*. Figure 5.6 shows an example. It is a directed, possibly cyclic, graph. The left and right functions each have their own synchronization graphs. The nodes in the graph are instructions that introduce progenitor symbolic variables (Section 5.2.3.b); that is, loop- ϕ instructions and

```

1 void scale_cell(double c[3][3], double s)
2 {
3     if (s < 0)
4         s = pow(-s/cell_volume(c), 1.0/3);
5     for(int i = 0; i<3; ++i) {
6         real_prod(c[i], s);
7     }
8 }

```

(a) A small function featuring several function calls and a loop. `cell_volume` is a dataflow dependency of `pow`, which in turn is a dataflow dependency of `real_prod`. The loop- ϕ for `i` is a dataflow and control dependency of `real_prod`.



(b) The synchronization graphs for Figure 5.6a (left), and a hypothetical decompilation of it (right). Dashed lines indicate a back-dependency edge going from the loop body to its head.

Figure 5.6: A function and the synchronization graph derived from it. A synchronization graph captures the control and dataflow dependencies between the instructions which introduce progenitor symbolic variables (Section 5.2.3.b), which we refer to as synchronization instructions.

function calls. The edges represent transitive dataflow and/or control dependencies between the nodes: if there exists some combination of dataflow and/or control flow edges between two progenitor-instructions in the IR, then there is an edge between them in the synchronization graph. If there is an edge from node n to node m in the synchronization graph, we say that m is dependent on n , and that n is a dependency of m .

Control dependencies are basic-block-level abstractions while synchronization graph nodes and dataflow dependencies are instruction-level abstractions. To convert control-dependencies to instruction level-dependencies, FAULTLESS adds an edge in the joint dataflow/control flow graph used to build the synchronization graph between each instruction in the dependent block and the branch instruction that induced the control flow.

The synchronization graph precisely captures the free variables in the expressions associated with each progenitor variable. In Figure 5.6, the execution of `cell_volume` produce a unique symbolic progenitor variable, $c_{\text{cell_volume}}$. The symbolic expression for the first argument of `pow` is $-p_1/c_{\text{cell_volume}}$, is defined in terms of this variable. This is reflected in the edge between `cell_volume` and `pow` in the synchronization graph.

5.2.6.b Building the Coupling Relation

FAULTLESS builds the coupling relation from the left and right synchronization graphs G_l and G_r . An entry in the coupling relation consists of a pair of synchronization graph nodes—an element of the product graph—which have been shown to be equivalent to each other. Only entries of the same operation type can be equivalent: a function call cannot be equivalent to a loop- ϕ instruction. Then the coupling relation C is a subset of the cartesian products of the call nodes and the ϕ nodes in the synchronization graphs:

$$C \subseteq \{(n_l, n_r) \mid \text{type}(n_l) = \text{type}(n_r) \wedge n_l \in G_l \wedge n_r \in G_r\} \quad (5.5)$$

Naïvely, FAULTLESS would have to check each element of the product graph and determine if the pair of nodes is actually equivalent based on the logged execution data. But enumerating

all combinations is expensive, and as discussed previously, performing such equivalence checking often requires a smaller partial coupling relation over the free variables for the expressions involved.

Instead, FAULTLESS uses the edges in G_l and G_r to structure the building process. The coupling relation is initially empty. The algorithm begins with *independent nodes*: those associated with no free variables or whose free variables consist of only parameters. In Figure 5.6, the `cell_volume` and `func7` nodes are *independent nodes*. (By FAULTLESS’ induction system, discussed below, the ϕ_i instructions are actually independent nodes as well). For each pair (n_l, n_r) of same-operation, independent nodes, FAULTLESS proposes the conjecture that the left and right elements of the pair are equivalent, say that `cell_volume` = `func7`. If a conjecture can be proven, FAULTLESS adds the corresponding progenitor variables to the coupling relation and adds each pair of same-operation direct dependent nodes of n_l and n_r as conjectures if no such conjecture already exists. In Figure 5.6, if `cell_volume` is proven equivalent to `func7`, then $(c_{\text{cell_volume}}, c_{\text{func7}})$ will be added to the coupling relation and `pow` and `func3` will be added as a conjecture. When each free variable in the left node of a conjecture is related to one in the right node and vice versa, FAULTLESS will attempt to prove the conjecture. If it can’t but additional relational information between the nodes’ free variables is later proven, then FAULTLESS will attempt to re-prove that conjecture. FAULTLESS uses a worklist algorithm similar to the one used in `codealign` [34] to perform this process.

Proving a Conjecture. To prove a conjecture that two function calls are equivalent to each other, FAULTLESS checks the conditions defined in the necessary equivalence conditions (Section 5.2.1) under the coupling relation. This includes that checking that each pair of arguments to the function calls is positionally equivalent, that heap space memory reachable from each addressable pair of arguments is equivalent, and that the calls occur under the same control-flow conditions. For the control flow check, FAULTLESS uses only the components of the path condition that are transitive control dependencies of the function call because only those components actually influence the control flow conditions under which a given instruction executes.

Loop- ϕ instructions are not described in the necessary equivalence conditions because they are internal features of a function that do not, by themselves, produce externally observable behavior. To prove two loop- ϕ nodes equivalent, the “base case” arguments from outside the loop must be equivalent, the “recursive case” arguments from inside the loop must be equivalent, and the path conditions required to reach any back edge from the start of the loop must be equivalent.

Dependency Cycles and Induction. Loops can create cycles of dependencies in the synchronization graph. Self-loops are common: in a basic `for` (`int i = 0; i < n; ++i`) loop like the one in Figure 5.6, the recursive case for the ϕ_i node is $\phi_i + 1$; thus, there is a dataflow dependency between the ϕ_i instruction and itself. In general, all recursive case arguments of a ϕ instruction must involve some kind of cycle. Cycles mean the coupling relation can’t contain the necessary variable relationships when attempting to prove the recursive arguments equivalent. To solve this, FAULTLESS uses induction to break cycles of dependencies, also mirroring `codealign` [34]. For the base case of the induction, FAULTLESS will attempt to show that the base case arguments of two ϕ nodes are equivalent. If it can, it makes the inductive hypothesis—if the values are equivalent at the start of the loop iteration, then the values at the end of the loop iteration are

equivalent. Concretely, FAULTLESS provisionally adds an equivalence relationship between the ϕ variables to the coupling relation and then re-visits the node to prove the recursive cases and path conditions equivalent with the updated coupling relation when it has finished proving relationships inside the loop. If the inductive hypothesis cannot be proven, FAULTLESS revokes the inductive assumption and anything from inside the loop proven based on the unprovable assumption. For instance, if the ϕ_i instructions in the original and decompiled code in Figure 5.6 cannot be proven equivalent, then that assumption is revoked and the lemma `real_prod == func4` is also revoked if it exists. To maintain soundness, FAULTLESS only allows elements of the coupling relation that are transitive dependencies of a given conjecture for use in the proof of that conjecture; that way, if one of those dependencies was a bad inductive assumption, FAULTLESS knows to revoke it.

5.2.6.c Heaplet equivalence

Several of the necessary equivalence conditions require that memory reachable from a certain address e.g. (function arguments, the return value) is equivalent. Determining whether or not this is the case is the job of the heaplet equivalence checker. In other words, the heaplet equivalence checker implements support for McCarthy’s extensionality axiom for arrays. The heaplet equivalence checker takes as input a base address from the left function and one from the right function, as well as the heaps for each of these functions and a coupling relation.

Under the extensionality axiom, two arrays—or in our case, Address Spaces—are equivalent if they are equivalent at all indices. To accomplish this, we read a value of the type of the base address, t , at an *arbitrary* index ϵ . Concretely, we run the query `heap.read(sizeof( $t$ ) ·  $\epsilon$ ,  $t$ )`. Note that this implements the extensionality axiom with respect to a type as opposed to treating every memory cell as a collection of untyped bits. Because FAULTLESS offers the ability to represent integers as “math” integers rather than bitvectors, (see Table 5.1) and floats as real numbers rather than IEEE-754 floating point numbers, this distinction is required to perform heaplet equivalence checking under all primitive type modes. Math integers can’t be represented as untyped bits.

The reads are done for both the left and right base addresses on their respective heaps, and the result is checked for equivalence subject to the coupling relation.

Memory Formatting for Decompiled Code. One of the key challenges with decompiled code is the lack of user-defined types, especially `structs`. In the validation task, it is often the case that the base address is a pointer to a structure while in the decompiled code it is a pointer to a primitive type, usually a size-based placeholder type (see Section 5.2.4.a). The resulting values read will be trivially nonequivalent because one is a composite data type and the other is a primitive data type. To prevent this, FAULTLESS provides the option to allow the decompiled code to be read with the original code’s composite type.

5.2.6.d Global Assumptions

Global variables represent inputs and outputs for a function. Unlike parameters, however, which we can reliably relate to each other positionally on the parameter list, or return values, which are trivial to relate because a function can only return one value, it is not immediately clear how global variables relate to each other across the left and right functions, especially when both the

left and right functions have multiple global variables. And unlike ϕ or callee return values, globals cannot be aligned by the way that they are computed from function inputs, because globals themselves are function inputs.

Instead, FAULTLESS determines that two globals are the same if they are used in the same context. When two symbolic expressions are checked for equivalence in the construction of the coupling relation and the expressions are nonequivalent, FAULTLESS checks to see if there is one global variable involved in each of the expressions. If so, then FAULTLESS checks to see if equating those two global variables results in the expression being equivalent. If it does, then FAULTLESS assumes that those variables are equivalent and records this assumption.

While it's difficult to tell which global variables correspond to each other, it is sometimes easy to tell which globals do *not* correspond to each other by looking at the values stored in the globals when the function terminates: globals holding different values are definitely different variables. FAULTLESS rejects any assumed equivalence relationship between globals that store different values.

5.2.6.e Checking Equivalence Conditions

In building the coupling relation, FAULTLESS has already started to satisfy the necessary equivalence conditions (Section 5.2.1) by showing certain function calls in the left function equivalent to those in the right function. But there are other necessary equivalence conditions that must be satisfied to declare the left and right function equivalent.

For callees, FAULTLESS must show that *each* function call in the left function is equivalent to at least one in the right function.

Using the entire coupling relation, FAULTLESS attempts to prove the return values equivalent. If both return values are addressable, it attempts to prove that the heaplets reachable from the left and right base address are equivalent. If one is addressable and the other is not, FAULTLESS returns that the left and right function are not equivalent. FAULTLESS also checks that memory reachable from each input parameter in the left function is equivalent to the memory reachable from the corresponding input parameter in the right function using the heaplet equivalence checker.

Consistency Checking. Finally, FAULTLESS must ensure that function and global variable names are consistent across the left and right function; that is, that the function and global variable names form a bijective mapping. Each of these name types is checked for consistency separately.

To do this, FAULTLESS sets up a constraint satisfaction problem. Each name is assigned a unique integer variable: l_i for the left function and r_i for the right function. Within a given function, distinct names are constrained to have different integer variables: $l_i \neq l_j$. The cross-functional constraints are somewhat more complicated. Let l_i be the name variable of a call in the left function. Then let R be the set of name variables in the right function for calls that have been shown equivalent in the coupling relation (for function names) or assumed equivalent (for global variable names). Then FAULTLESS adds the constraint $\bigvee_{r \in R} l_i = r$. FAULTLESS also adds its mirror image on the right.

The generalized-disjunctive form of the cross-functional constraints is important to handle the case when there are multiple possible bijective mappings amongst the names. For instance,

consider decompiled code containing the calls `func1(0)` and `func2(0)` which a neural decompiler might rename to `foo(0)` and `bar(0)`. Here, by arguments, these are all equivalent. There are multiple possible bijective mappings: `func1 ↔ foo` and `func2 ↔ bar` or `func1 ↔ bar` and `func2 ↔ foo`. If we forced every name variable in the left function to be equivalent to the name variable on the right, the constraint problem would be unsatisfiable:

$$l_{\text{func1}} = r_{\text{foo}} \wedge l_{\text{func2}} = r_{\text{foo}} \wedge l_{\text{func1}} \neq l_{\text{func2}}.$$

The disjunctive constraint prevents this issue.

5.3 Evaluation

We ask three research questions to empirically characterize FAULTLESS’ capabilities.

Because program equivalence is undecidable, any program equivalence technique must be sound, complete, or neither, but not both. Faultless is designed to be sound to foster trust between the reverse engineer and the neural decompiler by ensuring that there are no semantic mistakes in the neural decompiler’s prediction. A consequence of this design decision is that it is necessarily incomplete: there are equivalent pairs of code which the technique cannot prove equivalent. Therefore, we ask **RQ1**: *How complete is FAULTLESS?* That is, how often can FAULTLESS prove equivalent pairs of functions equivalent?

There are some known soundness limitations in its current implementation (see Section 5.5.1). Therefore, we also ask **RQ2**: *How frequently do FAULTLESS’ soundness limitations cause and unsound equivalence judgment?*

Finally, on the validation task, FAULTLESS is intended to be used live by reverse engineers as part of their tool suite. That means runtime performance is especially important, so we ask **RQ3**: *What are the runtime performance characteristics of FAULTLESS?*

We answer these questions with data from two experiments using two complementary oracles. We observe that the original code and deterministically decompiled code for the same function are equivalent by construction, but significantly syntactically different (excepting some decompiler unsoundness). In the first experiment, we use this “deterministic transformation” oracle. However, because the pairs of functions are all equivalent, this doesn’t provide any pairs of nonequivalent functions for answering RQ2. Instead, in the second experiment, we use unit tests as an oracle. They’re unsound (there could be some differentiating behavior between the two functions that the tests do not exercise) but complete. That means that if the unit tests return that two functions are *not* equivalent, they aren’t; the test case serves as a counterexample.

5.3.1 Experimental Design

We conduct two experiments, one using each of the two complementary oracles. In both experiments, we use codealign’s [34] dependency-based equivalence as a baseline. We are not aware of other techniques designed to check the equivalence of C functions in isolation without the rest of the codebase, especially for decompiled and predicted code—a key motivator for the existence of FAULTLESS in the first place.

5.3.1.a Experiment 1: Deterministic Transformation Oracle

The deterministic transformation oracle uses compiler and decompiler transformations to provide equivalent pairs of code. In Chapter 4, we constructed `REALTYPE`, which already contains these pairs, there as labeled training data for a machine learning model. We reused `REALTYPE` for this experiment, though we rebuilt it for this experiments to include definitions of types referenced in the type descriptors of the function that are not part of a variable declaration. (IDioms only predicted variables’ types so those type definitions were unnecessary.)

We use the test set in the experiments. We filter from the test set examples that include unsupported features, including `gotos`, `unions`, `enums`, the variadic parameter API, inline assembly, compiler extensions, SIMD vector types and `#pragma` directives that were not handled in the preprocessor run when constructing the dataset. `FAULTLESS` does not currently support Hex-Rays’ partial-object macros (like `LODWORD`). Also unsupported is the subtraction of two pointers, a rarely used feature of C, which is only defined if those pointers point to the same array object; under context independence, it usually cannot be determined if two pointers do actually point to the same object. When an array type is declared in a header and its length is defined not as integer but as an arbitrary C expression, the `REALTYPE` dataset construction script provides a sentinel array length of `-1`. This is of course an invalid length so `FAULTLESS` filters these examples out. Finally, sometimes the Hex-Rays decompiler will recognize and define variables of standard-library types like `jmp_buf` and `__compar_fn_t`. We provided a header of many standard library types (all those that show up in the decompiled code in the validation set), but there are some types in the test set that aren’t in the validation set. After this filtering, the rebuilt test set’s 2229 examples were reduced to 1565.

In the experiment, we run `faultless` on each original/decompiled pair and report whether `FAULTLESS` reports the functions equivalent, nonequivalent, or crashes. We also record the time it takes to perform the equivalence checking. We provide the definitions of UDTs used in the functions along with the original code.

We also perform some normalization modifications prior to calling `FAULTLESS` on each example pair. The compiler or decompiler sometimes replaces `printf` with `puts`, `putchar`, `fputc`, or `fwrite`, adjusting the arguments to match the function. When these functions do not appear in the original code but do appear in the decompiled code, we normalize the functions in the decompiled code back to `printf`. Additionally, we normalize decompiler fidelity issue C4.c. from the taxonomy in Chapter 3, which refers to the pattern where the decompiled code includes an extra address-of `&` around global variables in certain circumstances, like function calls and return values. If it is observed in the decompiled code, the `&` is stripped out.

In general, we try to use math-int ($\mathbb{Z}$) representations of integers rather than bitvectors, if possible. Math-int modeling allows `FAULTLESS` to abstract away issues with imprecise primitive type recovery, putting the emphasis on showing the algorithms equivalent. However, bitwise operations like `>>` and `|` are not defined on math integers, so we switch to bitvectors if any such operations occur in either function.

Deterministic decompilers are known to be unsound, and indeed in Section 3 we found cases where the decompiled code was not equivalent to the original. Many of these instances followed systematic patterns, which `FAULTLESS` is designed to handle via the options in Table 5.1. However, some instances of unsoundness do not fit any such pattern; the decompiled code is truly just

Table 5.2: Experiment 1 Results: Completeness

		Correct		Incorrect		Crash	
O0	FAULTLESS	805	(51.4%)	495	(31.6%)	265	(16.9%)
	Dependency-Based Equivalence	69	(4.4%)	1273	(81.3%)	223	(14.2%)
	Dependency-Based Equiv. (Consistency)	109	(7.0%)	1233	(78.8%)	223	(14.2%)
O1	FAULTLESS	664	(45.1%)	491	(33.4%)	316	(21.5%)
	Dependency-Based Equivalence	65	(4.4%)	1199	(81.5%)	207	(14.1%)
	Dependency-Based Equiv. (Consistency)	85	(5.8%)	1179	(80.1%)	207	(14.1%)
O2	FAULTLESS	599	(43.1%)	515	(37.1%)	276	(19.9%)
	Dependency-Based Equivalence	69	(5.0%)	1094	(78.7%)	227	(16.3%)
	Dependency-Based Equiv. (Consistency)	88	(6.3%)	1075	(77.3%)	227	(16.3%)
O3	FAULTLESS	581	(44.7%)	491	(37.7%)	229	(17.6%)
	Dependency-Based Equivalence	70	(5.4%)	1009	(77.6%)	222	(17.1%)
	Dependency-Based Equiv. (Consistency)	86	(6.6%)	993	(76.3%)	222	(17.1%)

Results of the completeness experiment. All pairs of examples are equivalent by construction; scores reflect how many pairs each technique was able to prove equivalent (correct), unable to prove equivalent (incorrect), or which crashed on the input (crash). Codealign [34] is used for dependency-based equivalence.

incorrect. This somewhat undermines the reliability of the deterministic transformation oracle. To account for this, we manually analyze the results for a random sample of the predictions.

5.3.1.b Experiment 2: Unit Test Oracle

For the unit test oracle, we use EXEBENCH [4], a dataset of C functions mined from GitHub which we also used in evaluating IDIOMS in Chapter 4. Its test and validation sets have unit tests bundled with each example. EXEBENCH provides, then, the unit test oracle itself as well as the original code. For the functions to compare with the original code, we use IDIOMS’ predictions (particularly those from the best performing model, codegemma-7b). Code predicted by a neural decompiler is exactly the sort of thing that FAULTLESS is designed to handle. Conveniently, codegemma-7b IDIOMS scores around 50% accuracy on EXEBENCH (Table 4.4d) so there is an approximately equal number of both equivalent pairs and nonequivalent pairs of code.

EXEBENCH is inconsistent with itself on some examples; the original code functions don’t always pass the unit tests included in the dataset. We filter out these examples. We also filter out examples with unsupported features as in Experiment 1. After filtering, there are 881 examples remaining of 1741.

In this experiment, we turn on the option which requires two functions to have equivalent names (instead of just consistent names) to be considered equivalent. This better reflects the behavior of the unit test oracle, which counts differently-named functions as incorrect.

Table 5.3: Experiment 2 Results: Comparison with Unit Tests

(a) FAULTLESS				(b) Dependency-Based Equivalence			
	equivalent	not equivalent	crash		equivalent	not equivalent	crash
tests pass	525	81	168	tests pass	367	402	5
tests fail	6	70	31	tests fail	1	105	1
Agreement: 87.243%				Agreement: 53.942%			

Results of Experiment 2, comparing FAULTLESS and codealign with unit tests, including confusion matrices and agreement scores. The agreement scores are the sum of diagonal, where both unit tests and the equivalence technique agree that the code is equivalent or not, divided by the sum of all four noncrash cells of the matrix. Function-name consistency is disabled to better conform to the unit test oracle. The bottom left cell represents instances of unsoundness.

5.3.2 Results

The results are shown in Table 5.2 for experiment 1 and Table 5.3 for experiment 2. The runtime performance results are in Figure 5.7.

5.3.2.a RQ1: How complete is FAULTLESS?

FAULTLESS is able to prove equivalent above 50% of the equivalent original/decompiled code pairs equivalent at O0, and around 43–45% at the other optimization levels as shown in Table 5.2. FAULTLESS greatly outperforms codealign [34] in this experiment, which simulates performing the validation task on a perfect neural decompiler, because dependency-based equivalence is too brittle to handle features of decompiled code like desugared field-access operations. On EXEBENCH, (Table 5.3) FAULTLESS agrees with unit tests 87.2% of the time, substantially closing the completeness gap between sound codealign and unsound unit tests, which are an over approximation of equivalence.

FAULTLESS has a higher success rate in Experiment 2 on EXEBENCH than in Experiment 1 on REALTYPE because REALTYPE is a more challenging dataset containing the full spectrum of C complexity. We sampled a set of 30 original/decompiled function pairs judged nonequivalent by FAULTLESS and manually analyzed the reasons for failures. Many of these—17, over half—are fairly mundane and unrelated to FAULTLESS’ novel modeling and equivalence checking contributions. Five are FAULTLESS compiler bugs, three are `printf` canonicalization bugs, seven are dataset generation issues, and two contain static variables, which FAULTLESS doesn’t currently support. This suggests that with additional implementation effort, FAULTLESS has room to improve its performance.

Four of the 30 cases were from oracle failures, where the decompiler is *not* semantics preserving, even when accounting for the systematic issues identified in Chapter 3 with the options in Table 5.1. FAULTLESS correctly rejects these as not equivalent, whereas the experiment presumes them to be equivalent. While this was not the motivation behind the design of FAULTLESS, it is fairly effective as a decompiler-testing technique like D-Helix [115], helping to locate bugs

in decompiled code.

The final nine cases are fundamental completeness limitations in FAULTLESS. FAULTLESS must necessarily be incomplete. Three of these examples highlight the limitations of operating in a context-independent environment—while the decompiled code is not technically incorrect, FAULTLESS does not have enough information to soundly show the functions equivalent. FAULTLESS’ completeness limitations are discussed in detail in Section 5.5.2.

The crash rate of around 17% reported in Table 5.2 is relatively high. A minority of these crashes, 36 in total, are due to limitations in the addressability model: attempting to access an address for which the base address cannot be uniquely determined. The majority of these cases occur when there are base addresses from different paths encapsulated in an if-then-else expression. The fix is conceptually simple: access both base addresses subject to the additional read/write condition of the if-condition (for the then branch) or its negation (for the else branch). Other crashes are due to corner cases in compiling and modeling the full breadth of the complex C programming language.

Codealign is designed for the evaluation task while this experiment simulates the validation task on perfect predictions; as a result, its crash rate is around five times higher than reported in the original paper due to difficulties with processing decompiled code.

Answer to **RQ1**: Faultless is able to prove equivalent more than 50% of equivalent pairs of decompiled/original functions in unoptimized code and more than 43% in optimized code.

5.3.2.b RQ2: How frequently do FAULTLESS’ soundness limitations cause and unsound equivalence judgment?

Table 5.3a shows that FAULTLESS’ soundness limitations are relatively uncommon, at least on EXEBENCH. Virtually all of the disagreement is in the “tests pass”/“FAULTLESS reports nonequivalent” category on the top right of the confusion matrix, which represents incompleteness. Of the six instances of unsoundness in the bottom left corner of the confusion matrix only five are real: one is due to undefined behavior in the EXEBENCH-provided harness and test case; the actual generated functions are equivalent. We discuss soundness limitations further in Section 5.5.1; three of the instances of unsoundness fall into categories discussed there.

Answer to **RQ2**: FAULTLESS’ soundness limitations are exploited less than 1% of the time on EXEBENCH.

5.3.2.c RQ3: What are the runtime performance characteristics of FAULTLESS?

As shown in Figure 5.7, FAULTLESS is generally very fast. FAULTLESS can produce an equivalence judgement in under one second for 90.1% of examples in REALTYPE and in under two seconds for 96.7% of examples.

SMT solver queries dominate the runtime of FAULTLESS, so it follows that functions which produce more or larger solver queries constitute the remaining 3.3% of high runtimes. There are

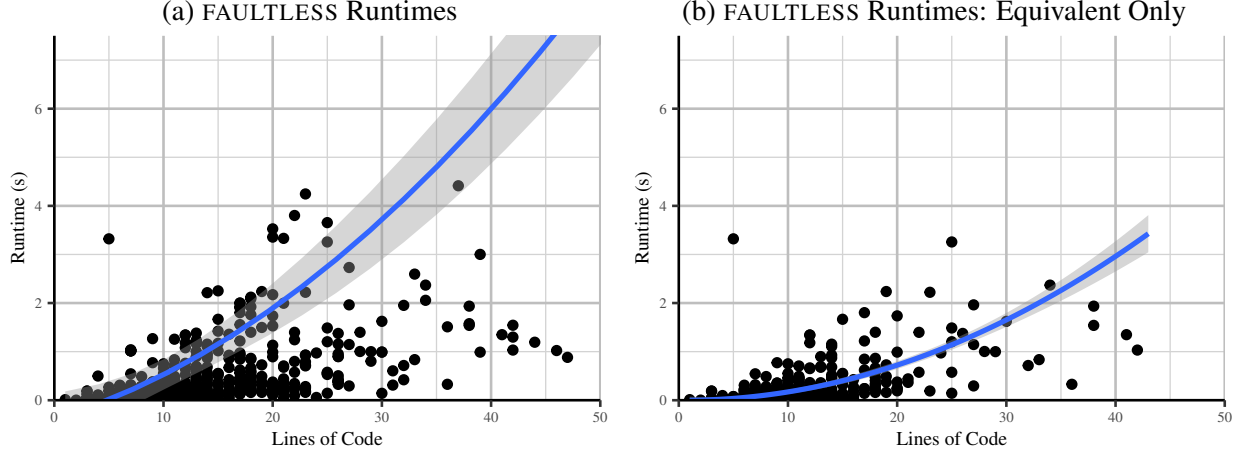


Figure 5.7: Runtime efficiency for FAULTLESS on Experiment 1 with quadratic regression trend lines. Crash runtimes are excluded because crashes usually happen before the expensive proof engine stage. Figure 5.7a includes pairs of functions that FAULTLESS reported nonequivalent. The trend line in Figure 5.7a is pulled up by a few outliers with runtimes greater than 7.5 seconds.

two main culprits. The first is large synchronization graphs with many synchronization nodes—function calls or loop- ϕ instructions—with equivalent dependencies. If there are n calls in each function and each has no dependencies, the proof engine will perform n^2 call equivalence checks, which is expensive. In contrast, if the i th call was a dependency of the $i + 1$ st call, then the proof engine will have to perform only n call equivalence checks. The dependency structure can substantially influence FAULTLESS’ runtime. Most dependency structures produce relatively small sets of synchronization nodes with the same dependencies, but exceptions exist.

The second is large fixed size arrays (e.g. stack allocated or as a struct field) in circumstances where the entire array needs to be read from memory, such as when a struct with a large fixed-size array member is passed by value to a callee. Reads for full composite data types are decomposed into reads of each of their members. That means n reads for an array of length n . Then suppose there are m writes relative to the base address for that array. That means that in the worst case, there are $2nm$ solver queries, an overlap query (Equation 5.1) and a coverage query (Equation 5.3) for each Write node, plus any additional queries for Join nodes. This quickly becomes expensive.

Answer to **RQ3**: FAULTLESS is very fast, producing equivalence judgements in under one second for 90.1% of functions and in under two seconds for 96.7% of functions on REALTYPE.

5.4 Utility

FAULTLESS’ ability to compared decompiled, predicted, and original code allow it to fulfill multiple roles in both research and practice, assisting in both evaluation and validation tasks. In this section, we show FAULTLESS’ utility on these tasks.

Recently, large language models have anecdotally become increasingly effective at performing neural decompilation. But it has been difficult to measure how good they have become. FAULTLESS offers the ability to quickly and effectively benchmark these models’ correctness. In this section, we perform a series of experiments on a frontier model, GPT-5.5. To begin, we ask **RQ4**: *How effective is a modern frontier model on neural decompilation?*

In Chapter 4 we show that for smaller, fine-tuned or adapted models, adding neighboring context leads to an increase in model performance. The motivation for adding neighboring context is independent of model size or architecture, though; it’s a question of information: there is sometimes not enough information within a compiled function to correctly decompile that function, especially when it comes to user-defined types. While FAULTLESS does not evaluate UDT predictions directly, they are factored into the equivalence judgements by the offsets to which their fields map. We use FAULTLESS to help answer the research question **RQ5**: *To what degree does neighboring context improve the correctness of a frontier model’s predictions?*

One of the key challenges that FAULTLESS faces is optimizations which transform loop structures, like loop unrolling or loop tiling. By requiring ϕ instructions to be equivalent to prove values and calls affected by loops equivalent, FAULTLESS effectively requires one iteration of one loop in the left function to correspond to one in the right function. In traditional neural decompilation, the model is trained to predict the original code, thereby *undoing* the optimizations. This is problematic for validation because FAULTLESS won’t be able to prove even a perfectly correct prediction equivalent. However, with large language models, one mitigation is to ask the model to preserve loop structure while otherwise making the code match the likely original source code as closely as possible, thereby offering the benefit of cleaned-up neurally decompiled code while also gaining the correctness assurances that FAULTLESS provides. We use FAULTLESS to help answer the research question **RQ6**: *To what degree does preserving loop structure increase FAULTLESS’ validation effectiveness?*

FAULTLESS’ validation equivalence judgements can be used by reverse engineers to reduce the risk of receiving incorrect code from a neural decompiler. But these scores need not only be used by reverse engineers: FAULTLESS’ equivalence judgement and textual description of the reason for nonequivalence can also be used to provide feedback to the model to correct its prediction. Thus, we ask the research question **RQ7**: *How effective is FAULTLESS at improving neural decompilers’ predictions through re-prompting?*

5.4.1 Experiment Methodology

To perform the experiments, we use REALTYPE and prompt the frontier model GPT-5.5. We use the same filtering as in Section 5.3.1.a. RQ6 requires performance data on optimized code; to save API costs, we chose one optimization level, O2, because it is the most difficult for FAULTLESS in Table 5.2. To answer the four research questions in this section, we perform seven different experiment runs with the settings:

- O0, function-level context
- O0, neighbors-level context
- O0, neighbors-level context with re-prompting
- O2, function-level context

Table 5.4: GPT-5-5 Neural Decompilation Performance

(a) O0 Results

Context Type and Experiment Settings	Evaluation Accuracy	Validation Accuracy
functions	56.5%	78.6%
neighbors	55.9%	73.5%
neighbors (reprompting)	58.2%	92.1%

(b) O2 Results

Context Type and Experiment Settings	Evaluation Accuracy	Validation Accuracy
functions	46.2%	74.1%
neighbors	48.7%	72.8%
neighbors (loop preservation)	48.3%	78.1%
neighbors (loop preservation & reprompting)	48.0%	92.4%

- O2, neighbors-level context
- O2, neighbors-level context with loop preservation prompting
- O2, neighbors-level context with loop preservation prompting and re-prompting

For the loop-preservation prompting experiments, we append the following text to the prompt:

Preserve the loop structure shown in the decompiled target. Do not attempt to undo loop-transforming compiler optimizations that may have changed the number, nesting, ordering, or boundaries of loops. Otherwise, make the code match the likely original source as closely as possible.

For the re-prompting experiments, we re-prompt up to three times before giving up; re-prompting only occurs if FAULTLESS judges the prediction to be incorrect.

In these experiments, we have all three source-level representations from Figure 5.1. Therefore, it’s possible to judge the models on how well they would perform in validation, how well the prediction matches the decompiled code, and evaluation, how well it matches the original code.

When reporting scores, we exclude FAULTLESS crashes. In this section, we are not evaluating FAULTLESS (as we were in Section 5.3), rather, we are using FAULTLESS to evaluate (and validate) a large language model. FAULTLESS crashes reflect bugs in or limitations of FAULTLESS and do not provide any useful information on the correctness of the model’s prediction.

5.4.2 Results

The experiment results are shown in Table 5.4.

5.4.2.a RQ4: Evaluation of A Frontier Model

The frontier model GPT-5.5 performs well at neural decompilation, scoring above 55% accuracy on evaluation and 73% on validation on unoptimized code and at least 46% on evaluation and 72% on validation at O2. The fact that the validation scores are higher may be surprising, because neural decompilers are supposed to predict the original code and clean up decompiler artifacts in the decompiled code. While GPT-5.5 does certainly clean up decompiler artifacts and the code looks reasonable, the structure of the code it generates more closely matches the decompiled code than the original code. For instance, despite disabling inlining via command line arguments, some functions are still inlined in REALTYPE. (In at least some instances, we observed this was due to an `always-inline` attribute present in the original code.) To succeed in evaluation, the model would have to recognize which expressions or statements correspond to an inlined function, extract the function, and choose the correct argument order. Meanwhile, to succeed at validation, the function need only propagate the body of the inlined function in the decompiled code to the predicted code like any other statement or expression—a much easier task!

The scores here are also an underestimate of GPT-5.5’s performance because FAULTLESS is incomplete. While the model does make mistakes, the majority of FAULTLESS’ nonequivalence judgements we observed are due to its incompleteness; the predictions were actually correct.

Answer to **RQ4**: Frontier models are very proficient at recovering semantics in neural decompilation, scoring above 70% validation accuracy on FAULTLESS.

5.4.2.b RQ5: Function-Level vs. Neighboring Context for a Frontier Model

Central to IDIOMS design and the driving force behind the addition of neighboring context is the scattered evidence problem for UDTs. In this experiment, we see how neighboring context impacts the much larger GPT-5.5.

Surprisingly, the addition of neighboring context has little effect, or actually *decreases* overall performance: validation accuracy decreases from 78.5% to 73.5% at O0 and 74.1% to 72.8% at O2. Some functions are entirely self-contained and need no additional context, and irrelevant context has been shown to distract large language models [91]. The scattered evidence problem means there are fundamental limits to the UDT definitions that the LLM can recover, and incorrect offsets, will, if they are related to a function behavior that FAULTLESS checks, result in nonequivalence. However, we frequently observed the LLM, unbidden, inserting padding fields of the appropriate size to make the offsets correct, preventing this from being an issue. Additionally, the LLM has memorized the definitions of all standard-library `structs`, making it easy to bypass the scattered evidence problem in these instances and predict the entire `struct` correctly.

Answer to **RQ5**: Neighboring context does not help improve frontier models’ FAULTLESS scores, though it is necessary for full UDT definitions in some cases.

5.4.2.c RQ6: The Impact of Loop Structure Preservation

A key limitation of FAULTLESS is that it requires on loop in the left function to correspond to one in the right and vice versa. Prompting the model to preserve loop structure helps to mitigate this limitation.

In particular, we see that adding the loop preservation prompt increases validation scores modestly from 72.1% to 78.1%. The baseline scores are already fairly high. Not all loops are impacted by loop optimizations and many functions are relatively small (the median length for REALTYPE is 14.2 lines of code) and contain no or one loop, so the probability of having a loop with unaffected by structure-altering optimizations is reasonably high. Combined with GPT-5.5’s propensity to conform to the decompiled code, this means that it can score relatively well without the additional prodding in the prompt. However, for functions with loops that are impacted, structure preservation can help.

Evaluation scores are largely unaffected but decrease slightly; this is in line with expectations because the preservation of optimized loop structures makes the code less like the unoptimized original source code and therefore less likely to be judged equal to the original code.

Answer to **RQ6**: Loop preservation prompting provides a modest increase in validation accuracy from an already high baseline, raising it from 72.1% to 78.1%.

5.4.2.d RQ7: Reprompting Based on FAULTLESS Feedback

FAULTLESS can provide quick equivalence judgements along with error messages indicating the reason for nonequivalence (i.e. differing return values, a specific callee in one function doesn’t match one in the other). These properties make it suitable for re-prompting the LLM if FAULTLESS is unable to prove the LLM’s prediction correct. Table 5.4 shows that re-prompting offers significant validation performance gains over the baselines, jumping from 73.5% to 92.1% at O0 and from 78.1% to 92.4% at O2.

FAULTLESS does catch and help the model correct genuine errors. Figure 5.8 shows an example of how FAULTLESS corrects GPT-5.5 about a missing function call. However, GPT-5.5 often overreacts to FAULTLESS’ feedback by aggressively matching the decompiled code. For instance, it may revert a struct field access operation back to pointer arithmetic, e.g. changing `root->left` to `*(long *) (a1 + 8)` in the second prediction. In the most extreme instances, the new prediction is nearly identical to the decompiled code. The especially high (90%+) scores are partially a result of this kind of reward hacking. The model may also change variable or function names back to the corresponding names in the decompiled code. It is surprising that even a single reprompting instance can cause GPT-5.5 to completely disregard the detailed original prompt to produce code like the expected original.

Neural decompilation is ultimately a complex, multiobjective problem. As discussed in Section 4.3.2, a neural decompiler has two main goals: (1) correctly representing the functionality in the binary or decompiled code, and (2) improving the code by re-introducing higher level abstractions, adding names, removing decompiler artifacts, and generally making the code easier to read and more similar to the original code. FAULTLESS offers significant improvement in the state of the art for checking (1), but does not make any attempt to evaluate (2). Indeed,

```

1 void *hash_find(struct hash *h, void *item)
2 {
3     int index = hash_find_index(h, item);
4     if (index == -1)
5         return ((void *)0);
6     void *out = gap_get(h->data, index);
7     return out;
8 }

```

(a) The Original Code

```

1 __int64 func5(__int64 a1, __int64 a2) {
2     __int64 result; // rax
3     int v3;         // [rsp+1Ch] [rbp-4h]
4
5     v3 = func4(a1, a2);
6     if (v3 == -1)
7         result = 0LL;
8     else
9         result = func1(*(_QWORD *) (a1 + 8), v3);
10    return result;
11 }

```

(b) The Decompiled Code

```

1 void *hash_table_lookup(struct hash_table *
2     table, void *key) {
3     int index;
4     index = hash_table_find_index(table, key);
5     if (index == -1)
6         return 0;
7
8     return table->entries[index];
9 }

```

(c) GPT 5.5's Initial Prediction

```

1 void *hash_table_lookup(struct hash_table *
2     table, void *key) {
3     int index = hash_table_find_index(table,
4         key);
5     if (index == -1)
6         return 0;
7     return hash_table_get_entry(table->entries
8         , index);
9 }

```

(d) GPT 5.5's Corrected Prediction

Figure 5.8: An example of a successful instance of re-prompting. GPT 5.5's initial prediction based on Figure 5.8b is shown in Figure 5.8c. This prediction is not correct; it's missing the call to `gap_get`. (`gap_get` is not simply `h->data[index]`; it performs additional checks.) FAULTLESS reports that the call is missing, and when prompted again the call is added as shown in Figure 5.8d. FAULTLESS then judges the prediction correct.

when a validation “nonequivalent” judgement is a result of a FAULTLESS’ incompleteness and the prediction is actually correct, reprompting can actually degrade the final prediction by undoing some of the improvements under the model’s aggressive reward-hacking behavior. This suggests that FAULTLESS should be part of a larger suite of validation metrics which help push the model towards producing code that is both correct and higher quality.

Especially in combination with suitable improvement metrics, FAULTLESS could be used as part of a reinforcement-learning setup to help improve neural decompiler predictions.

Answer to **RQ7**: Re-prompting is very effective at improving FAULTLESS validation scores and can help correct mispredictions, but models may overfit to the feedback and degrade the quality of the prediction.

5.5 Limitations

Program equivalence is an extremely challenging task, and any program equivalence technique must navigate its inherent undecidability by making tradeoffs between soundness, completeness, and utility to end users.

5.5.1 Soundness

While FAULTLESS is designed with soundness in mind, in its current implementation, there are several well-defined areas in which it can report equivalence for nonequivalent functions. None of these soundness limitations are fundamental; rather, they represent features that were deprioritized because the implementation effort required to support them was disproportionate with respect to their frequency in our datasets.

5.5.1.a Heap Base Case and Inductive Disassociation

An important feature of FAULTLESS’ execution model is that ϕ symbolic variables are valid base addresses; writes relative to a ϕ variable represents a write on an arbitrary loop iteration. However, because ϕ variables are distinct from the corresponding base-case arguments, a heap space memory check from the base case will not incorporate the ϕ instruction’s memory. Therefore, FAULTLESS reports the two functions in Figure 5.9 equivalent, seeing no writes relative to the n parameters.

In its simplest form, the solution is simple: when checking that memory accessible from two arguments is equivalent, the corresponding ϕ instructions’ heaplets should be checked for equivalence as well. The heap already tracks the relationship between base cases and ϕ variables in the heap replay buffer (Section 5.2.5.e), making it easy to know which ϕ variables’ heaplets to check. This becomes more expensive when there are multiple ϕ instructions’ heaplets to check.

Dissociation between the base cases’ and ϕ -variables’ heaps is the most common soundness issue detected in Experiment 2 (Table 5.3a) accounting for four of the five real soundness issues.

<pre> 1 typedef struct node { 2 int val; 3 struct node * next; 4 } Node; 5 void initialize(Node *n) { 6 while (n) { 7 n->val = 1; 8 n = n->next; 9 } 10 } </pre>	<pre> 1 typedef struct node { 2 int val; 3 struct node * next; 4 } Node; 5 void zerolist(Node *n) { 6 while (n) { 7 n->val = 0; 8 n = n->next; 9 } 10 } </pre>
--	--

Figure 5.9: Two differing writes to memory in a loop that FAULTLESS does not detect when checking the equivalence of memory reachable from the input arguments.

<pre> 1 typedef struct node { 2 int val; 3 struct node * next; 4 } Node; 5 Node * newlist() { 6 Node * tail = malloc(sizeof(Node)); 7 tail->val = 0; 8 tail->next = NULL; 9 Node * head = malloc(sizeof(Node)); 10 head->val = 0; 11 head->next = tail; 12 return head; 13 } </pre>	<pre> 1 typedef struct node { 2 int val; 3 struct node * next; 4 } Node; 5 Node * newlist() { 6 Node * tail = malloc(sizeof(Node)); 7 tail->val = -1; 8 tail->next = NULL; 9 Node * head = malloc(sizeof(Node)); 10 head->val = 0; 11 head->next = tail; 12 return head; 13 } </pre>
--	---

Figure 5.10: A pair of functions which are equivalent at one level of indirection but not at two levels of indirection; the tail node's `val` field is initialized with a 0 on the left and -1 on the right. FAULTLESS currently reports these as equivalent.

```

1 void divloop(int *x, int y) {
2     while (*x > y) {
3         *x = *x / 2;
4     }
5 }

```

```

1 void div(int *x, int y) {
2     if (*x > y) {
3         *x = *x / 2;
4     }
5 }

```

(a) A function which mutates the same heap memory (b) A loop-free function which FAULTLESS reports equivalent to Figure 5.11a

Figure 5.11: A pair of examples which FAULTLESS currently reports equivalent but which are not equivalent. Because the value controlling the iteration is on the heap rather than the stack in `divloop`, there is no ϕ instruction for `*x`.

5.5.1.b Recursive Heap Equivalence

As part of the necessary equivalence conditions (Section 5.2.1), FAULTLESS requires that memory reachable from the function’s arguments and callee arguments to be equivalent to the corresponding arguments in the other function. However, at present FAULTLESS does not apply this requirement recursively: it does not require that memory referenced at heap locations accessible from the input arguments are equivalent across the two functions. Thus, FAULTLESS will report the functions in Figure 5.10 as equivalent.

The fix is involved but conceptually simple: for each addressable component of the memory locations read by the heaplet equivalence checker, the checker should be called recursively until a difference or an empty heaplet (no writes relative to that base address) is found.

This feature was deprioritized because programming paradigms around multi-indirection make it less of a pressing issue. Many functions which recursively process linked data structures operate one “layer” (e.g. node) of a data structure at a time. Or, alternatively, the whole data structure is processed sequentially in a loop, which results in special loop- ϕ related semantics. No soundness issues of this type were discovered in the soundness testing experiment, though the EXEBENCH dataset is light on `structs`.

5.5.1.c Heap Induction

FAULTLESS uses SSA ϕ instructions to identify variables which are mutated during the loop. ϕ variables are used to help perform induction to prove loops equivalent. This works for the vast majority of loops.

However, ϕ instructions apply to variables on the stack that are assigned to during the loop. It is also possible to mutate the same *heap* location on each iteration of a loop; Figure 5.11a shows an example. This function has no ϕ instructions of any kind because all definitions (the two in the parameter lists) dominate all uses, so FAULTLESS never performs induction. Essentially, FAULTLESS treats the loop as an if-statement as in Figure 5.11b.

To handle this case, FAULTLESS would have to perform induction on heap memory locations that are modified on multiple loop iterations. Note that most in-loop heap modifications that occur in real code, like iterating over a linked list or array while to initialize it, are not an issue because the modifications are defined in terms of ϕ instructions. Instead, heap induction is required when there is a read from and write to the heap in a loop, where neither the base address

```

1 struct node { char * s; struct node * next; };
2 void alloc_str(struct node * n, int len) {
3     n->s = my_alloc(len);
4     if (!n->s) {
5         exit(0);
6     }
7 }

1
2 void func3(_QWORD *a1, int a2) {
3     *a1 = func0(a2);
4     if (!*a1) {
5         exit(0);
6     }
7 }

```

Figure 5.12: A function and its hypothetical decompilation. They are equivalent, but FAULTLESS will judge them nonequivalent due to the indirect heap-space dependency between `my_alloc` and `exit`.

nor offset is defined in terms of a ϕ symbolic variable: if this is the case, it means the memory location accessed is constant with respect to the loop and is thus acting like a stack variable which also has a constant address with respect to the loop.

A proposed, though unimplemented, solution to this issue is to place instructions of a new type called ψ instructions at loop heads in which heap induction occurs. These should have one argument: the address of the heap location that is modified during the loop. To perform heap induction, for the base case, the values at the address must be equivalent when the loop iteration begins. Then if they have the same value at the start of the loop, they should have the same values after the loop. ψ instructions should write mediloop and postloop symbolic ψ variables to their memory addresses, just like ϕ instructions. This is related to memory SSA, which we discuss further in Section 5.5.2.a.

In the soundness experiments, only one instance of heap induction was observed. This is a rare programming pattern.

5.5.2 Completeness

Because faultless is designed with soundness in mind, it is necessarily incomplete. Some completeness limitations are due to limitations in what FAULTLESS can soundly assume in a context-independent environment. Below are several important completeness limitations. There are others besides those listed here; these are some of the most consequential.

5.5.2.a Indirect Dependencies through Memory

In the proof engine, the synchronization graph captures the dependencies between the proof obligations. Dependencies reflect membership in the free-variable sets of synchronization nodes; a synchronization node is dependent on each other node which defines a symbolic variable in the former's free-variable set. FAULTLESS traverses a joint SSA and instruction-level control dependency graph to find these relationships. However, not all dependencies are reflected in this graph.

The missing dependencies occur indirectly through the heap. Figure 5.12 shows an example. The function and its decompilation are judged as not equivalent because FAULTLESS cannot prove the `exit` calls equivalent, despite the function in theory satisfying the necessary equivalence conditions. The `exit` calls are control dependent on the `if` statement condition; to be

considered equivalent two functions must be called under the same control flow conditions. These `if` statements are in theory dataflow dependent on the values returned from `my_alloc` and `func0`. But there is no SSA edge between the `if` condition and source of those calls. Thus when FAULTLESS goes to attempt to prove the `exit` functions equivalent, it compares their control flow conditions, by asking if $(c_{\text{my_alloc}} = 0) = (c_{\text{func0}} = 0)$ is valid (always true), which it is not because FAULTLESS does not know that `cmy_alloc` actually represents the same thing as `cfunc0`.

The best solution for this problem would be an analog of SSA for the heap, which would also solve the heap-induction soundness limitation (Section 5.5.1.c). This is harder than regular SSA because def-use relations for heap space memory is implicit and alias-dependent, but the problem is well studied in the literature [22, 60, 78] and could represent a promising future direction.

5.5.2.b Byte-Level Type Reinterpretation

A key goal of FAULTLESS is to see if the *algorithm* captured by a neural decompiler’s prediction—the partial sequence of operations it performs—is correct without focusing too much on the types in which that algorithm is expressed. To this end, FAULTLESS models primitive and pointer types—those which conceptually represent a single number from the perspective of the C language—as atomic, indivisible values. Composite values, like `structs` and arrays, are represented as aggregations of these atomic values at specific offsets. This is advantageous when dealing with deterministic decompilers’ necessarily imprecise type recovery and helping to provide compiler independence, bypassing trivial nonequivalence results due to type differences, and allowing for a seamless transition between math-int mode and bitvector mode.

However, this approach has its limits and also incurs some trade offs, particularly when processing decompiled code. Decompiled code especially may abuse the memory layout of a given variable and represent multiple conceptual values in the original code as a single value. For instance, consider the following snippet from a REALTYPE example:

```
1 char query[] = "aby";
2 int res = bs(query, p, 5);
```

```
1 v2 = 7955041;
2 v3 = bs(&v2, v1, 5LL);
```

In the decompiled code, the four-byte (including the null terminator) string `"aby"` is packed into a four-byte integer variable. FAULTLESS sees the value in `query` as four distinct values while the value in `v2` as one value, so it rejects the calls to the binary search function as nonequivalent, despite the fact that they may be represented as identical byte sequences.

5.5.2.c Composite Globals in Validation

Global variables are generally difficult for FAULTLESS because there is no a priori mapping between global variables; rather, which globals in the left function correspond to those in the right must be inferred based on context. In the validation task, where the right function is produced via deterministic decompilation, this problem is compounded when the global variable is a composite variable like a struct or an array because of the ways in which deterministic decompilers recover composite global variables. In particular, different components (e.g. fields, elements) of the variable may be recovered as entirely different variables, sometimes just referencing a spe-

cific offset relative to the binary’s base address, as occurs in the following original/decompiled snippets from a function in REALTYPE:

```
1 root->i_size = (2 * sizeof(struct dir_entry));      1 qword_4044D0 = 774766593LL;
2 root->i_time = time((void *)0);                    2 qword_4044D8 = 0LL;
```

In the original C code, even if FAULTLESS had the definition of the composite type, which FAULTLESS does not assume for globals, the offsets to the fields are relative to the start of the global variable `root`, not the binary’s base address as with the `qword_X` symbols in the decompiled code. Additionally, FAULTLESS sees the components as derived symbolic variables relative to a single unique variable base address, while the three `qwords` are seen as three separate global variables. The decompiled code has no way to delineate where a composite global variable ends and another begins.

This is a context-independence and compiler-independence limitation; without knowing more about the global variables and the way they are laid out in memory, it’s difficult to prove anything equivalent here.

5.6 Conclusion

Neural decompilation is a promising technique for helping overburdened reverse engineers to more quickly reconstruct abstractions in the target program. Neural decompilation’s greatest pitfall, however, is its probabilistic nature; predictions may be incorrect. Assessing semantic correctness reduces to program equivalence. FAULTLESS is a novel program equivalence technique which is designed with neural decompilers in mind. It is designed for two tasks, evaluation and validation, the former involving comparing a neurally predicted function with a gold-standard reference original version and the latter involving comparing the prediction with deterministic decompilation. FAULTLESS is largely context independent and compiler independent, requiring no information from the functions’ calling contexts or dependencies aside from type definitions and it abstracts over most compiler-level details, modeling the C language at a higher level. FAULTLESS is substantially more capable than codealign [34], an existing program equivalence technique for neural decompilation. We show how FAULTLESS is useful in evaluating a model’s performance, comparing modeling approaches, ensuring that predictions are correct, and guiding the model to produce better predictions through re-prompting.

Chapter 6

Conclusion

In summary, this thesis contains a wholistic exploration of neural decompilation: motivating it through a rigorous analysis of the underlying problem, providing datasets and a modeling approach, as well as methods for evaluating the output of neural decompilers and validating their predictions’ correctness. This work is in service of reverse engineers, helping to reduce the difficulty of an extremely challenging task and thereby help them handle the increasingly dangerous cyber landscape.

The nature of the field has changed substantially since I began my PhD in this area six years ago. My first paper featured research on DIRE [62], a variable-name prediction model initially built by my colleagues from an LSTM and graph neural network, so unlike the ubiquitous transformer models of today. It also fit comfortably on a consumer GPU. Since then, models have exploded in size and power, with profound implications for the future of neural decompilation and reverse engineering more broadly. While the future always has surprises in store, I do expect the field to shift away from smaller, custom trained or fine-tuned models towards large remotely hosted models. Smaller custom models trained from scratch or fine-tuned have made great strides, but typically fall short of the reliability of large, powerful frontier models, which can handle optimized or even obfuscated code with ease. IDIOMS, for instance, has correctness rates on unoptimized code peaking at 50-60%. This is pretty good, but pales in comparison to the observed accuracy rates of nearly 80% in Section 5.4.2.d, with the true accuracy rates being even higher due to limitations in the evaluation methodology. Further, local models, especially the bigger and more performant ones, require some fairly expensive hardware, while API access is comparatively inexpensive: performing neural decompilation via the frontier model GPT-5.5 involved making predictions for over 10,000 functions, much of it with neighboring context, and the total cost was less than \$300: less than 3 cents per function. And there are cheaper models available for less intensive tasks; just as we saw in Figure 4.4, decreasing model size by an order of magnitude leads to much less than an order of magnitude decrease in performance. For the experiments, I wanted to evaluate the best of what was currently available, but a smaller and cheaper model may sometimes be a better choice.

That being said, there are definitely some trade offs. Some organizations may be unwilling to send sensitive data to a third-party company’s servers. Other organizations might be large enough that it is cost effective to run a medium-size model accessible from their local network. Still others may want to be able to tinker with the source code or weights of their models. In

these cases, validation approaches like FAULTLESS are especially valuable, making modest error rates admissible while still providing utility.

What does this mean for the neural decompilation research of the future? The most pressing need that I can see is better evaluation strategies, on both the correctness and code improvements fronts. FAULTLESS is an excellent step in the direction of the former, and could get better with more engineering work. That being said, context independence and comparing with decompiled code have fundamental limits. FAULTLESS’ approach of modeling C abstractly with as few lower-level details as possible also pins the success of the approach on the ability of the decompiler to propagate low-level details into the higher-level space of C, as shown in Section 5.5.2. Evaluating code improvements is also extremely difficult, but in a different way—what does high quality code look like? Measuring code quality is notoriously difficult because it’s such a difficult thing to define. Nevertheless, it is a critical step—without the ability to accurately measure performance, it’s hard to know if a given approach is successful or not. And based on the results in RQ7 5.4.2.d, effective code quality metrics may also be important for validation and re-prompting as well as for evaluation.

While I expect direct modeling work like IDIOMS to reduce in prominence, there’s still a lot left to do from the machine learning side as well. An important contribution of IDIOMS was not just the modeling tools that it introduced, but also the modeling approach, which addressed a fundamental problem (scattered evidence) in neural decompilation. However, as RQ5 5.4.2.b shows, for models not trained to handle interprocedural context in the neighboring context style, irrelevant extra context might actually undermine the model instead; a different approach to providing interprocedural context may be needed. Also unresolved in current work is making identifier names inferred across multiple model queries consistent, so that reverse engineers can consolidate all of the information that they’ve learned. More broadly, while it’s hard to compete with the sheer scale of frontier AI models, there is substantial work to be done in leveraging them, either through building proof-of-concept prototypes or directly interacting with the models. Even FAULTLESS, through re-prompting interactions, could be considered to be a part of a modeling approach.

Finally, most neural decompilation research has been confined to C because most major deterministic decompilers only output C and performing cross-lingual evaluations (i.e. comparing predicted C with Golang) is very challenging. But the flexibility of modern LLMs has opened the door to purely ML-based decompilation, allowing any target output language, in the vein of Hosseini and Dolan-Gavitt’s thesis in their 2022 paper [48]. Evaluating correctness and code improvements in languages beyond C is an attendant challenge, especially when considering language features that don’t exist in C and aren’t handled by existing techniques, like objects and smart pointers.

Of course, AI has also made reverse engineering harder; adversaries have access to it as well. Their innovations might spur the next generation of reverse engineering challenges.

Bibliography

- [1] Jd-gui 1.6.6, 2019. URL <https://github.com/java-decompiler/jd-gui>. 6
- [2] V Alfred, S Monica, Sethi Ravi, Ullman Jeffrey D, et al. *Compilers Principles, Techniques*. Pearson, 2006. 5.2.2.a
- [3] 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. 1.3, 5.2.2.a
- [4] Jordi Armengol-Estapé, Jackson Woodruff, Alexander Brauckmann, José Wesley de Souza Magalhães, and Michael F. P. O’Boyle. Exebench: an ml-scale dataset of executable c functions. In *Proceedings of the 6th ACM SIGPLAN International Symposium on Machine Programming*, MAPS 2022, page 50–59, New York, NY, USA, 2022. Association for Computing Machinery. ISBN 9781450392730. doi: 10.1145/3520312.3534867. URL <https://doi.org/10.1145/3520312.3534867>. 4.2.2, 4.1, 4.3.1, 4.3.3, 5.3.1.b
- [5] Jordi Armengol-Estapé, Jackson Woodruff, Chris Cummins, and Michael F.P. O’Boyle. Slade: A portable small language model decompiler for optimized assembly. In *2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, pages 67–80, 2024. doi: 10.1109/CGO57630.2024.10444788. 1, 2.3.1, 4.1, 4.2.1, 4.3.1, 4.3.3, 4.5
- [6] 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. 5
- [7] Sahar Badihi, Faridah Akinotcho, Yi Li, and Julia Rubin. Ardif: scaling program equivalence checking via iterative abstraction and refinement of common code. In *Proceedings of the 28th ACM joint meeting on European software engineering conference and symposium on the foundations of software engineering*, pages 13–24, 2020. 2.4
- [8] Dzmitry Bahdanau, Kyung Hyun Cho, and Yoshua Bengio. Neural machine translation by jointly learning to align and translate. In *3rd International Conference on Learning Representations, ICLR 2015*, 2015. 2.3.2
- [9] Mike Barnett, Bor-Yuh Evan Chang, Robert DeLine, Bart Jacobs, and K Rustan M Leino. Boogie: A modular reusable verifier for object-oriented programs. In *Formal Methods for Components and Objects: 4th International Symposium, FMCO 2005, Amsterdam, The Netherlands, November 1-4, 2005, Revised Lectures 4*, pages 364–387. Springer, 2006.

- [10] Gilles Barthe, Juan Manuel Crespo, and César Kunz. Relational verification using product programs. In *International Symposium on Formal Methods*, pages 200–214. Springer, 2011. 2.4
- [11] Virginia Braun and Victoria Clarke. *Thematic analysis*. American Psychological Association, 2012. 3.1, 3.2
- [12] Andrei Z Broder. On the resemblance and containment of documents. In *Proceedings. Compression and Complexity of SEQUENCES 1997 (Cat. No. 97TB100171)*, pages 21–29. IEEE, 1997. 4.2.2
- [13] 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. 3.1
- [14] Juan Caballero and Zhiqiang Lin. Type inference on executables. *ACM Comput. Surv.*, 48(4), May 2016. ISSN 0360-0300. doi: 10.1145/2896499. URL <https://doi.org/10.1145/2896499>. 2.1
- [15] Juan Caballero and Zhiqiang Lin. Type inference on executables. *ACM Computing Surveys*, 48(4), May 2016. ISSN 0360-0300. doi: 10.1145/2896499. URL <https://doi.org/10.1145/2896499>. 2.3.2, 5.2.4.b
- [16] Cristian Cadar, Daniel Dunbar, Dawson R Engler, et al. Klee: unassisted and automatic generation of high-coverage tests for complex systems programs. In *OSDI*, volume 8, pages 209–224, 2008. 5.2.3.d
- [17] Ying Cao, Ruigang Liang, Kai Chen, and Peiwei Hu. Boosting neural networks to decompile optimized binaries. In *Proceedings of the 38th Annual Computer Security Applications Conference*, pages 508–518, 2022. 1, 2.3.1, 3.1, 3.4.1, 4.2.1
- [18] Ying Cao, Runze Zhang, Ruigang Liang, and Kai Chen. Evaluating the effectiveness of decompilers. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 491–502, 2024. 5.1
- [19] Casey Casalnuovo, Earl T. Barr, Santanu Kumar Dash, Prem Devanbu, and Emily Morgan. A theory of dual channel constraints. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering: New Ideas and Emerging Results*, ICSE-NIER ’20, page 25–28, New York, NY, USA, 2020. Association for Computing Machinery. ISBN 9781450371261. doi: 10.1145/3377816.3381720. URL <https://doi.org/10.1145/3377816.3381720>. 1
- [20] Roe 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. 3.4.1.c
- [21] 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. 1, 2.3.2, 3.1, 3.3, 3.4.1, 3.4.1.b, 4.1, 4.2.2, 4.3.4, 4.5

- [22] Fred Chow, Sun Chan, Shin Ming Liu, Raymond Lo, and Mark Streich. Effective representation of aliases and indirect memory operations in ssa form. In Tibor Gyimóthy, editor, *Compiler Construction*, pages 253–267, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg. ISBN 978-3-540-49939-8. 5.5.2.a
- [23] Berkeley Churchill, Oded Padon, Rahul Sharma, and Alex Aiken. Semantic program alignment for equivalence checking. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, page 1027–1040, New York, NY, USA, 2019. Association for Computing Machinery. ISBN 9781450367127. doi: 10.1145/3314221.3314596. URL <https://doi.org/10.1145/3314221.3314596>. 2.4
- [24] Cristina Cifuentes. *Reverse compilation techniques*. Queensland University of Technology, Brisbane, 1994. 2.1
- [25] Cristina Cifuentes and K John Gough. Decompilation of binary programs. *Software: Practice and Experience*, 25(7):811–829, 1995. 2.1
- [26] Kevin Coogan, Gen Lu, and Saumya Debray. Deobfuscation of virtualization-obfuscated software: a semantics-based approach. In *Proceedings of the 18th ACM conference on Computer and communications security*, pages 275–284, 2011. 4.5
- [27] Manjeet Dahiya and Sorav Bansal. Black-box equivalence checking across compiler optimizations. In *Programming Languages and Systems: 15th Asian Symposium, APLAS 2017, Suzhou, China, November 27-29, 2017, Proceedings 15*, pages 127–147. Springer, 2017. 2.4
- [28] Robin David, Luigi Coniglio, Mariano Ceccato, et al. Qsynth-a program synthesis based approach for binary code deobfuscation. In *BAR 2020 Workshop*, 2020. 4.5
- [29] 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. 1, 2.3.2
- [30] Emanuele De Angelis, Fabio Fioravanti, Alberto Pettorossi, and Maurizio Proietti. Relational verification through horn clause transformation. In *Static Analysis: 23rd International Symposium, SAS 2016, Edinburgh, UK, September 8-10, 2016, Proceedings 23*, pages 147–169. Springer, 2016. 2.4
- [31] Leonardo De Moura and Nikolaj Bjørner. Z3: an efficient smt solver. In *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS’08/ETAPS’08*, page 337–340, Berlin, Heidelberg, 2008. Springer-Verlag. ISBN 3540787992. 5.2.3.a
- [32] Tim Dettmers, Artidoro Pagnoni, Ari Holtzman, and Luke Zettlemoyer. Qlora: Efficient finetuning of quantized llms. *Advances in Neural Information Processing Systems*, 36, 2024. 2.2, 4.2.3
- [33] Weiyu Dong, Jian Lin, Rui Chang, and Ruimin Wang. Cadecff: Compiler-agnostic deobfuscator of control flow flattening. In *Proceedings of the 13th Asia-Pacific Symposium on Internetware*, pages 282–291, 2022. 4.5

- [34] Luke Dramko, Claire Le Goues, and Edward J Schwartz. Fast, fine-grained equivalence checking for neural decompilers. *arXiv preprint arXiv:2501.04811*, 2025. 4.3.2, 5.1, 5.1, 5.2.4, 5.2.6.b, 5.3.1, 5.2, 5.3.2.a, 5.6
- [35] Dennis Felsing, Sarah Grebing, Vladimir Klebanov, Philipp Rümmer, and Mattias Ulbrich. Automating regression verification. In *Proceedings of the 29th ACM/IEEE international conference on Automated software engineering*, pages 349–360, 2014. 2.4
- [36] 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. 2.3.1, 3.1, 3.4.1, 4.2.1
- [37] Leo Gao, Stella Biderman, Sid Black, Laurence Golding, Travis Hoppe, Charles Foster, Jason Phang, Horace He, Anish Thite, Noa Nabeshima, et al. The pile: An 800gb dataset of diverse text for language modeling. *arXiv preprint arXiv:2101.00027*, 2020. 4.2.2
- [38] 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. 3.6
- [39] 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. 3.6
- [40] Ilfak Guilfanov. Decompilers and beyond. In *Black Hat USA*, 2008. 1
- [41] Shubhani Gupta, Aseem Saxena, Anmol Mahajan, and Sorav Bansal. Effective use of smt solvers for program equivalence checking through invariant-sketching and query-decomposition. In *Theory and Applications of Satisfiability Testing—SAT 2018: 21st International Conference, SAT 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9–12, 2018, Proceedings 21*, pages 365–382. Springer, 2018. 2.4
- [42] Shubhani Gupta, Abhishek Rose, and Sorav Bansal. Counterexample-guided correlation algorithm for translation validation. *Proceedings of the ACM on Programming Languages*, 4(OOPSLA):1–29, 2020. 2.4
- [43] 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. 2.3.2, 3.3
- [44] M. S. Hecht and J. D. Ullman. Characterizations of reducible flow graphs. *J. ACM*, 21(3):367–375, July 1974. ISSN 0004-5411. doi: 10.1145/321832.321835. URL <https://doi.org/10.1145/321832.321835>. 5.2.2.a
- [45] Abram Hindle, Earl T Barr, Mark Gabel, Zhendong Su, and Premkumar Devanbu. On the naturalness of software. *Communications of the ACM*, 59(5):122–131, 2016. 1
- [46] Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. The curious case of neural text degeneration. *arXiv preprint arXiv:1904.09751*, 2019. 4.3.4

- [47] Kurt Hornik, Maxwell Stinchcombe, and Halbert White. Multilayer feedforward networks are universal approximators. *Neural networks*, 2(5):359–366, 1989. 2.2
- [48] Iman Hosseini and Brendan Dolan-Gavitt. Beyond the c: Retargetable decompilation using neural machine translation. 2022. 1, 2.3.1, 4.2.1, 6
- [49] Edward J Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yanzhi Li, Shean Wang, Lu Wang, and Weizhu Chen. Lora: Low-rank adaptation of large language models. *arXiv preprint arXiv:2106.09685*, 2021. 2.2
- [50] Peiwei Hu, Ruigang Liang, and Kai Chen. Degpt: Optimizing decompiler output with llm. 2024. 1
- [51] Zecong Hu. Ghcc, 2021. URL <https://github.com/huzecong/ghcc>. 3.2.3, 4.2.2
- [52] Binyuan Hui, Jian Yang, Zeyu Cui, Jiaxi Yang, Dayiheng Liu, Lei Zhang, Tianyu Liu, Jiajun Zhang, Bowen Yu, Keming Lu, et al. Qwen2. 5-coder technical report. *arXiv preprint arXiv:2409.12186*, 2024. 4.2.3
- [53] Nan Jiang, Chengxiao Wang, Kevin Liu, Xiangzhe Xu, Lin Tan, Xiangyu Zhang, and Petr Babkin. Nova: Generative language models for assembly code with hierarchical attention and contrastive learning. In Y. Yue, A. Garg, N. Peng, F. Sha, and R. Yu, editors, *International Conference on Learning Representations*, volume 2025, pages 95905–95926, 2025. URL https://proceedings.iclr.cc/paper_files/paper/2025/file/ef283d62b4bce30854a8d4827f331229-Paper-Conference.pdf. 1, 2.3.1, 4.2.1, 4.2.2, 4.2.2, 4.3.1, 4.5
- [54] Xin Jin, Kexin Pei, Jun Yeon Won, and Zhiqiang Lin. Symlm: Predicting function names in stripped binaries via context-sensitive execution-aware code embeddings. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, pages 1631–1645, 2022. 1, 2.3.2
- [55] 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. 6, 3.4.3
- [56] Aditya Kanade, Amitabha Sanyal, and Uday P Khedker. Validation of gcc optimizers through trace generation. *Software: Practice and Experience*, 39(6):611–639, 2009. 2.4
- [57] Deborah S. Katz, Jason Ruchti, and Eric Schulte. Using recurrent neural networks for decompilation. In *IEEE International Conference on Software Analysis, Evolution and Reengineering*, pages 346–356, 4 2018. doi: 10.1109/SANER.2018.8330222. 1, 2.3.1, 3.1, 3.4.1, 4.2.1
- [58] Shahedul Huq Khandkar. Open coding. 2009. 3.2
- [59] Hyunjin Kim, Jinyeong Bak, Kyunghyun Cho, and Hyungjoon Koo. A transformer-based function symbol name inference model from an assembly language for binary reversing. In *Proceedings of the 2023 ACM Asia Conference on Computer and Communications Security*, pages 951–965, 2023. 1, 2.3.2
- [60] Kathleen Knobe and Vivek Sarkar. Array ssa form and its use in parallelization. In

Proceedings of the 25th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL '98, page 107–120, New York, NY, USA, 1998. Association for Computing Machinery. ISBN 0897919793. doi: 10.1145/268946.268956. URL <https://doi.org/10.1145/268946.268956>. 5.5.2.a

- [61] Denis Kocetkov, Raymond Li, Loubna Ben Allal, Jia Li, Chenghao Mou, Carlos Muñoz Ferrandis, Yacine Jernite, Margaret Mitchell, Sean Hughes, Thomas Wolf, Dzmitry Bahdanau, Leandro von Werra, and Harm de Vries. The stack: 3 tb of permissively licensed source code. *Preprint*, 2022. 4.2.2
- [62] Jeremy Lacomis, Pengcheng Yin, Edward Schwartz, Miltiadis Allamanis, Claire Le Goues, Graham Neubig, and Bogdan Vasilescu. Dire: A neural approach to decompiled identifier naming. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 628–639. IEEE, 2019. 1, 2.3.2, 3.1, 3.3, 3.4.1, 3.4.1.c, 6
- [63] Shuvendu K Lahiri, Chris Hawblitzel, Ming Kawaguchi, and Henrique Rebêlo. Symdiff: A language-agnostic semantic diff tool for imperative programs. In *Computer Aided Verification: 24th International Conference, CAV 2012, Berkeley, CA, USA, July 7-13, 2012 Proceedings 24*, pages 712–717. Springer, 2012. 2.4
- [64] 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. 3.6
- [65] JongHyup Lee, Thanassis Avgerinos, and David Brumley. Tie: Principled reverse engineering of types in binary programs. 2011. 1, 2.3.2, 5.2.4.b
- [66] Daniel Lehmann and Michael Pradel. Finding the dwarf: recovering precise types from webassembly binaries. In *Proceedings of the 43rd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 410–425, 2022. 2.3.2
- [67] Mingyue Liang, Zhoujun Li, Qiang Zeng, and Zhejun Fang. Deobfuscation of virtualization-obfuscated code through symbolic execution and compilation optimization. In *Information and Communications Security: 19th International Conference, ICICS 2017, Beijing, China, December 6-8, 2017, Proceedings 19*, pages 313–324. Springer, 2018. 4.5
- [68] Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. Automatic reverse engineering of data structures from binary execution. In *Proceedings of the 11th Annual Information Security Symposium*, 2010. 2.3.2
- [69] Nuno P Lopes, Juneyoung Lee, Chung-Kil Hur, Zhengyang Liu, and John Regehr. Alive2: bounded translation validation for llvm. In *Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation*, pages 65–79, 2021. 2.4, 5.2.3.d
- [70] Ilya Loshchilov and Frank Hutter. Sgdr: Stochastic gradient descent with warm restarts. *arXiv preprint arXiv:1608.03983*, 2016. 4.2.3

- [71] Leandro T. C. Melo, Rodrigo G. Ribeiro, Marcus R. de Araújo, and Fernando Magno Quintão Pereira. Inference of static semantics for incomplete c programs. *Proc. ACM Program. Lang.*, 2(POPL), December 2017. doi: 10.1145/3158117. URL <https://doi.org/10.1145/3158117>. 5.2.4.b
- [72] Leandro T. C. Melo, Rodrigo G. Ribeiro, Breno C. F. Guimarães, and Fernando Magno Quintão Pereira. Type inference for c: Applications to the static analysis of incomplete programs. *ACM Trans. Program. Lang. Syst.*, 42(3), November 2020. ISSN 0164-0925. doi: 10.1145/3421472. URL <https://doi.org/10.1145/3421472>. 5.2.4.b
- [73] Alan Mycroft. Type-based decompilation (or program reconstruction via type reconstruction). In *Programming Languages and Systems*, pages 208–223, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg. 2.1
- [74] Kedar S Namjoshi and Lenore D Zuck. Witnessing program transformations. In *International Static Analysis Symposium*, pages 304–323. Springer, 2013. 2.4
- [75] George C Necula. Translation validation for an optimizing compiler. In *Proceedings of the ACM SIGPLAN 2000 conference on Programming language design and implementation*, pages 83–94, 2000. 2.4, 5.1
- [76] Vikram Nitin, Anthony Saieva, Baishakhi Ray, and Gail Kaiser. Direct: A transformer-based model for decompiled variable name recovery. *NLP4Prog 2021*, page 48, 2021. 1, 2.3.2, 3.3, 3.4.1
- [77] Matt Noonan, Alexey Loginov, and David Cok. Polymorphic type inference for machine code. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI ’16, page 27–41, New York, NY, USA, 2016. Association for Computing Machinery. ISBN 9781450342612. doi: 10.1145/2908080.2908119. URL <https://doi.org/10.1145/2908080.2908119>. 5.2.4.b
- [78] Diego Novillo et al. Memory ssa-a unified approach for sparsely representing memory operations. In *Proceedings of the GCC Developers’ Summit*, pages 97–110, 2007. 5.5.2.a
- [79] Kuntal Kumar Pal, Ati Priya Bajaj, Pratyay Banerjee, Audrey Dutcher, Mutsumi Nakamura, Zion Leonahenahe Basque, Himanshu Gupta, Saurabh Arjun Sawant, Ujjwala Anantheshwaran, Yan Shoshitaishvili, et al. ”len or index or count, anything but v1”: Predicting variable names in decompilation output with transfer learning. In *2024 IEEE Symposium on Security and Privacy (SP)*, pages 152–152. IEEE Computer Society, 2024. 1, 2.3.2, 4.5
- [80] 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*, ESEC/FSE 2021, page 690–702, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450385626. doi: 10.1145/3468264.3468607. URL <https://doi.org/10.1145/3468264.3468607>. 2.3.2
- [81] Suzette Person, Matthew B Dwyer, Sebastian Elbaum, and Corina S Păsăreanu. Dif-

- ferential symbolic execution. In *Proceedings of the 16th ACM SIGSOFT International Symposium on Foundations of software engineering*, pages 226–237, 2008. 2.4
- [82] Amir Pnueli, Michael Siegel, and Eli Singerman. Translation validation. In *Tools and Algorithms for the Construction and Analysis of Systems: 4th International Conference, TACAS’98*, pages 151–166. Springer, 1998. 2.4
- [83] J.C. Reynolds. Separation logic: a logic for shared mutable data structures. In *Proceedings 17th Annual IEEE Symposium on Logic in Computer Science*, pages 55–74, 2002. doi: 10.1109/LICS.2002.1029817. 5.2.3
- [84] 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. 1.3, 5.2.2.a
- [85] Baptiste Roziere, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Romain Sauvestre, Tal Remez, et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023. 4.2.3
- [86] David E Rumelhart, Geoffrey E Hinton, and Ronald J Williams. Learning representations by back-propagating errors. *Nature*, 323(6088):533–536, 1986. 2.2
- [87] Eric Schulte, Jason Ruchti, Matt Noonan, David Ciarletta, and Alexey Loginov. Evolving exact decompilation. In *workshop on binary analysis research (BAR)*, 2018. 2.1
- [88] Edward J Schwartz. *Abstraction Recovery for Scalable Static Binary Analysis*. Carnegie Mellon University, 2014. 2.1
- [89] 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. 1
- [90] 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. 1
- [91] Freda Shi, Xinyun Chen, Kanishka Misra, Nathan Scales, David Dohan, Ed H. Chi, Nathanael Schärli, and Denny Zhou. Large language models can be easily distracted by irrelevant context. In Andreas Krause, Emma Brunskill, Kyunghyun Cho, Barbara Engelhardt, Sivan Sabato, and Jonathan Scarlett, editors, *Proceedings of the 40th International Conference on Machine Learning*, volume 202 of *Proceedings of Machine Learning Research*, pages 31210–31227. PMLR, 23–29 Jul 2023. URL <https://proceedings.mlr.press/v202/shi23a.html>. 5.4.2.b
- [92] 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. 3.1, 3.4.2.d
- [93] Asia Slowinska, Traian Stancescu, and Herbert Bos. Howard: A dynamic excavator for reverse engineering data structures. In *Network and Distributed System Security Symposium (NDSS)*, 2011. 2.3.2
- [94] Ian Smith. Binsub: The simple essence of polymorphic type inference for machine

- code. In *Static Analysis: 31st International Symposium, SAS 2024, Pasadena, CA, USA, October 20–22, 2024, Proceedings*, page 425–450, Berlin, Heidelberg, 2024. Springer-Verlag. ISBN 978-3-031-74775-5. doi: 10.1007/978-3-031-74776-2_17. URL https://doi.org/10.1007/978-3-031-74776-2_17. 5.2.4.b
- [95] Diomidis Spinellis, Zoe Kotti, and Audris Mockus. A dataset for github repository deduplication. In *Proceedings of the 17th international conference on mining software repositories*, pages 523–527, 2020. 4.2.2
 - [96] Michael Stepp, Ross Tate, and Sorin Lerner. Equality-based translation validator for llvm. In *Computer Aided Verification: 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings 23*, pages 737–742. Springer, 2011. 2.4
 - [97] Xiangzhe Xu Zhuo Zhang Zian Su, Ziyang Huang Shiwei Feng Yapeng Ye, Nan Jiang Danning Xie Siyuan Cheng, and Lin Tan Xiangyu Zhang. Unleashing the power of generative model in recovering variable names from stripped binary. 2025. 2.3.2
 - [98] Hanzhuo Tan, Qi Luo, Jing Li, and Yuqun Zhang. LLM4Decompile: Decompiling binary code with large language models. In Yaser Al-Onaizan, Mohit Bansal, and Yun-Nung Chen, editors, *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, pages 3473–3487, Miami, Florida, USA, November 2024. Association for Computational Linguistics. doi: 10.18653/v1/2024.emnlp-main.203. URL <https://aclanthology.org/2024.emnlp-main.203/>. 1, 2.3.1, 4.1, 4.1c, 4.1, 4.2.1, 4.2.2, 4.2.2, 4.1, 4.2.2, 4.2.3, 4.3.1, 4.3.4, 4.2, 4.4, 4.5
 - [99] Ross Tate, Michael Stepp, Zachary Tatlock, and Sorin Lerner. Equality saturation: a new approach to optimization. In *Proceedings of the 36th annual ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, pages 264–276, 2009. 2.4
 - [100] CodeGemma Team, Heri Zhao, Jeffrey Hui, Joshua Howland, Nam Nguyen, Siqi Zuo, Andrea Hu, Christopher A Choquette-Choo, Jingyue Shen, Joe Kelley, et al. Codegemma: Open code models based on gemma. *arXiv preprint arXiv:2406.11409*, 2024. 4.2.3
 - [101] Ramtine Tofighi-Shirazi, Maria Christofi, Philippe Elbaz-Vincent, and Thanh-Ha Le. Dose: Deobfuscation based on semantic equivalence. In *Proceedings of the 8th Software Security, Protection, and Reverse Engineering Workshop*, pages 1–12, 2018. 4.5
 - [102] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. Attention is all you need. In I. Guyon, U. Von Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 30, 2017. URL https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf. 2.3.1, 2.3.2, 2.3.2
 - [103] Freek Verbeek, Pierre Olivier, and Binoy Ravindran. Sound c code decompilation for a subset of x86-64 binaries. In *Software Engineering and Formal Methods: 18th International Conference, SEFM 2020, Amsterdam, The Netherlands, September 14–18, 2020, Proceedings 18*, pages 247–264. Springer, 2020. 5.1
 - [104] 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. 1, 3.1, 3.4.1, 4.1, 5.1, 5.1
- [105] Danning Xie, Zhuo Zhang, Nan Jiang, Xiangzhe Xu, Lin Tan, and Xiangyu Zhang. Resym: Harnessing llms to recover variable and data structure symbols from stripped binaries. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security*, pages 4554–4568, 2024. 1, 2.3.2, 4.1
 - [106] Jiaqi Xiong, Guoqiang Chen, Kejiang Chen, Han Gao, Shaoyin Cheng, and Weiming Zhang. Hext5: Unified pre-training for stripped binary code information inference. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 774–786. IEEE, 2023. 1, 2.3.2, 4.2.2, 4.2.2
 - [107] 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 *NDSS*, 2015. 1
 - [108] 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 *2016 IEEE Symposium on Security and Privacy (SP)*, pages 158–177, 2016. doi: 10.1109/SP.2016.18. 1
 - [109] Wu Yang, Susan Horwitz, and Thomas Reps. Detecting program components with equivalent behaviors. Technical report, University of Wisconsin-Madison Department of Computer Sciences, 1989. 2.4, 4.3.2
 - [110] 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. 3.1
 - [111] Geunha You, Gyoosik Kim, Sangchul Han, Minkyu Park, and Seong-Je Cho. Deoptfuscator: Defeating advanced control-flow obfuscation using android runtime (art). *IEEE Access*, 10:61426–61440, 2022. 4.5
 - [112] 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. 3.4.1
 - [113] Zhuo Zhang, Yapeng Ye, Wei You, Guanhong Tao, Wen-chuan 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. 1, 2.3.2, 3.3
 - [114] Chang Zhu, Ziyang Li, Anton Xue, Ati Priya Bajaj, Wil Gibbs, Yibo Liu, Rajeev Alur, Tiffany Bao, Hanjun Dai, Adam Doupé, et al. Tygr: Type inference on stripped binaries using graph neural networks. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 4283–4300, 2024. 1, 2.3.2, 4.1

- [115] Muqi Zou, Arslan Khan, Ruoyu Wu, Han Gao, Antonio Bianchi, and Dave Jing Tian. D-Helix: A generic decompiler testing framework using symbolic differentiation. In *33rd USENIX Security Symposium (USENIX Security 24)*, pages 397–414, 2024. 2.4, 5.1, 5.3.2.a
- [116] Lukáš 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. 1