Idioms: A Simple and Effective Framework for Turbo-Charging Local Neural Decompilation with Well-Defined Types

Luke Dramko
Carnegie Mellon University
School of Computer Science
lukedram@cs.cmu.edu

Claire Le Goues
Carnegie Mellon University
School of Computer Science
clegoues@cs.cmu.edu

Edward J. Schwartz
Carnegie Mellon University
Software Engineering Institute
eschwartz@cert.org

***Abstract*—Decompilers help reverse engineers analyze software at a higher level of abstraction than assembly code. Unfortunately, because compilation is lossy, traditional decompilers, which are deterministic, produce code that lacks many characteristics that make source code readable in the first place, such as variable and type names. Neural decompilers offer the exciting possibility of *statistically* filling in these details. Unfortunately, existing work in neural decompilation suffers from substantial limitations that preclude its use on real code, such as the inability to provide definitions for user-defined composite types. In this work, we introduce IDIOMS, a simple, generalizable, and effective neural decompilation approach that can finetune any LLM into a neural decompiler capable of generating the appropriate user-defined type definitions alongside the decompiled code, and a new dataset, REALTYPE, that includes substantially more complicated and realistic types than existing neural decompilation benchmarks. We show that our approach yields state-of-the-art results in neural decompilation. On the most challenging existing benchmark—EXEBENCH—our model achieves 54.4% accuracy vs. 46.3% for LLM4Decompile and 37.5% for Nova; on REALTYPE, our model performs at least 95% better.**

I. Introduction

Decompilation—the reconstruction of a source code representation from an executable program—is critical for a variety of security tasks, including malware analysis, vulnerability research, and fixing legacy software when the original source code is unavailable [1], [2]. Unfortunately, because the compilation process loses many programmer-oriented abstractions, such as variable names, types, and comments, the code produced by traditional deterministic compilers is often difficult to read and understand.

To address these problems, researchers have been applying machine learning, which offers the possibility to guess or predict such missing abstractions *statistically* based on the surrounding context that is *not* removed during compilation. Some work restricts itself to recover specific abstractions, such as variable names [3], [4], [5], [6], function names [7], [8], [9], variable types [10], [11], [12], or several abstractions at once [13], [14], [15]. While promising, there are numerous issues with decompiled code [16] and maintaining a model for each one is burdensome and inflexible.

More recently, researchers are leveraging large language models (LLMs) to predict the original source code in its entirety, rather than specific abstractions, which we call *neural decompilation* [17], [18], [19]. Neural decompilation is appealing because, in theory, it can statistically recover *any* type of abstraction that is missing or distorted, including the specific abstractions above. Neural decompilers have the potential to vastly outperform traditional, deterministic decompilers.

For example, Figure 1a shows a function that finds the index of an element in a hash table which uses robin-hood hashing for collision management. Figure 1b shows the same function, but after having been compiled and decompiled with the industry-standard Hex-Rays decompiler. The decompiled code is challenging to interpret because it lacks meaningful names, and it also misrepresents the pointer to the hash table as an `__int64`. Figure 1c shows the prediction of LLM4Decompile [19], a state-of-the-art neural decompiler. Although LLM4Decompile did not recover meaningful names, it did predict that the function's first argument had a structure type, and converted the raw pointer arithmetic into more readable field accesses.

As the example shows, current neural decompilers are promising, but their output is far from ideal. A security practitioner glancing at the original code (Figure 1a) can immediately tell that the function is hashing-related because of names like `hash_find_index` and `struct hash`. The neurally-decompiled code (Figure 1c) contains no such clues. More problematically, reverse engineers often work across multiple levels of abstraction [20] such as assembly code and decompiled code. This requires knowledge of the memory layout of the data structures in the decompiled code. Although `struct FUN_0009ff84` is clearly a structure, its memory map is undefined because existing neural decompilers *are not trained to produce type definitions* which prevents reverse

Network and Distributed System Security (NDSS) Symposium 2026
23 - 27 February 2026 , San Diego, CA, USA
ISBN 979-8-9919276-8-0
https://dx.doi.org/10.14722/ndss.2026.240795
www.ndss-symposium.org

```c
struct hash {
  int hash_size;
  int item_cnt;
  struct gap_array *data;
  int (*hash_make_key)(void *item);
  int (*cmp_item)(void *item1, void *item2);
}
struct gap_array {
  int len;
  void **array;
}

int hash_find_index(struct hash *h, void *item) {
    void *cnx;
    int index = hash_make_key(h, item);
    int cnt = 0;
    cnx = gap_get(h->data, index);
    while (cnx != NULL) {
        if (cnt++ > h->hash_size) return -1;
        if (!h->cmp_item(cnx, item)) break;
        index = hash_next_index(h, index);
        cnx = gap_get(h->data, index);
    }
    if (cnx == NULL) return -1;
    return index;
}
```

(a) A function which finds the index of an element in a hash table where collisions are handled with robin-hood hashing.

```c
__int64 __fastcall func4(__int64 a1, __int64 a2) {
int v2;           // eax
__int64 result;   // rax
int v4;           // [rsp+10h] [rbp-10h]
unsigned int v5; // [rsp+14h] [rbp-Ch]
__int64 i;        // [rsp+18h] [rbp-8h]
v5 = func2(a1, a2);
v4 = 0;
for (i = func1(*(_QWORD *)(a1 + 8), v5); i;
     i = func1(*(_QWORD *)(a1 + 8), v5)) {
  v2 = v4++;
  if (v2 > *(_DWORD *)a1) return 0xFFFFFFFFLL;
  if (!(*(unsigned int(__fastcall **)(__int64, __int64
       ))(a1 + 24))(i, a2))
    break;
  v5 = func3((_DWORD *)a1, v5);
}
if (i)
  result = v5;
else
  result = 0xFFFFFFFFLL;
return result;
}
```

(b) The function from Figure 1a, post-decompilation by Hex-Rays. It is missing meaningful and informative name and type information.

```c
int FUN_00100155(struct FUN_0009ff84 *VAR_0,void *VAR_1)
    {
  int VAR_2;
  int VAR_3;
  void *VAR_4;
  VAR_2 = FUN_0009ff86(VAR_0, VAR_1);
  VAR_3 = 0;
  VAR_4 = FUN_0009ff88(VAR_0->VAR_5, VAR_2);
  while (VAR_4) {
    if (VAR_0->VAR_6 < VAR_3) { return -1; }
    if (!VAR_0->VAR_7(VAR_4, VAR_1)) { break; }
    VAR_2 = FUN_0009ff89(VAR_0, VAR_2);
    VAR_4 = FUN_0009ff88(VAR_0->VAR_5, VAR_2);
    VAR_3++;
  }
  if (VAR_4) { return VAR_2; }
  return -1;
}
```

(c) Figure 1a, decompiled by LLM4Decompile-Ref [19]. The sequence of operations closely matches the original, but the semantics are underspecified and thus unverifiable without a definition of `struct FUN_0009ff84`.

```c
struct hash_t {
  int size;
  int count;
  struct hash_table_t *table;
  int (*hash)(void *key);
  int (*cmp)(void *key1, void *key2);
};
struct hash_table_t {
  int size;
  void **items;
};
int hash_find(struct hash_t *hash, void *key) {
    int index = hash_index(hash, key);
    int i = 0;
    void *item = hash_get(hash->table, index);
    while (item != ((void *)0)) {
        if (i++ > hash->size) { return -1; }
        if (hash->cmp(item, key) == 0) { break; }
        index = hash_next(hash, index);
        item = hash_get(hash->table, index);
    }
    return (item == ((void *)0)) ? -1 : index;
}
```

(d) IDIOMS output for Figure 1a, including the necessary type definitions missing from Figure 1c.

Fig. 1: A function with two user-defined types and different decompilations of it.

engineers from working at a lower abstraction level. Type definitions are also required to compile the code, and for most types of static analysis, such as those that find and patch vulnerabilities, or deploy certain types of software defenses. As we discuss in Section II-B (Table I), such user-defined types (UDTs) (e.g., structs) are widespread in real code. This problem has been masked in existing work because current benchmarks for neural decompilation feature very few UDTs.

In this work, we propose a suitable new dataset, REALTYPE, and a novel method that harnesses it for training neural decompilers that explicitly reconstructs UDT definitions alongside reconstructed code. Figure 1d shows the output of our approach, which we call IDIOMS, since it recovers idiomatic code. IDIOMS predicts a complete type definition for the output structure, `struct hash_t`, as well as the names of its fields. The resulting code is well-defined and thus amenable to the types of reasoning common in reverse engineering.

IDIOMS is motivated by two main insights:

Insight 1: Code and type definitions should be predicted jointly. As we showed in Figure 1, decompiled code and type definitions are fundamentally interdependent: meaningful variable names and field accesses depend on the underlying type definitions, while accurate type reconstruction requires knowing how those types are used throughout the code. This interdependence argues strongly for joint prediction rather than sequential approaches. Existing neural decompilers that

predict code without type definitions produce *underspecified* outputs—the semantics of struct field accesses cannot be determined without knowing the memory layout of those structs. They do not tell a reverse engineer what memory offsets will be accessed. Conversely, predicting types in isolation from their usage context discards valuable information about how fields are accessed and manipulated.

Insight 2: Scattered evidence for UDT type recovery necessitates consideration of broad context. Predicting a UDT definition is fundamentally difficult due to we call the *scattered evidence* problem: typically, any given function accesses only a subset of a UDT's fields [21]. Full UDT structure and semantics cannot be determined from partial usage patterns, challenging type inference for a single function in isolation. This challenge mirrors the fundamental problem faced by traditional type inference algorithms when analyzing executables—they must aggregate evidence across multiple functions to reconstruct complete type definitions, which is precisely why most algorithms are interprocedural [21].

To address this, we provide broader context in the form of *neighboring* functions—those close to the target function in the call graph. A function's callees and callers process related input, output, and internal values, often revealing additional clues about UDT structure and usage patterns. This interprocedural evidence enables more accurate type inference by aggregating partial information scattered across the program's call graph, allowing the model to reconstruct complete UDT definitions that would be impossible to infer from any single function alone.

We propose a new training strategy and family of IDIOMS neural decompilation models that (a) jointly predict code and type definitions, enabling consistent type application, and (b) leverage neighboring functions in the call graph to provide the necessary context for UDT reconstruction. Like LLM4Decompile-Ref [19] and earlier work on specific renaming tasks [3], [13], our models take as input the output of deterministic decompilers applied to binaries. This approach leverages decades of progress in deterministic decompilation, reducing the difficulty of the model's task. Unlike existing neural decompilers [22], [19], [17], IDIOMS models explicitly predict definitions of all UDTs used by a function alongside its definition.

The IDIOMS approach is designed to be lightweight, flexible, and easily generalizable to arbitrary LLMs. (We apply it to five different LLMs in Section IV.) It is text-based and makes no architecture-specific assumptions. This means that as newer, more powerful LLMs are released, they can be easily adapted via IDIOMS to perform joint neural decompilation with type definition prediction. While the dataset preparation and evaluation software introduced in this work is sophisticated, IDIOMS itself is easy to use, requiring small amounts of python code and widely available, and actively maintained python packages, unlike many existing tools.

We demonstrate experimentally that IDIOMS substantially outperforms existing work: IDIOMS scores 17–36% better than LLM4Decompile [19] and 37–78% better than Nova [17] on EXEBENCH [23], the most challenging existing benchmark, and 95–205% better on the realistic REALTYPE with its realistic UDTs. These performance improvements hold as well over prior work focused strictly on type recovery. We perform controlled experiments to demonstrate that decompiling functions with UDTs is substantially more challenging than those without, and that including neighboring context contributes substantially to IDIOMS performance (improving structural accuracy by up to 63%). Finally, we go beyond introducing a new modeling approach, making numerous impactful methodological improvements along the pipeline, from dataset preparation to evaluation.

In summary, we contribute:

- A new dataset, REALTYPE, containing 154,301 training functions and 2,862 evaluation functions with realistic user-defined types (UDTs) and their complete definitions extracted from preprocessed source code.
- A novel approach that enables neural decompilers to jointly predict both function code and complete user-defined type definitions simultaneously.
- IDIOMS, a state-of-the-art family of neural decompilation models, which outperform LLM4Decompile and Nova by 95-205%, and existing standalone type recovery techniques by at least 73%, on realistic code.
- Experimental evidence demonstrating that (a) UDTs substantially increase the difficulty of neural decompilation, (b) neighboring function context significantly improves UDT prediction accuracy by up to 63%, and (c) these findings generalize across model architectures and sizes.
- A fine-grained evaluation that measures performance on different aspects of neural decompilation, including both correctness and code improvements.

For reproducibility, we release the code used to build datasets and train and evaluate models; REALTYPE; and all IDIOMS and other models we train in our experiments.[1]

II. Approach

Figure 2 overviews our approach. Decompilation aims to produce source or source-like representations of compiled code to help reverse engineers more easily understand or manipulate it. Neural decompilation generally entails *training* machine learning models to predict original source code for a target binary function of interest (left-hand-side of Figure 2).

Section II-A details the modeling approach. IDIOMS models take the output of a deterministic decompiler and predicts the original source code of the target function and a list of user-defined types in that function. IDIOMS operates on decompiler output rather than on the compiled binary directly because doing so naturally leverages significant advances in deterministic decompilation. Tan et al. [19] compare neural decompilation from assembly and from deterministic decompiler output and find that the latter produces correct output more often.

[1]Code can be found at https://github.com/squaresLab/idioms; models and datasets can be found at https://doi.org/10.5281/zenodo.15683630.

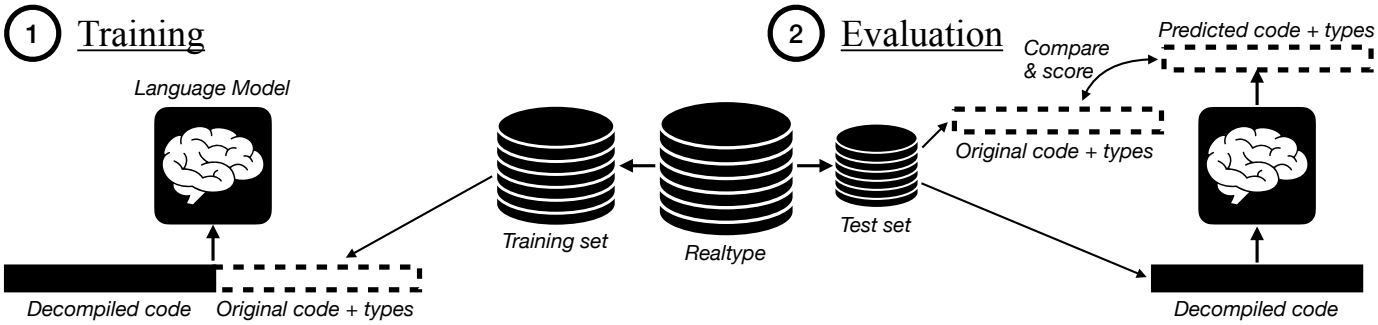

Fig. 2: Our high-level approach. We finetune causal language models on our dataset, REALTYPE, into IDIOMS models, then evaluate them.

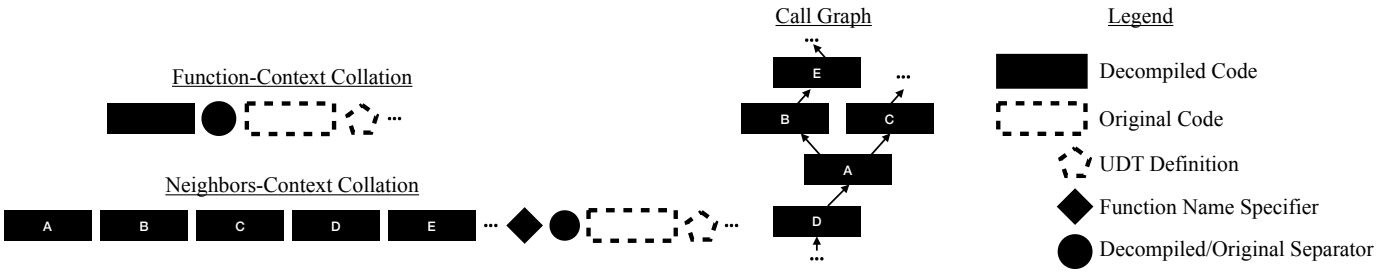

Fig. 3: Model training sequence organization for IDIOMS models. Neighboring context entails listing decompiled functions starting the target (**A**, here) in breadth-first call graph order. We compare neighboring- to function-context in our experiments.

Section II-B describes the dataset, REALTYPE. The IDIOMS approach requires suitable example input and output data. Existing datasets are inadequate for this task for two reasons: (1) a lack of variables with UDTs and their definitions, and (2) they are function level, and thus we cannot build call graphs. We therefore build a suitable new dataset, REALTYPE.

Once trained, models can predict the original code and UDT definitions for a given decompiled function (right-hand-side of Figure 2). In practice, this entails applying a deterministic decompiler first (standard, in reverse engineering) and then applying an IDIOMS model to the result. For evaluation, we take advantage of the ground truth available in our data (the original code, pre-compilation). However, care must be taken to evaluate on data not included in the training dataset.

A. Modeling

Model architecture and finetuning. Language models are state-of-the-art neural networks that encode information in learned (*trained*) parameters. To partially overcome the challenges of developing sufficient training data for a particular task, a model can be *pretrained* on data for a related task, and then *finetuned* on a smaller quantity of data relevant to the target task. We follow this paradigm for neural decompilation by finetuning causal language models pretrained on code.

Figure 3 illustrates the organization of the token sequence our model expects. Causal language models process sequences of tokens, which are small, discrete units of text that serve as the basic input representation on which language models operate. In this kind of supervised context, the task is to learn to predict an output (original code + UDT definitions) from an input sequence (decompiled code). For training, the input and output are concatenated into a single sequence, delimited with a *separator token*. For evaluation, or use in practice, the model is prompted with the input and separator token alone to generate the output (the rest of the sequence).

Jointly predicting code and type definitions. The "output" side of the (training) context contains the function definition and associated user defined types from the original source code (everything to the right of ● in Figure 3). Code and type definitions are in the same sequence. Causal language models use the entire preceding sequence to predict the next token. Thus, field names, and how they are used in the predicted code, are all available to the model as it generates type definitions. That is, code and type definitions are generated *jointly*.

Additional context. The evidence for UDT type recovery is scattered across multiple functions [16]. IDIOMS models use neighboring function context (left of ● in Figure 3). It is arranged so the target decompiled code is first; callees and callers of the target function are placed subsequently. Their callees and callers are then added (if not previously included), etc. This can include all functions in the call graph, space allowing. This breadth-first order is deliberate. A model should relate information it learns from neighboring context to inform type prediction in the target function. This requires an unbroken trace from the source to the target function. Practically, as well, providing call graph context in BFS order allows an arbitrary cutoff on the sequence's right side to respect model context limits, while maintaining trace integrity. The neighboring context is followed by a special

name-indicator token, and the target function's name in the decompiled code.

Figure 3 also shows sequences with function-level context, used by existing prior work on neural decompilation [17], [19], [22], [24], [25], [26], [27]. Function-level context consists of the target decompiled function. We demonstrate the value of neighboring context (Section IV-D) by comparing with models built using only the decompiled function as context. For a full sample prompt, see the appendix.

An advantage of the IDIOMS process is that it interfaces with generative neural models at a purely textual level, requiring no architecture-specific details, unlike some existing work (e.g. Nova [17]). It can therefore be applied to newer, more powerful pre-trained models as they are released. The flexibility and generalizability of IDIOMS means that it can get better with time; the modeling approach is about suitably capturing and asking for the necessary information from the generative models' input/output sequence.

B. Dataset

Our model architecture and overall goals require a training dataset that (1) contains code that uses representative UDTs, (2) supports the construction of interprocedural call graphs from the compiled binaries. Existing neural decompilation datasets like EXEBENCH [23] and HUMANEVAL-DECOMPILE [19], used to evaluate state-of-the-art LLM4Decompile [19] and Nova [17], provide only single-function context, and have fewer and simpler UDTs than real-world code. We therefore construct a novel dataset, REALTYPE, with 154,301 training functions and 2,862 evaluation functions. Table I compares REALTYPE with existing datasets.

TABLE I: Complexity in Evaluation Datasets

	HEDecomp*	exebench	REALTYPE
Lines of code	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	N/A	2.2	17.6
Type-tree complexity	1.4	1.5	16.2
UDT Type-tree complexity	N/A	4.2	57.3

UDTs are user-defined types (`struct`, `union`). REALTYPE is our dataset. HUMANEVAL-DECOMPILE [19] (HEDECOMP*) is based on programming challenge problems. EXEBENCH [23] is mined from GitHub, but contains many fewer UDTs. Type complexity is the number of nodes in the type's tree representation (primitive types are leaf nodes). UDT Type-tree complexity includes only UDTs; Type-tree complexity, all types.

Mining functions with UDTs. We cloned and compiled majority-C-language repositories from GitHub using GHCC [28]. GHCC executes standard configuration and build scripts, extracts any resulting ELF-format binary files, and archives repositories under 100MB. Among the ELF files are object (`.o`) files. Including object files increases the amount of data available because not all projects build completely. However, for projects that do build completely, the same function is present at least twice (in the object file and overall binary). We filter object files for which its corresponding functions appear in another binary in the same repository. For convenience, we reused preprocessing scripts from the DIRTY [13] replication package to decompile binaries using Hex-Rays. We filter out PLT stubs, then canonicalize functions' names to `funcX`, where X is an integer, a standard scheme [14], [17], [19].

For each archive, we record original code, and parse its `gcc`- preprocessed version to record `typedef` aliases and UDT definitions (together, our output data). We store all types in a python object model representing the C type system (built atop DIRTY's [13] type system), allowing fine-grained analysis. The result reflects the expressive power of C's type system, including arbitrary nesting and typed function pointers.

To simplify evaluation and reduce ambiguity, we canonicalize all type descriptors. We traverse each function's AST and record information from variable declarations, typecasts, and return types. We de-alias by following previously created `typedef` alias chains, and standardize type names to canonical C forms. For example, platform-specific aliases like `int32_t` and `__int32_t` are both replaced with the standard type `int`. [2] This standardization simplifies evaluation and reduces ambiguity in mapping between a type's memory representation and its syntax as learned by the neural decompiler. We store each canonicalized function with types for all its variables.

We then matched preprocessed with decompiled functions, organized by binary. This reflects realistic decompilation use cases: a reverse engineer often analyzes one or more binaries at once, without access to original source. We compute and store interprocedural call graphs within each binary.

Deduplication. *Data leakage* is a key risk in machine learning, because models tend to "memorize" examples in the training dataset. Training and evaluation sets must therefore be disjoint. The risk of data leakage via pretraining is relatively low for neural decompilation (Section V). However, finetuning *does* risk leakage because duplicates on GitHub are extremely common [29]. Exact-match deduplication is insufficient: many project copies represent different past versions or otherwise contain small modifications. We deduplicate the dataset to enable disjoint training/testing in two complementary ways:

- Minhashing [30] clusters similar text files, and is a popular, robust choice to deduplicate code datasets [31], [32]. We treat all C files in a repository as a "document" and clusters of repositories as duplicates. We select repository from each cluster which produced the most data.
- By-project splitting: we ensure that all data from a given repository is assigned entirely to only one of the train/validation/test splits. This prevents models from memorizing project-specific details or conventions (including UDTs) from some functions in a project and applying them to

[2] Because the C standard specifies minimum but not maximum type sizes, `int` is technically ambiguous—motivating alternative names in the first place. We performed all experiments on a x86_64 `linux` platform with `gcc`, so this was not an issue. We chose C keywords for simplicity, but any alternative consistent convention describing, e.g., type sizes explicitly, would be fine.

different functions in the same project. This has been demonstrated as empirically important in prior evaluations of ML-based decompilation [14].

III. Experimental Design

Our experiments address six research questions:

RQ1: *How does* IDIOMS *perform relative to the best existing neural decompilation techniques?*

RQ2: *How does* IDIOMS *fare when exposed to optimizations?* Optimizations are common, and thus we evaluate whether IDIOMS' advantages apply to optimized code.

RQ3: *To what extent do UDTs affect neural decompilation?* Existing work is evaluated on benchmarks that contain limited UDTs, despite their prevalence in real-world code.

RQ4: *How does performance change with model and dataset size?* We examine scaling's impact on performance.

RQ5: *How does adding neighboring context affect the quality of neural decompilers?* Evidence needed to understand UDTs may be spread throughout a program.

RQ6: *How does* IDIOMS *recover types compared with existing type recovery techniques?* IDIOMS is implicitly in part a type recovery technique; we compare to prior work.

Baselines. We primarily compare against two state-of-the-art neural decompilation models: Nova [17] and LLM4Decompile [19]. Nova takes assembly as input (rather than decompiled C), featuring a custom attention designed for assembly. LLM4Decompile has two versions: one for assembly input and one for Ghidra-decompiled code (a Hex-Rays alternative). We use the latter, higher-performing version. The prior works' replication packages include model code, weights and minimal example snippets. We therefore wrote new evaluation scripts, totaling 726 lines of code, including reconstructing Nova's complex assembly preprocessing.

We do not compare against either SLaDe [22] or DecLLM [33], both notable recent approaches for neural decompilation. By design, both use the tests associated with each target binary or function during prediction. This is unrealistic for reverse engineering, where source code and test cases are typically unavailable. Even with source, EXEBENCH [23] authors could only generate tests for 15.4% of the dataset.

A. Datasets and Processing

We use two datasets in our experiments: REALTYPE (our new dataset) and EXEBENCH. EXEBENCH is the most challenging existing benchmark in prior work. It features code extracted from GitHub, with definitions for external symbols for some functions. Most of the definitions are synthetic, automatically generated by the authors, and do not feature the full UDT definitions. Many have a single field named `dummy`, marking placeholder values. A subset (approximately 15.4%) of the functions are associated with automatically-generated unit tests that can be used as a proxy for decompilation correctness. The test and validation sets are selected out of those 15.4% (hence the bias illustrated in Table I). We evaluate EXEBENCH experiments only on the "real" subpartition of EXEBENCH's test set, avoiding the above-noted problems. The "synth" subpartition is also easier, inflating performance [22].

We decompile both test sets again using Ghidra, to evaluate LLM4Decompile (which expects Ghidra output as input); evaluating it on Hex-Rays data would represent a covariate shift that would unfairly hurt LLM4Decompile.

Test set processing. We exclude EXEBENCH examples where the provided oracle solution and dependencies don't compile or don't pass all tests (that is, those that are internally inconsistent)—about 16% of the `test_real` partition. We exclude a further 16% of the EXEBENCH `test_real` set on which DIRTY [13]'s decompilation scripts fail (producing no input for the model). Following LLM4Decompile [19], we do not evaluate on examples for which the decompiled function exceeds the model's context window (about 3% of each test set). When comparing expanded context to the function-only context (Section IV-D), we evaluate both on the same subset of the test set.

Model predictions on REALTYPE sometimes include degenerate text [34][3] in the form of repeating type definitions, after predicting sensible function definitions and UDTs. We hypothesize this happens simply because the task of predicting complex UDT definitions is extremely difficult. Because of the scattered evidence problem, sometimes the decompiled context is *not* sufficient to fully predict the full UDT definitions, so the model may learn to generate arbitrary type definitions after anything it *can* figure out. Empirically, we saw that degenerate text always appears after non-degenerate predictions and are never referenced in the function text. Thus, in our evaluation, we automatically ignore any type definitions unused by the predicted code, which we assume are degenerate.

B. Metrics

A good neural decompiler (1) preserves execution semantics, and (2) produces names, types and code that are more readable and idiomatic (in a style mimicking a human developer's). The former goal means that a good neural decompiler should be no worse than a deterministic one, and the latter means it must add value. 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. In some respects, an ideal decompiler is one that perfectly "undoes" the compilation, reconstructing the original source.

Our metrics suite is significantly more comprehensive than those used in prior work [17], [19], [22]. The original Nova and LLM4Decompile works approximate semantic preservation via unit tests, a coarse measure [35]; they do not evaluate other code improvements that motivate neural decompilation. SLaDe [22] uses normalized string-edit distance to coarsely estimate code improvement. Some work measures re-compilability [33], a necessary but insufficient condition for unit test accuracy (code that compiles may be incorrect).

[3] A common but poorly-understood phenomenon, even in large models.

```c
int env_loc(char *s){
  int h = 0;
  while (*s++)
    h = h * 31 + *s;
  return h % 131;
}
```

(a) Original code

```c
int hash(char *key) {
  int h = 0;
  while (*key++)
    h = h * 31 + *key;
  return h % 131;
}
```

(b) IDIOMS' prediction

```c
unsigned func0(const char *str) {
    unsigned hash = 0;
    while (*str)
        hash = hash * 31 + *str++;
    return hash % 131;
}
```

(c) Nova's prediction

Fig. 4: A polynomial rolling hash function, decompiled by IDIOMS and Nova [17]. The original and IDIOMS' prediction have isomorphic dependency graphs; Nova's differs. The original code starts hashing the string's *second* character; Nova's prediction starts with the first. This is reflected in the function's dataflow dependencies.

1) Semantic preservation: Because program equivalence is undecidable, we seek a practical approximation of semantic preservation. We use complementary proxies:

(1) Unit tests. EXEBENCH includes unit tests for examples in the test set, which we leverage to evaluate the semantic fidelity of decompiled code. Tests are complete but unsound; this means that if they fail, we can be sure the decompiled code is incorrect, but if they pass, we cannot be sure the decompiled code is correct. This makes them a useful (if optimistic) proxy for correctness. *(Metric name: passes exebench tests)*

(2) Static, dependency-based equivalence. REALTYPE does not include tests. Instead, we also use a static, dependency-based equivalence check to compare decompiled code to the original source [36]. This approach effectively asks "Do two pieces of code perform the same operations in the same order?" It compares function dependency graphs, with nodes representing operations (assignments, function calls, etc.) and data sources (constants, parameters, global variables), and edges representing the control-flow and data-flow dependencies between them. UDT fields are represented as constants. If two dependency graphs are isomorphic, their corresponding functions have the same structure, performing the same operations in the same (partial) order, even if variable names differ.

Figure 4 illustrates this with a particularly subtle example where a neural decompiler produces plausible-looking but incorrect output. The original code begins hashing from the *second* character in the string, while Nova's prediction starts from the first character. This seemingly minor difference creates distinct dataflow dependencies that dependency-equivalence can detect. Beyond such subtle errors, dependency-equivalence can capture other types of incorrect predictions, including omissions, additions, and reordered expressions. However, the approach has limitations: it may fail to recognize semantically-equivalent syntactic forms, such as `x * 2` versus `x << 1`[4].

Dependency graph isomorphism provides a sound approximation of program equivalence [37], with two caveats: (1) side effects, a rare source of unsoundness in practice [36]; and (2) type correctness. To mitigate the second issue, we therefore also measure the fraction of a neural decompiler's predictions for which all variable types are equivalent to those in the ground truth *in addition to* being dependency-equivalent to the ground truth. Note that this type requirement is very conservative; vanilla dependency-equivalence captures whether the operations in a decompiled function are at minimum correct, and in the correct order. *(Metric names: dependency-equivalence, strict dependency-equivalentce (typechecks).)*

Dependency-equivalence relies on UDT definitions to align field accesses, and thus can be overly harsh on prior work, like LLM4Decompile and Nova, that do not produce full type definitions. However, these neural decompilers still often usefully predict variables to have `struct` types, and rewrite pointer arithmetic into struct field-access operations (as in Figure 1c). However, the *names* of the fields often differ, making it difficult to determine which fields correspond to each other. Thus, we introduce a more permissive version of dependency-equivalence under a principle of *consistency*: there must be a bijective mapping between the names of the fields that occur in dependency-graph nodes that map together. For instance the fragment `pt->x + pt->y` is consistent with `pt->a + pt->b` because there exists a bijective mapping between the field names: `x` $\leftrightarrow$ `a`, `y` $\leftrightarrow$ `b`. However, no such mapping exists between `pt->x + pt->y` and `pt->a + pt->a`. We do the same for function names.[5] *(Metric name: relaxed dependency-equivalent (consistency))*

2) 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.

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, 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-equivalence. With this mapping, we compare variables' names and types: variables that store the results of executing the mapped instructions map together. *(Metric names: variable name accuracy, variable type accuracy)*

UDT accuracy. One of the key challenges with decompiling real C code is reconstructing user-defined types. One novelty

[4]While this represents a common optimization, neural decompilers are trained to predict the original source code, so they typically learn to *undo* such optimizations rather than preserve them.

[5]There is no need for consistent (local) variable names since these manifest in dataflow dependencies.

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 both is challenging. That said, knowing and predicting a type's structure—the type of its fields, and their order—is still helpful to a reverse engineer, even if the names are imperfectly predicted. We therefore decompose UDT prediction accuracy, reporting the fraction of UDTs for which the structural layout matches (ignoring type and field names). When computing our metric "strict dependency-equivalentce (typechecks)", we use structural equivalence in determining type equivalence. *(Metric names: UDT variable nominal accuracy, UDT variable structural accuracy)*

In summary, unit test accuracy is an upper bound on semantic preservation; strict dependency-equivalence approximates a lower bound. Variable name accuracy and the three type accuracy measures quantify code improvements. Dependency-equivalence also approximates code improvement, as it captures structural similarity to the original.

C. Setup

Overall performance. The first research question compares IDIOMS overall performance (trained on the REALTYPE training set) to Nova [17] and LLM4Decompile [19] on both EXEBENCH and REALTYPE. We use the IDIOMS models trained by fine-tuning the 7-billion parameter version of CodeGemma [38] with a QLoRA [39] adapter, and compare to the 6.7b-sized versions of the related work.

Compiler optimizations. To evaluate performance across optimization levels (RQ2), we compare an IDIOMS model against similarly-sized versions of Nova and LLM4Decompile on REALTYPE code compiled at gcc levels O0–O3. We disable inlining to maintain evaluation rigor on REALTYPE's realistic code.[6] By merging multiple source functions, inlining creates a one-to-many mapping between binary and original/gold standard source functions. Boundaries between inlined functions become ambiguous after optimization, yielding no clear or unique ground truth. Inlining also alters structure such that the dependency graph isomorphism and variable correspondence metrics would break down, even for otherwise excellent predictions. Importantly, however, inlining poses *evaluation* challenges rather than decompilation challenges. Decompiling a function with inlined callees is simply decompiling a larger function, and REALTYPE includes various function sizes. If anything, we expect inlining may sometimes benefit UDT recovery by consolidating otherwise scattered context, but we can't be sure: it's possible that Idioms could perform worse on inlined code (since we can't evaluate on it).

Dataset creation did not always succeed at all optimization levels. For fair comparison, we include only functions that successfully decompiled at all levels (O0-O3). This reduces the test set by 21% for HexRays, but 76% for Ghidra. Because the resulting dataset is problematically small, for the Ghidra-based LLM4Decompile evaluation, we use the subset of Hex-Rays-decompilable functions that also decompiled in Ghidra. Thus, the Ghidra test sets for the compiler optimization results have somewhat different compositions. However, as these are still large test sets sampled from the same distribution, the impact on the aggregate results should be negligible.

[6]Prior work also evaluates on optimizations [22], [19], [17] but on simpler datasets with isolated functions, sidestepping inlining challenges.

Model size. To evaluate the degree to which our innovations are applicable to a variety of model types and sizes, we finetune IDIOMS models from five pretrained models:

- CodeQwen2.5 [40], 0.5 billion parameter version
- LLM4Decompile [19], 1.3 billion parameter version
- CodeGemma [38], 2 billion parameter version
- CodeGemma [38], 7 billion parameter version
- CodeLlama [41], 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.

We perform traditional finetuning on the smallest model, CodeQwen2.5-0.5b. Finetuning becomes computationally prohibitive for models above 1 billion parameters in size; we therefore leverage recent results using adapters and quantization (QLoRA [39] adapters), allowing for a high-fidelity, computationally tractable approximation of full finetuning. We provide additional training details in the Appendix.

Ablation. Overall IDIOMS performance is evaluated by training models on REALTYPE with all features. However, both the nature and size of training datasets can impact model performance. We evaluate the effect of both, as well as the effect of our design choices, by training and evaluating alternative models that allow for controlled comparisons:

- **exebench**: This experimental setting trains and evaluates a neural decompilation model on EXEBENCH, the most complex benchmark in prior work. Because EXEBENCH does not support interprocedural callgraphs, this version of IDIOMS still jointly predicts names and UDT definitions, but does so without neighboring context.
- **parity-exebench**: EXEBENCH's training set (2,383,839 functions) is substantially larger than REALTYPE (154,301 functions). To control for training set size in model performance, we subsampled EXEBENCH to create a smaller training set that matched REALTYPE in size for training.
- **functions-realtype**: To assess the effect of function context, we train a version of IDIOMS with only function context (i.e., *without* neighboring context) on the REALTYPE dataset. Comparing these results to the full IDIOMS results controls for the effect of the neighboring context design decision. Additionally, recall that models trained on EXEBENCH by necessity only provide function context. Thus, this setting varies from **parity-exebench** by dataset *composition* only. We compare these two settings to reveal the effect of training on EXEBENCH versus the more-realistic REALTYPE.

We perform these ablations across all five fine-tuned models.

(a) Relaxed dependency-equivalence (consistency)

	exebench	REALTYPE O0	O1	O2	O3
IDIOMS	**34.1**	**32.3**	**28.0**	**26.2**	**25.5**
LLM4Decompile	27.9	10.6	6.2	6.3	5.9
Nova	24.8	16.6	9.4	8.0	7.5

(b) Dependency-equivalence

	exebench	REALTYPE O0	O1	O2	O3
IDIOMS	**33.7**	**21.6**	**19.1**	**18.0**	**18.0**
LLM4Decompile	27.4	7.3	3.6	4.6	5.1
Nova	23.9	6.7	5.6	5.0	4.7

(c) Strict dependency-equivalence (typechecks)

	exebench	O0	O1	O2	O3
IDIOMS	**23.9**	**9.8**	**8.6**	**7.2**	**7.0**
LLM4Decompile	17.6	3.3	2.6	3.5	3.7
Nova	13.4	0.9	2.5	2.7	2.5

(d) passes EXEBENCH tests

	exebench
IDIOMS	**54.4**
LLM4Decompile	46.3
Nova	37.5

(e) Variable name accuracy

(f) Variable type accuracy

	exebench	O0	O1	O2	O3
IDIOMS	**20.6**	**19.8**	**18.7**	**17.9**	**17.8**
LLM4Decompile	14.7	3.4	3.2	3.9	3.5
Nova	12.9	4.5	3.2	3.5	3.5

(g) UDT variable nominal accuracy

	exebench	O0	O1	O2	O3
IDIOMS	**58.2**	**38.3**	**34.0**	**33.3**	**32.6**
LLM4Decompile	45.5	13.4	11.4	11.7	10.7
Nova	41.8	13.0	9.9	11.5	12.1

(h) UDT variable structural accuracy

	exebench	O0	O1	O2	O3
IDIOMS	**20.7**	**6.4**	**5.6**	**6.0**	**5.7**
LLM4Decompile	0.0	0.0	0.0	0.0	0.0
Nova	0.0	0.0	0.0	0.0	0.0

	exebench	O0	O1	O2	O3
IDIOMS	**34.5**	**15.0**	**14.7**	**13.4**	**12.3**
LLM4Decompile	0.0	0.0	0.0	0.0	0.0
Nova	0.0	0.0	0.0	0.0	0.0

TABLE II: Performance of IDIOMS, Nova [17] and LLM4Decompile [19]) on EXEBENCH [23] and a subset of REALTYPE that decompiled at all optimization levels. All values are percentages; higher is better. Nova and LLM4Decompile score 0 on UDT metrics because they do not predict UDTs. (UDT metric are computed on the set of UDT variables in the original code.)

IV. RESULTS

Table II shows results for RQ1, comparing IDIOMS performance to prior work on neural decompilation, and RQ2, evaluating the impact of compiler optimization levels. We discuss both questions in Section IV-A.

Results to support the remaining RQs are show in Table III, which is organized by model and by training configuration. The table is structured such that each column differs by one key experimental setting from adjacent settings, allowing for controlled comparisons. The **exebench** and **parity-exebench** columns differ by training set size (the first is larger); **parity-exebench** and **functions-realtype** differ by dataset type (EXEBENCH vs. REALTYPE) and thus the complexity and number of UDTs in both training and evaluation; **functions-realtype** and IDIOMS differ by the context provided to the model (decompiled function only vs. decompiled function and neighboring functions).

Section IV-B discusses the impact that realistic UDTs have on the complexity on the neural decompilation; Section IV-C addresses model size and scaling, Section IV-D evaluates the use of neighboring function context in IDIOMS design, and Section IV-E compares IDIOMS with existing type recovery work.

Note that, in general, there is high variance in UDT metrics for the EXEBENCH-based experiments because there very few UDTs in the `test_real` subset of EXEBENCH.

A. Decompilation performance (RQ1 and RQ2)

Table II shows performance of IDIOMS, LLM4Decompile, and Nova on EXEBENCH and REALTYPE, including performance at different compiler optimization levels.

Overall performance. IDIOMS substantially outperforms existing work. On EXEBENCH, IDIOMS scores 17-36% higher than LLM4Decompile and 38-78% better than Nova on all correctness metrics, while scoring similarly highly on code improvements. This improvement is despite the fact that the EXEBENCH experiments do not leverage IDIOMS' key differentiators. One possible reason is that Nova and LLM4Decompile both bear the hallmarks of being trained on code where the function names are left in the decompiled code (via debug information). In particular, both tend to copy the generic function name (e.g. `FUN_00100155` or canonicalized into `func0`) from the input to the output unchanged—a pattern learned when the input (decompiled code or assembly) and output (original code) have the same name in training. Indeed, sample training data linked to from the LLM4Decompile repository has function names in the decompiled code. Function names confer a large amount of information about the function to neural models [42], rendering the overall task substantially easier. Except where dynamic linking is required, function names in binaries are uncommon in practice, so we evaluate on code with the names removed. Other confounding factors include model architecture and input type—anecdotally, we find Hex-Rays' output to be better than Ghidra's, for instance.

However, the performance gaps are much larger on RE-

ALTYPE, where the innovations that differentiate IDIOMS are most relevant. Notably, Nova and LLM4Decompile score 0 on the UDT-related metrics (Tables IIh and IIg) because they do not predict any user-defined type definitions. But UDTs and the code that interacts with them are intrinsically linked. IDIOMS scores 95-205% higher than Nova and LLM4Decompile on correctness metrics—even the most permissive that does not require UDT definitions (Table IIa). In turn, variable name and type metric scores are low partially because there are many variables in the original code that don't correspond to anything in the nonequivalent predicted code.

Impact of compiler optimizations. Table II also delineates results by gcc optimization level; these reults entail training and testing on REALTYPE. In line with related work [17], [19], [22] on easier datasets, IDIOMS' performance degrades somewhat as optimization levels increase. There is a large drop at O1, and smaller drops with more optimizations. Still, IDIOMS performs much better at O3 than related work does at O0: 25.5% on relaxed dependency equivalence vs. 16.6%.

Takeaway: IDIOMS greatly outperforms state-of-the-art neural decompilers, especially on realistic code with UDTs. Performance drops sharply at O1, but higher levels of optimization have little additional effect.

B. RQ3: The Challenge of Real-World UDTs

Table III shows the results of our ablation study, where we causally show the impact of dataset size, dataset composition, and model context. The second and third columns (**parity-exebench** and **functions-realtype**) show models that are the same in terms of context provided and training dataset size. However, they differ in dataset complexity, as the **functions-realtype** configuration trains on REALTYPE, which has far more, and more complex, UDTs than EXEBENCH (Table I).

This change causes a substantial decrease in performance on all metrics except variable name accuracy. For dependency-equivalence (row 2 on each of Tables IIIa-IIIe), the drop is 38-42%, relative to performance on EXEBENCH. The drop is even steeper when type correctness is factored in (row 3): from 55%-68%. These data highlight the challenge that UDTs provide for real code. Operations on UDTs usually decompose into complex series of assembly or primitive source level instructions, especially when combined (e.g. `foo->bar->baz`). Reconstructing these higher-level abstractions is more difficult than recovering math operations or operations on arrays, as are common in EXEBENCH and HUMANEVAL-DECOMPILE. The absence of UDTs in existing benchmarks masks the value of joint code-UDT prediction. REALTYPE helps close the gap.

Takeaway: Code with UDTs is substantially more challenging to neurally decompile than code without; strict dependency-equivalence drops by 55%-68% with more realistic data.

C. RQ4: 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 models in the results in Table III. For instance, CodeQwen2.5-0.5b, our smallest model, produces dependency-equivalent code 30.7% on EXEBENCH. Meanwhile, CodeGemma-7b scores 33.7%, an 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 full idioms approach, in the IDIOMS column, follows a similar trend, albeit on a more difficult dataset (Section IV-B). CodeQwen2.5's dependency-equivalence score is 15.6%, though this drops to 7.2% when type correctness is factored in. Meanwhile, the two 7b models, CodeGemma-7b and CodeLlama-7b score 18.3, 8.3 and 18.4, 8.0, respectively. However, the gains in UDT variable structural accuracy increase more rapidly with model size than other metrics (see Section IV-D). The best IDIOMS model, CodeGemma-7b, recovers structurally accurate UDTs 15.1% of the time. To be counted as correct, a prediction must have identical fields, recursively (including field names for nominal struct accuracy). The UDTs in REALTYPE are very challenging: the mean recursive number of fields in its UDTs is 17.6 (Table I).

Takeaway: Larger models and training sets increase performance modestly across all models, including IDIOMS. The best IDIOMS models achieve dependency-equivalence scores of over 18% and UDT structural accuracy over 15% on real code and types.

D. RQ5: The Role of Neighboring Context

The last two columns of Table III show configurations that control for the utility of the neighboring function content: **functions-realtype** are models trained and tested on REALTYPE with only the decompiled function code as context, while IDIOMS is the full IDIOMS design including neighboring functions. Additional context in the form of neighboring functions improves performance for many metrics, especially for larger models. Interestingly, adding context does not have a significant impact on correctness, when measured with dependency-equivalence. This is not surprising, because the details necessary to predict code *operations* are all present in the function of interest (except for field names).

On the other hand, UDT accuracy increases, roughly, with model size, especially in terms of structural accuracy where names are ignored. As model sizes increases from 0.5b in Table IIIa to 7b in Table IIId, gains in UDT structural accuracy increase from 7% to 11% to 55% to 64%. CodeLlama (Table IIIe), while also a 7b model, scores only a 41% increase in UDT structural accuracy—there is clearly some variance stemming from the base pretrained model.

Small models also suffer more tradeoffs from attempting to handle the larger context. CodeGemma-7b and CodeLlama-7b each only suffer a drop in one metric (respectively, -3.5% relaxed dependency-equivalence and -13.8% in strict dependency-equivalence), while the smallest model, CodeQwen-0.5b, suffers decreases in 4 of 7 metrics.

(a) CodeQwen2.5-0.5b

		exebench	parity-exebench	functions-realtype	IDIOMS
Correctness	Relaxed dependency-equivalence (consistency)	30.8	26.0	24.2	22.3
	Dependency-equivalence	30.6	26.0	16.2	15.6
	Strict dependency-equivalence (typechecks)	20.4	17.0	7.1	7.2
	Passes EXEBENCH tests	44.4	32.1	–	–
Improvement	Variable name accuracy	18.8	15.2	13.8	13.6
	Variable type accuracy	54.4	45.6	32.9	30.4
	UDT variable nominal accuracy	11.1	6.9	3.2	3.7
	UDT variable structural accuracy	37.0	10.3	6.0	6.4

(b) LLM4Decompile-1.3b-v2

		exebench	parity-exebench	functions-realtype	IDIOMS
Correctness	Relaxed dependency-equivalence (consistency)	32.1	29.3	26.9	26.8
	Dependency-equivalence	31.9	29.1	18.0	17.3
	Strict dependency-equivalence (typechecks)	23.3	19.8	8.4	8.3
	Passes EXEBENCH tests	49.1	41.3	–	–
Improvement	Variable name accuracy	19.2	17.2	16.1	16.9
	Variable type accuracy	54.9	49.6	36.1	35.3
	UDT variable nominal accuracy	13.8	12.0	3.8	4.6
	UDT variable structural accuracy	41.4	28.0	10.0	11.1

(c) CodeGemma-2b

		exebench	parity-exebench	functions-realtype	IDIOMS
Correctness	Relaxed dependency-equivalence (consistency)	31.1	29.2	26.5	26.1
	Dependency-equivalence	30.9	29.0	17.0	18.1
	Strict dependency-equivalence (typechecks)	21.5	21.1	8.5	8.6
	Passes EXEBENCH tests	48.3	41.3	–	–
Improvement	Variable name accuracy	19.5	16.9	15.7	17.6
	Variable type accuracy	55.6	50.6	36.1	35.4
	UDT variable nominal accuracy	24.1	24.1	3.8	4.9
	UDT variable structural accuracy	55.2	41.4	7.4	11.5

(d) CodeGemma-7b

		exebench	parity-exebench	functions-realtype	IDIOMS
Correctness	Relaxed dependency-equivalence (consistency)	34.1	31.7	28.5	27.5
	Dependency-equivalence	33.7	31.3	18.1	18.3
	Strict dependency-equivalence (typechecks)	23.9	22.1	7.1	8.3
	Passes EXEBENCH tests	54.4	47.4	–	–
Improvement	Variable name accuracy	20.6	18.2	16.3	19.1
	Variable type accuracy	58.2	53.2	36.4	37.9
	UDT variable nominal accuracy	20.7	17.2	4.0	5.7
	UDT variable structural accuracy	34.5	44.8	9.2	15.1

(e) CodeLlama-7b

		exebench	parity-exebench	functions-realtype	IDIOMS
Correctness	Relaxed dependency-equivalence (consistency)	33.0	29.5	26.6	27.2
	Dependency-equivalence	32.7	29.1	17.8	18.4
	Strict dependency-equivalence (typechecks)	22.2	20.0	9.1	8.0
	Passes EXEBENCH tests	48.4	42.0	–	–
Improvement	Variable name accuracy	19.3	18.0	17.2	19.3
	Variable type accuracy	55.8	53.9	36.5	37.7
	UDT variable nominal accuracy	20.7	25.9	4.1	5.5
	UDT variable structural accuracy	41.4	48.1	10.0	14.1

TABLE III: Ablation study. All values are percentages; higher is better. Adjacent columns differ in one experimental condition.

```
struct hash_table {
    int size;
    struct hash_entry **ht;
    int (*cmp)(void *, void *);
    int (*hash)(void *);
};
struct hash_entry {
    void *key;
    void *value;
    struct hash_entry *next;
};
int hash_table_find(struct hash_table *ht,void *key){
    struct hash_entry *he;
    int hash_val = hash_table_hash(ht, key);
    int i = 0;
    for (he = hash_table_get(ht->ht, hash_val); he;
         he = hash_table_get(ht->ht, hash_val)) {
        if (i++ > ht->size) return -1;
        if (!ht->cmp(he, key)) break;
        hash_val = hash_table_next(ht, hash_val);
    }
    if (he) return hash_val;
    return -1;
}
```

Fig. 5: IDIOMS-functions' prediction on Figure 1b. The lack of context causes a subtle but substantial mistake. The version produced with neighboring context (Figure 1d) is correct.

(a) Type Accuracy

	coreutils		realtype	
	O0	O3	O0	O3
Idioms	56.0	55.5	48.7	46.7
BinSub	13.1	1.8	10.1	6.2
Retypd	14.2	1.7	10.8	7.3
TRex	27.2	23.5	28.2	25.6

(b) UDT Structural Accuracy

	coreutils		realtype	
	O0	O3	O0	O3
Idioms	7.4	4.5	13.3	10.9
BinSub	0.6	0.0	0.1	0.0
Retypd	0.3	0.0	0.2	0.2
TRex	0.0	0.0	1.7	1.4
TypeForge*	3.9	1.2	–	–

TABLE IV: IDIOMS compared to type recovery tools. For fair comparison, only function parameters are considered (unlike Tables II and III). *TypeForge only predicts UDTs and failed on REALTYPE, so we report results only on coreutils on UDT accuracy.

To see the difference that neighboring context can make, compare Figure 5, made without additional context, and Figure 1d, made with it. The function definition in Figure 5 looks almost correct relative to the original source, when allowances are made for differing but consistent function names. However, the types are slightly wrong: in addition to missing a current-capacity field, the `struct` definitions suggest that the table is backed by an array of linked lists—suggesting a separate-chaining collision avoidance strategy, not the robin-hood hashing actually used in the original code. The identifier names are in turn less reflective of the actual functionality presented in the code. IDIOMS-functions interprets the return value as a hash value, not an index into the hash table—likely because indexing into a linked list is uncommon. Crucially, *the evidence available in the decompiled code (Figure 1b) is consistent with both* IDIOMS-*functions' and* IDIOMS-*neighbors predictions*, but only IDIOMS-functions' prediction is incorrect relative to the original source and misleading. Details from the additional context help improve the prediction.

Takeaway: Neighboring context improves UDT accuracy with little to no downside, especially for larger models.

E. RQ6: Comparison With Type Recovery Tools

Reconstructing variable types is a fundamental component of decompilation. We compare IDIOMS with four existing standalone type recovery techniques: Retypd [43], BinSub [44], TRex [45], and TypeForge [46]. We evaluate on the binaries from `coreutils`—a common evaluation benchmark in type recovery work—and REALTYPE, at both O0 and O3. We limit our evaluation to function parameters, ignoring local variables, because (a) the Retypd[7] implementation we used and BinSub only emit only function signatures, and (b) it is challenging to compute ground truth for local variables in neural decompilation, even in correctly-predicted code (Sections III-B1 and V). Limiting predictions to function parameters presents little threat to validity; we compared the scores of TRex and TypeForge on all variables vs. parameter variables, and they are only on average 3.4% and 0.6% different (in absolute terms) for all types and UDTs, respectively.

Table IV shows the results of our comparison. IDIOMS greatly outperforms existing type recovery techniques, both in general and on UDTs specifically. Most type recovery techniques infer constraints from the binary and then solve those constraints to produce type predictions for each variable. The difficulty is that type reconstruction is usually *under-constrained*, which makes it challenging for deterministic approaches to decide what the emitted type should be. This is a significant limitation, as the scores of BinSub, Retypd, and TRex show: they are correct under 30% of the time: 10.1%, 10.8%, and 28.2%, on REALTYPE-O0 and 6.2%, 7.3%, and 25.6% accurate on REALTYPE-O3, compared with IDIOMS' 48.7% and 46.7%—at minimum 73% better, in relative terms. (BinSub and Retypd's especially poor scores on coreutils-O3 are because they often failed to produce output on this dataset.) Further, these three are only intraprocedural[8], analyzing one function at a time, meaning they, like prior neural decompilation techniques, fall afoul of the scattered evidence problem when trying to predict real-world UDTs: they correctly recover UDT types less than 2% of the time. IDIOMS is much more effective, scoring up to 13.3% on the same data.

TypeForge, like IDIOMS, *is* interprocedural, and accordingly outperforms the intraprocedural tools on UDTs. Like TypeForge, IDIOMS takes advantage of constraint-solving-

[7]The original implementation is unavailable, so we used a publicly available implementation built into the `angr` toolchain [47].

[8]`angr`'s [47] implementation of Retypd is intraprocedural.

based approaches by leveraging the output of a deterministic decompiler as input. TypeForge and IDIOMS also both use machine learning to address the underconstrained nature of type recovery. Where the techniques differ is in how they apply machine learning to handle this uncertainty: TypeForge allows models to choose between candidates in a double-elimination mechanism, whereas IDIOMS allows the model to generate type definitions as it sees fit in conjunction with the code.

Notably, all of TypeForge's UDT prediction accuracy comes from variables whose types are labeled as decompiler-inferred. Some decompilers, including Ghidra, can recognize standard library functions, look up the types of their parameters and return value, and propagate that information throughout the binary. We disabled Hex-Rays' equivalent feature when we were generating REALTYPE. Despite this advantage granted to TypeForge, IDIOMS still outperforms TypeForge on UDTs: 7.4% vs 3.9% and at O0 and 4.5% vs 1.2% at O3.

Takeaway: IDIOMS performs effective type recovery, scoring at least 73% better than prior work.

V. Discussion

Evaluation Strictness, or "Why are the scores so low?" In general, the scores in our evaluations are somewhat low *for all techniques*. For example, even the relaxed dependency-equivalence scores, which are less strict, are below 30% on the full REALTYPE dataset. This is a result of our decision to use strict evaluation metrics. Much of evaluating neural decompilation is about determining whether the decompiled code is semantically equivalent to its original, and how. Naturally, this is undecidable in the general case. We address this problem by using evaluation metrics that are strict and conservative (except unit tests), leaning towards soundness. Actual semantic equivalence likely exceeds what the metrics suggest. Dependency-based equivalence allows evaluation without compilation or test execution—which are difficult to do in this context—and helps evaluate variable names and types.

The name and type prediction metrics are particularly difficult to score well on. Only variable names or types that *exactly* match the ground truth are considered correct. As a prerequisite, the evaluation must determine which variable in the predicted code corresponds to which variable in the original code. To be conservative, if the function's semantics are incorrectly predicted, we assume we cannot find the corresponding ground truth variable, and thus the impacted names and types are considered incorrect by default. This is exacerbated by the fact that the sound but (necessarily) incomplete dependency-based equivalence technique can't always match variables even in equivalent functions. This is generally not a problem in traditional type inference and type recovery techniques, because they deal with compiled programs, and use registers, addresses and offsets for alignment. Meanwhile, it is difficult to compile neurally-decompiled code [48]. Even when it can be compiled, the memory layout may not match the original binary, such as when the same variables are declared in a different order (potentially impacting register placement) in otherwise functionally-equivalent code. This is why type accuracy scores are higher in Table IV than Tables II or III. In Table IV, the comparison is restricted to function parameters, for which the ground truth is trivially easy to find by stack offset.

Even when variables can be matched, the score is conservative. Many acceptable name variations exist (e.g., `length` vs. `len` vs. `size`), which strict name accuracy will not recognize. Relaxing metrics to allow common aliases could yield more optimistic results, though it's difficult to do this reliably. Some work uses lexical similarity metrics like character-error rate [3], [5], but this is problematic: 'minimum' and 'maximum' are lexically similar but have opposite meanings, while 'len' and 'size' are lexically different but interchangeable. VarCLR [49] encodes the semantics of variable names in an embedding space and measures the cosine similarity between embeddings, but this produces a unitless number which cannot be interpreted as a score (e.g. as a percentage).

Meanwhile, type accuracy scores (30-40% for IDIOMS) primarily reflect the difficulty in associating variables in predicted source code with those in ground-truth, the inherent ambiguities of having multiple type sizes that fit into a register (e.g., `int` vs `long`), and the prevalence of UDTs (26.4% of REALTYPE variables). Indeed, the latter fact explains most of the type accuracy performance differences observed between REALTYPE and EXEBENCH. REALTYPE's UDTs contain 17.6 fields (recursively) on average; this means that on average, 17.6 fields need to be predicted correctly for a type to be counted as correct. This is, of course, extremely challenging. Many type recovery techniques include metrics that accommodate for partial correctness, sometimes very generously. We discuss these further in the appendix.

That said, when evaluating all models in the same way, on the same data, IDIOMS greatly outperforms existing work.

Limitations and Threats. A key threat to validity with most work involving large language models is data leakage through pretraining. Most LLMs include code from GitHub, from which we also draw REALTYPE. 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, never mind the crucial decompiled-to-original mapping needed for neural decompilation; pretrained models are likely unfamiliar with it.

Our dataset is constructed of open source projects on GitHub that compile. It is possible that some decompilation targets, especially malware, may have systematic differences. Malware in particular is often *obfuscated*, or transformed in a way that makes it more difficult to understand. We view deobfuscation as an orthogonal problem. There are a variety of deobfuscation techniques [50], [51], [52], [53], [54], [55] that can de-obfuscate code before neurally decompiling it. 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 this to future work.

VI. Related work

Neural Decompilation. Early neural decompilation evolved from RNNs [24] through various architectures [25], [26], to

transformers [27]. The dominance of the transformer architecture [56] has driven more recent work to leverage its potential: SLaDe [22] outperforms prior work, but requires test cases for inference, which is unrealistic in most reverse engineering scenarios. LLM4Decompile [19] introduce a family of large causal transformer models with sizes in the billions of parameters, accepting either assembly or Ghidra output as input. Nova [17], which operates directly on assembly, uses hierarchical attention to adapt to and handle typically long sequences of assembly instructions. General-purpose LLMs like ChatGPT perform poorly at decompilation [22], [19], [17], likely because their training data contains little decompiled code.

Type Recovery. Type inference and type recovery [57], [58], [59], [45], [13], [60], [11], [12], [15], [43], [46], [44] have been studied extensively in computer security for over two decades. Generally, type recovery entails inferring and then solving constraints on the types of each variable. However, in practice, most types are *underconstrained*: there is not enough information to definitively determine a specific type. A modern trend is to *probabilistically* determine what the final type should be given the surrounding code [11], [60], [12], [13]. Increasingly, this has involved using LLMs [15], [46]. IDIOMS can be understood as a continuation of this trend—IDIOMS leverages the type recovery routines of the underlying decompiler, then uses an LLM to decide the final types.

VII. Conclusion

In this work, we introduce the IDIOMS family of neural decompilers. Unlike prior work, we recast neural decompilation as joint code and type definition recovery, and are the first to leverage interprocedural context. We build a dataset, REALTYPE, which contains 157,163 total functions and the definitions of their user-defined types. We train IDIOMS models on REALTYPE, and demonstrate state-of-the-art performance across a variety of metrics relative to other leading neural decompilers. We causally show that REALTYPE, with its complex UDTs, is a more difficult dataset than EXEBENCH, the most challenging existing dataset. We show that neighboring context helps address the scattered evidence problem and thus improves performance on UDTs. We've illustrated the difficulty and importance of joint code-type definition recovery, and have made an important step in addressing it.

Acknowledgment

This material is based upon work funded and supported by the Department of Defense under Contract No. FA8702-15-D-0002 with Carnegie Mellon University for the operation of the Software Engineering Institute, a federally funded research and development center. This material is based upon work supported by the National Science Foundation Graduate Research Fellowship under Grant No. DGE2140739. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the National Science Foundation.

References

[1] E. J. Schwartz, J. Lee, M. Woo, and D. Brumley, "Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring," in *Proceedings of the USENIX Security Symposium*, 2013.

[2] M. V. Emmerik, "Static single assignment for decompilation," Ph.D. dissertation, The University of Queensland, 2007.

[3] J. Lacomis, P. Yin, E. Schwartz, M. Allamanis, C. Le Goues, G. Neubig, and B. Vasilescu, "Dire: A neural approach to decompiled identifier naming," in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 628–639.

[4] V. Nitin, A. Saieva, B. Ray, and G. Kaiser, "Direct: A transformer-based model for decompiled variable name recovery," *NLP4Prog 2021*, p. 48, 2021.

[5] K. K. Pal, A. P. Bajaj, P. Banerjee, A. Dutcher, M. Nakamura, Z. L. Basque, H. Gupta, S. A. Sawant, U. Anantheswaran, Y. 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)*. IEEE Computer Society, 2024, pp. 152–152.

[6] X. Xu, Z. Zhang, Z. Su, Z. Huang, S. Feng, Y. Ye, N. Jiang, D. Xie, S. Cheng, L. Tan *et al.*, "Unleashing the power of generative model in recovering variable names from stripped binary," 2025.

[7] Y. David, U. Alon, and E. Yahav, "Neural reverse engineering of stripped binaries using augmented control flow graphs," *Proceedings of the ACM on Programming Languages*, vol. 4, no. OOPSLA, pp. 1–28, 2020.

[8] X. Jin, K. Pei, J. Y. Won, and Z. 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*, 2022, pp. 1631–1645.

[9] H. Kim, J. Bak, K. Cho, and H. 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*, 2023, pp. 951–965.

[10] D. Lehmann and M. 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*, 2022, pp. 410–425.

[11] Z. Zhang, Y. Ye, W. You, G. Tao, W.-c. Lee, Y. Kwon, Y. Aafer, and X. Zhang, "Osprey: Recovery of variable and data structure via probabilistic analysis for stripped binary," in *2021 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2021, pp. 813–832.

[12] C. Zhu, Z. Li, A. Xue, A. P. Bajaj, W. Gibbs, Y. Liu, R. Alur, T. Bao, H. Dai, A. Doupé *et al.*, "TYGR: Type inference on stripped binaries using graph neural networks," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 4283–4300.

[13] Q. Chen, J. Lacomis, E. J. Schwartz, C. Le Goues, G. Neubig, and B. Vasilescu, "Augmenting decompiler output with learned variable names and types," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 4327–4343.

[14] J. Xiong, G. Chen, K. Chen, H. Gao, S. Cheng, and W. Zhang, "Hext5: Unified pre-training for stripped binary code information inference," in *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2023, pp. 774–786.

[15] D. Xie, Z. Zhang, N. Jiang, X. Xu, L. Tan, and X. 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*, 2024, pp. 4554–4568.

[16] L. Dramko, J. Lacomis, E. J. Schwartz, B. Vasilescu, and C. Le Goues, "A taxonomy of c decompiler fidelity issues," in *33th USENIX Security Symposium (USENIX Security 24)*, 2024.

[17] N. Jiang, C. Wang, K. Liu, X. Xu, L. Tan, X. Zhang, and P. Babkin, "Nova: Generative language models for assembly code with hierarchical attention and contrastive learning," in *The Thirteenth International Conference on Learning Representations*, 2025.

[18] P. Hu, R. Liang, and K. Chen, "Degpt: Optimizing decompiler output with llm," 2024.

[19] H. Tan, Q. Luo, J. Li, and Y. Zhang, "LLM4Decompile: Decompiling binary code with large language models," in *Proceedings of the 2024 Conference on Empirical Methods in Natural Language Processing*, Y. Al-Onaizan, M. Bansal, and Y.-N. Chen, Eds. Miami, Florida, USA: Association for Computational Linguistics, Nov. 2024, pp. 3473–3487. [Online]. Available: https://aclanthology.org/2024.emnlp-main.203/

[20] D. Votipka, S. Rabin, K. Micinski, J. S. Foster, and M. L. Mazurek, "An observational investigation of reverse engineers' processes," in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 1875–1892.

[21] J. Caballero and Z. Lin, "Type inference on executables," *ACM Comput. Surv.*, vol. 48, no. 4, May 2016. [Online]. Available: https://doi.org/10.1145/2896499

[22] J. Armengol-Estapé, J. Woodruff, C. Cummins, and M. F. O'Boyle, "Slade: A portable small language model decompiler for optimized assembly," in *2024 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*, 2024, pp. 67–80.

[23] J. Armengol-Estapé, J. Woodruff, A. Brauckmann, J. W. d. S. Magalhães, and M. 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*, ser. MAPS 2022. New York, NY, USA: Association for Computing Machinery, 2022, p. 50–59. [Online]. Available: https://doi.org/10.1145/3520312.3534867

[24] D. S. Katz, J. Ruchti, and E. Schulte, "Using recurrent neural networks for decompilation," in *IEEE International Conference on Software Analysis, Evolution and Reengineering*, 4 2018, pp. 346–356.

[25] C. Fu, H. Chen, H. Liu, X. Chen, Y. Tian, F. Koushanfar, and J. Zhao, "Coda: An end-to-end neural program decompiler," *Advances in Neural Information Processing Systems*, vol. 32, 2019.

[26] Y. Cao, R. Liang, K. Chen, and P. Hu, "Boosting neural networks to decompile optimized binaries," in *Proceedings of the 38th Annual Computer Security Applications Conference*, 2022, pp. 508–518.

[27] I. Hosseini and B. Dolan-Gavitt, "Beyond the c: Retargetable decompilation using neural machine translation," 2022.

[28] Z. Hu. (2021) Ghcc. [Online]. Available: https://github.com/huzecong/ghcc

[29] D. Spinellis, Z. Kotti, and A. Mockus, "A dataset for github repository deduplication," in *Proceedings of the 17th international conference on mining software repositories*, 2020, pp. 523–527.

[30] A. Z. Broder, "On the resemblance and containment of documents," in *Proceedings. Compression and Complexity of SEQUENCES 1997 (Cat. No. 97TB100171)*. IEEE, 1997, pp. 21–29.

[31] D. Kocetkov, R. Li, L. Ben Allal, J. Li, C. Mou, C. Muñoz Ferrandis, Y. Jernite, M. Mitchell, S. Hughes, T. Wolf, D. Bahdanau, L. von Werra, and H. de Vries, "The stack: 3 tb of permissively licensed source code," *Preprint*, 2022.

[32] L. Gao, S. Biderman, S. Black, L. Golding, T. Hoppe, C. Foster, J. Phang, H. He, A. Thite, N. Nabeshima *et al.*, "The pile: An 800gb dataset of diverse text for language modeling," *arXiv preprint arXiv:2101.00027*, 2020.

[33] W. K. Wong, D. Wu, H. Wang, Z. Li, Z. Liu, S. Wang, Q. Tang, S. Nie, and S. Wu, "Decllm: Llm-augmented recompilable decompilation for enabling programmatic use of decompiled code," vol. 2, no. ISSTA, 2025.

[34] A. Holtzman, J. Buys, L. Du, M. Forbes, and Y. Choi, "The curious case of neural text degeneration," in *International Conference on Learning Representations*, 2020.

[35] E. K. Smith, E. Barr, C. Le Goues, and Y. Brun, "Is the cure worse than the disease? Overfitting in automated program repair," in *Proceedings of the 10th Joint Meeting of the European Software Engineering Conference and ACM SIGSOFT Symposium on the Foundations of Software Engineering (ESEC/FSE)*. ACM, 2015, pp. 532–543.

[36] L. Dramko, C. Le Goues, and E. J. Schwartz, "Fast, fine-grained equivalence checking for neural decompilers," *ACM Trans. Softw. Eng. Methodol.*, Oct. 2025. [Online]. Available: https://doi.org/10.1145/3772368

[37] W. Yang, S. Horwitz, and T. Reps, "Detecting program components with equivalent behaviors," University of Wisconsin-Madison Department of Computer Sciences, Tech. Rep., 1989.

[38] C. Team, H. Zhao, J. Hui, J. Howland, N. Nguyen, S. Zuo, A. Hu, C. A. Choquette-Choo, J. Shen, J. Kelley *et al.*, "Codegemma: Open code models based on gemma," *arXiv preprint arXiv:2406.11409*, 2024.

[39] T. Dettmers, A. Pagnoni, A. Holtzman, and L. Zettlemoyer, "Qlora: Efficient finetuning of quantized llms," *Advances in Neural Information Processing Systems*, vol. 36, 2024.

[40] B. Hui, J. Yang, Z. Cui, J. Yang, D. Liu, L. Zhang, T. Liu, J. Zhang, B. Yu, K. Lu *et al.*, "Qwen2. 5-coder technical report," *arXiv preprint arXiv:2409.12186*, 2024.

[41] B. Roziere, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, R. Sauvestre, T. Remez *et al.*, "Code llama: Open foundation models for code," *arXiv preprint arXiv:2308.12950*, 2023.

[42] X. Jin, J. Larson, W. Yang, and Z. Lin, "Binary code summarization: Benchmarking chatgpt/gpt-4 and other large language models," *arXiv preprint arXiv:2312.09601*, 2023.

[43] M. Noonan, A. Loginov, and D. Cok, "Polymorphic type inference for machine code," *SIGPLAN Not.*, vol. 51, no. 6, p. 27–41, Jun. 2016. [Online]. Available: https://doi.org/10.1145/2980983.2908119

[44] I. Smith, "Binsub: The simple essence of polymorphic type inference for machine code," in *Static Analysis*, R. Giacobazzi and A. Gorla, Eds. Cham: Springer Nature Switzerland, 2025, pp. 425–450.

[45] J. Bosamiya, M. Woo, and B. Parno, "{TRex}: Practical type reconstruction for binary code," in *34th USENIX Security Symposium (USENIX Security 25)*, 2025, pp. 6897–6915.

[46] Y. Wang, R. Liang, Y. Li, P. Hu, K. Chen, and B. Zhang, "Typeforge: Synthesizing and selecting best-fit composite data types for stripped binaries," in *2025 IEEE Symposium on Security and Privacy (SP)*, 2025, pp. 1–18.

[47] F. Wang and Y. Shoshitaishvili, "Angr - the next generation of binary analysis," in *2017 IEEE Cybersecurity Development (SecDev)*, 2017, pp. 8–9.

[48] M. Zou, A. Khan, R. Wu, H. Gao, A. Bianchi, and D. J. Tian, "D-Helix: A generic decompiler testing framework using symbolic differentiation," in *33rd USENIX Security Symposium (USENIX Security 24)*, 2024, pp. 397–414.

[49] Q. Chen, J. Lacomis, E. J. Schwartz, G. Neubig, B. Vasilescu, and C. Le Goues, "Varclr: Variable semantic representation pre-training via contrastive learning," in *Proceedings of the 44th International Conference on Software Engineering*, 2022, pp. 2327–2339.

[50] W. Dong, J. Lin, R. Chang, and R. Wang, "Cadecff: Compiler-agnostic deobfuscator of control flow flattening," in *Proceedings of the 13th Asia-Pacific Symposium on Internetware*, 2022, pp. 282–291.

[51] K. Coogan, G. Lu, and S. Debray, "Deobfuscation of virtualization-obfuscated software: a semantics-based approach," in *Proceedings of the 18th ACM conference on Computer and communications security*, 2011, pp. 275–284.

[52] R. David, L. Coniglio, M. Ceccato *et al.*, "Qsynth-a program synthesis based approach for binary code deobfuscation," in *BAR 2020 Workshop*, 2020.

[53] M. Liang, Z. Li, Q. Zeng, and Z. 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*. Springer, 2018, pp. 313–324.

[54] R. Tofighi-Shirazi, M. Christofi, P. Elbaz-Vincent, and T.-H. Le, "Dose: Deobfuscation based on semantic equivalence," in *Proceedings of the 8th Software Security, Protection, and Reverse Engineering Workshop*, 2018, pp. 1–12.

[55] G. You, G. Kim, S. Han, M. Park, and S.-J. Cho, "Deoptfuscator: Defeating advanced control-flow obfuscation using android runtime (art)," *IEEE Access*, vol. 10, pp. 61 426–61 440, 2022.

[56] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. u. Kaiser, and I. Polosukhin, "Attention is all you need," in *Advances in Neural Information Processing Systems*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds., vol. 30. Curran Associates, Inc., 2017. [Online]. Available: https://proceedings.neurips.cc/paper_files/paper/2017/file/3f5ee243547dee91fbd053c1c4a845aa-Paper.pdf

[57] J. Lee, T. Avgerinos, and D. Brumley, "Tie: Principled reverse engineering of types in binary programs," 2011.

[58] A. Slowinska, T. Stancescu, and H. Bos, "Howard: A dynamic excavator for reverse engineering data structures." in *Network and Distributed System Security Symposium (NDSS)*, 2011.

[59] Z. Lin, X. Zhang, and D. Xu, "Automatic reverse engineering of data structures from binary execution," in *Proceedings of the 11th Annual Information Security Symposium*, 2010.

[60] J. He, P. Ivanov, P. Tsankov, V. Raychev, and M. Vechev, "Debin: Predicting debug information in stripped binaries," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, 2018, pp. 1667–1680.

[61] I. Loshchilov and F. Hutter, "SGDR: Stochastic gradient descent with warm restarts," in *International Conference on Learning Representations*, 2017.

APPENDIX: TRAINING DETAILS

We used the following conventions for training models in our experiments. 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 [61] starting from a learning rate of 5×10^{-5}. We use a batch size of 64.

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. We used a standard train/validate/test approach in our initial experiments to ensure that overfitting did not occur.

APPENDIX: OTHER TYPE RECOVERY METRICS

In our evaluation, we use type accuracy, and variants of it restricted to UDTs, to measure the effectiveness of IDIOMS' and other techniques' type recovery capabilities. If the type matches the type in the original code then it is correct; otherwise, it is incorrect. (We perform the type name normalization described in Section II-B so that each unique size/bitpattern/signedness combination has a unique name.) For UDT structural accuracy, we ignore identifier names for type tags and fields and pay attention only to the types and orders of the fields. However, many type recovery papers do not use type accuracy. Instead, there are a variety of different metrics that assess partial correctness which widely vary between papers. For instance, of the four type recovery techniques we compare with—Retypd [43], BinSub [44], TRex [45], and TypeForge [46]—only two share a metric (Retypd and BinSub), which is based on the internal type lattice that those techniques use, and even then BinSub uses a modified version. Choice of metric is especially impactful for UDTs, which are particularly difficult to recover correctly but are particularly important.

To better contextualize IDIOMS' effectiveness in predicting UDTs, we evaluate IDIOMS and the other type recovery techniques on TypeForge's [46] *Composite Data Structure Identification* and *Layout Recovery* evaluation metrics for UDTs.

Composite Data Structure Identification measures how well a technique determines that a given variable is a composite data structure: a UDT, an array, or a pointer to a composite data structure. It is expressed in terms of precision and recall, where

$$TP = \text{is\_composite}(p) \wedge \text{is\_composite(g)} \quad (1)$$

$$FP = \text{is\_composite}(p) \wedge \neg\text{is\_composite(g)} \quad (2)$$

$$FN = \neg\text{is\_composite}(p) \wedge \text{is\_composite(g)} \quad (3)$$

and where p is a predicted type, g is a ground truth type, TP is a true positive, FP a false positive, and FN a false negative. Notably this metric does not consider the *composition* of the composite types, just whether or not they are identified as composite data structures.

Layout Recovery is defined only for `struct`s, which make up the vast majority of UDTs in practice. In layout recovery, the structs are each represented as sets of 2-tuples, where the first element of the tuple is the offset of the field, and the second is the size of its type. If p represents such a set for the predicted struct, and g represents such a set for the ground truth struct, then the metrics' precision and recall are given by:

$$\text{Precision} = \frac{\sum |p \cap g|}{\sum |p|} \quad (4)$$

$$\text{Recall} = \frac{\sum |p \cap g|}{\sum |g|} \quad (5)$$

Notably, the identity of the type (e.g. `float` vs `int`) does not matter, only the size of the type. Importantly, all pointers have the same size on any given platform, regardless of the target type to which they point.

The results are shown in Table V. Table Va shows scores on composite data structure identification. IDIOMS generally has the highest precision, recall, and F1 scores, though it trails BinSub and Retypd slightly in precision at O0. BinSub and Retypd, however, are very conservative—BinSub correctly identifies at most 25% of composite types, and Retypd identifies less than 50%. Thus, they achieve much lower F1 scores. (BinSub and Retypd fail to produce output on a majority of functions in coreutils-O3, which is the reason for their especially low recall scores on that dataset.)

Table Vb shows scores on the layout recovery metric. IDIOMS' precision is relatively low because of the way in which IDIOMS can be wrong. In particular, sometimes incorrect `struct` definitions feature degenerate repeating patterns of the same fields over and over (see Section III-A), which can greatly inflate the metric's denominator, leading to a lower score. While we filter out extra degenerate struct definitions (which by nature are not used in the code), struct definition predictions—those that are actually used in the code as the type of a variable—can have extra degenerate repeating patterns of fields. In other words, when IDIOMS is wrong, it can be *very* wrong. On the whole, however, it is generally the best at identifying fields' types, and so receives the highest F1 score except on coreutils-O0, where it is edged out by TypeForge (42.9 vs 39.1). Both interprocedural techniques, IDIOMS and TypeForge, greatly outperform the intraprocedural techniques, which struggle with the scattered evidence problem. TRex,

(a) TypeForge-style Composite Data Structure Identification

	coreutils-O0			coreutils-O3			realtype-O0			realtype-O3		
	Prec.	Rec.	F1	Prec.	Rec.	F1	Prec.	Rec.	F1	Prec.	Rec.	F1
Idioms	63.2	83.9	72.1	60.8	75.0	67.1	88.6	83.6	86.0	88.5	78.0	82.9
BinSub	66.7	20.5	31.4	22.2	0.8	1.5	89.7	23.1	36.7	80.9	23.1	36.0
Retypd	77.0	45.7	57.3	57.1	3.2	6.0	86.9	38.9	53.7	82.2	28.7	42.5
TRex	56.3	44.1	49.5	31.9	42.1	36.3	81.4	42.5	55.8	71.2	50.4	59.0
TypeForge	47.9	42.2	44.9	39.4	34.1	36.6	–	–	–	–	–	–

(b) TypeForge-style Struct Layout Recovery (On All Predictions)

	coreutils-O0			coreutils-O3			realtype-O0			realtype-O3		
	Prec.	Rec.	F1	Prec.	Rec.	F1	Prec.	Rec.	F1	Prec.	Rec.	F1
Idioms	41.3	37.1	39.1	42.5	30.4	35.4	42.7	26.1	32.4	41.9	23.3	30.0
BinSub	72.9	2.2	4.2	50.0	0.1	0.1	72.0	3.9	7.4	62.9	3.3	6.3
Retypd	65.6	6.8	12.4	80.0	0.2	0.5	79.7	8.3	15.1	76.3	5.8	10.7
TRex	69.6	3.3	6.4	47.1	2.0	3.8	72.0	4.5	8.4	61.3	3.3	6.3
TypeForge	78.8	29.5	42.9	60.7	18.7	28.6	–	–	–	–	–	–

(c) TypeForge-style Struct Layout Recovery (On Only Predictions that are Structs)

	coreutils-O0			coreutils-O3			realtype-O0			realtype-O3		
	Prec.	Rec.	F1	Prec.	Rec.	F1	Prec.	Rec.	F1	Prec.	Rec.	F1
Idioms	41.3	45.2	43.1	42.5	43.4	42.9	42.7	37.4	39.9	41.9	37.7	39.7
BinSub	72.9	13.9	23.3	50.0	50.0	50.0	72.0	35.0	47.1	62.9	29.6	40.3
Retypd	65.6	23.4	34.5	80.0	8.9	16.0	79.7	46.5	58.8	76.3	43.2	55.2
TRex	69.6	32.5	44.3	47.1	32.4	38.4	72.0	59.7	65.3	61.3	48.1	53.9
TypeForge	78.8	61.3	68.9	60.7	44.2	51.2	–	–	–	–	–	–

TABLE V: IDIOMS and the type recovery techniques from Section IV-E, evaluated on TypeForge's metrics.

BinSub, and Retypd never score more than 10% on recall, meaning they correctly identify the offsets and sizes of very few fields across all `struct` variables.

Table Vc is effectively a combination of Tables Va and Vb. This table shows layout recovery for only the variables which are predicted `struct`s: this version of the metric asks, "given that a predicted variable is a `struct`, how likely is that `struct` definition to match the ground truth (in terms of type sizes at each offset)?" Even the intraprocedural techniques, BinSub, Retypd, and TRex, do very well here because they are extremely conservative in predicting variables `struct`s, as the low recall values in Tables Vb indicate. (Scores are higher in Table Va because composite data structure identification includes arrays.) By only predicting structs in the scenarios where it is easiest to identify and predict them, they can score well when the dataset is filtered this way. TypeForge, which is also more conservative than IDIOMS in producing predictions for UDTs (it identifies composite data structures at about half the rate as IDIOMS, as shown in the recall columns of Table Va and has lower recall scores in Table Vb) also benefits.

APPENDIX: SAMPLE PROMPT

The following is a sample IDIOMS prompt corresponding to the example in Figure 1d.

```
__int64 __fastcall func4(__int64 a1, __int64 a2)
{
int v2; // eax
__int64 result; // rax
int v4; // [rsp+10h] [rbp-10h]
unsigned int v5; // [rsp+14h] [rbp-Ch]
__int64 i; // [rsp+18h] [rbp-8h]

v5 = func2(a1, a2);
v4 = 0;
for ( i = func1(*(_QWORD *)(a1 + 8), v5); i; i = func1(*(_QWORD *)(a1 + 8), v5) )
{
v2 = v4++;
if ( v2 > *(_DWORD *)a1 )
return 0xFFFFFFFFLL;
if ( !(*(unsigned int (__fastcall **)(__int64, __int64))(a1 + 24))(i, a2) )
break;
v5 = func3((_DWORD *)a1, v5);
}
if ( i )
result = v5;
else
result = 0xFFFFFFFFLL;
return result;
}

__int64 __fastcall func2(__int64 a1, __int64 a2)
{
return (unsigned int)((*(int (__fastcall **)(__int64))(a1 + 16))
                                  (a2) % *(_DWORD *)a1);
}

__int64 __fastcall func1(__int64 a1, int a2)
{
__int64 result; // rax

if ( a2 < *(_DWORD *)a1 )
result = *(_QWORD *)(*(_QWORD *)(a1 + 8) + 8LL * a2);
else
result = 0LL;
return result;
}

__int64 __fastcall func3(_DWORD *a1, int a2)
{
return (unsigned int)((a2 + 1) % *a1);
}

__int64 __fastcall func5(__int64 a1, __int64 a2)
{
__int64 result; // rax
int v3; // [rsp+1Ch] [rbp-4h]

v3 = func4(a1, a2);
if ( v3 == -1 )
result = 0LL;
else
result = func1(*(_QWORD *)(a1 + 8), v3);
return result;
}

__int64 __fastcall func8(__int64 a1, __int64 a2)
{
int i; // eax
__int64 v4; // [rsp+10h] [rbp-20h]
int v5; // [rsp+1Ch] [rbp-14h]
__int64 v6; // [rsp+20h] [rbp-10h]
int v7; // [rsp+2Ch] [rbp-4h]

v6 = *(_QWORD *)(a1 + 8);
v7 = func4(a1, a2);
if ( v7 == -1 )
return 0xFFFFFFFFLL;
for ( i = func3((_DWORD *)a1, v7); ; i = func3((_DWORD *)a1, v7) )
{
v5 = i;
v4 = func1(*(_QWORD *)(a1 + 8), i);
if ( !v4 || !(unsigned int)func6(v5, (_DWORD *)a1, v4) )
break;
func0(v6, v7, v4);
v7 = func3((_DWORD *)a1, v7);
}
--*(_DWORD *)(a1 + 4);
func0(v6, v7, 0LL);
return 0LL;
}

__int64 __fastcall func6(int a1, _DWORD *a2, __int64 a3)
{
__int64 result; // rax
int v4; // [rsp+2Ch] [rbp-4h]

v4 = func2((__int64)a2, a3);
if ( v4 > a1 )
result = (unsigned int)(a1 - v4 + *a2);
else
result = (unsigned int)(a1 - v4);
return result;
}

__int64 __fastcall func7(__int64 a1, __int64 a2)
{
int v3; // ebx
__int64 v4; // [rsp+0h] [rbp-40h]
__int64 v5; // [rsp+10h] [rbp-30h]
__int64 i; // [rsp+20h] [rbp-20h]
int v7; // [rsp+2Ch] [rbp-14h]

v4 = a2;
v7 = func2(a1, a2);
v5 = *(_QWORD *)(a1 + 8);
if ( *(_DWORD *)(a1 + 4) == *(_DWORD *)a1 )
return 0xFFFFFFFFLL;
for ( i = func1(v5, v7); i; i = func1(v5, v7) )
{
v3 = func6(v7, (_DWORD *)a1, i);
if ( v3 < (int)func6(v7, (_DWORD *)a1, v4) )
{
func0(*(_QWORD *)(a1 + 8), v7, v4);
v4 = i;
}
v7 = func3((_DWORD *)a1, v7);
}
func0(v5, v7, v4);
++*(_DWORD *)(a1 + 4);
return 0LL;
}

__int64 __fastcall func0(__int64 a1, int a2, __int64 a3)
{
while ( a2 >= *(_DWORD *)a1 )
{
if ( (unsigned int)gap_extend(a1) == -1 )
return 0xFFFFFFFFLL;
}
*(_QWORD *)(8LL * a2 + *(_QWORD *)(a1 + 8)) = a3;
return 0LL;
}##func4@@
```