REBENCH: A Procedural, Fair-by-Construction Benchmark for LLMs on Stripped-Binary Types and Names

Anonymous Author(s)

Abstract

Large Language Models (LLMs) have achieved remarkable progress in recent years, driving their adoption across a wide range of domains, including computer security. In reverse engineering, LLMs are increasingly applied to critical tasks such as function and variable name recovery and type inference. However, despite the rapid growth of research in this area, progress has been hindered by the absence of a standardized dataset. Existing studies rely on disparate datasets, preprocessing pipelines, and evaluation metrics, making fair comparisons between approaches difficult and obscuring a clear understanding of LLM capabilities in binary analysis. To address these challenges, we present REBENCH, a comprehensive benchmark dataset for evaluating LLMs on binary reverse engineering tasks. REBENCH consolidates a superset of existing datasets, comprising hundreds of millions of lines of source code and a diverse collection of binaries spanning multiple architectures and optimization levels. REBENCH adopts a knowledge-base-driven methodology that stores byte-level stack information to generate ground truth, ensuring that task difficulty is preserved while maintaining universal applicability. This design enables fair evaluation across tasks while avoiding simplifications that could bias results. As a use case, we apply REBENCH to measure the reverse engineering performance of one LLM and the result demonstrates difficulties in complex tasks.

CCS Concepts

• **Security and privacy** → **Software reverse engineering**; • **Computing methodologies** → *Machine learning*.

Keywords

Function name recovery; variable name recovery; variable type inference; large language model; binary reverse engineering;

ACM Reference Format:

Anonymous Author(s). 2026. REBENCH: A Procedural, Fair-by-Construction Benchmark for LLMs on Stripped-Binary Types and Names. In . ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 Introduction

Large Language Models (LLMs) have recently gained significant attention due to rapid advances in model architectures, training techniques, and the increasing availability of large-scale datasets. Built on transformer architectures [21], LLMs have demonstrated remarkable success in natural language processing (NLP) tasks such as translation and question answering, with widely known examples including OpenAI's ChatGPT [16]. The adoption of transfer learning [20] has further expanded their applications across diverse domains, including healthcare [9, 15] and finance [26], where they have improved both efficiency and automation.

In the field of computer security, researchers are increasingly exploring LLMs for binary reverse engineering tasks, including function summarization [11], symbol name recovery in stripped binaries [10, 22, 24], and type inference, as LLMs can interpret human-readable representations, such as source code. These developments highlight the potential of LLMs to advance program analysis and support security-critical activities such as malware analysis, vulnerability discovery, and software maintenance. Although early results are promising, evaluating the effectiveness of LLMs in this domain remains a fundamental challenge.

The conventional approach to applying LLMs in reverse engineering involves fine-tuning pre-trained models on task-specific datasets. Although this strategy can substantially improve performance, it also introduces variability depending on the goals and experimental setups of individual studies. For example, SYMGEN [10] reports an F1 score of 0.351 on an x64 dataset when reproducing ASMDEPICTOR [13], which itself reports only 0.05 under the same comparison. Meanwhile, ASMDEPICTOR claims a considerably higher F1 score of 0.715 on its own evaluation dataset, revealing significant inconsistencies across studies.

A key barrier to resolving such discrepancies is the absence of a standardized dataset. Existing works evaluate models on heterogeneous datasets covering different processor architectures (e.g., x86, ARM, MIPS), compiler optimization levels (e.g., `O0`–`O3`), and program families (e.g., coreutils, binutils). Even when datasets overlap, inconsistencies arise from differences in preprocessing (e.g., assembly vs. decompiled code), naming conventions (e.g., memory vs. mem), and evaluation metrics. For instance, some studies consider only exact string matches when evaluating predicted variable names, while others award partial credit for semantically similar predictions. This lack of consistency prevents the community from fairly and systematically assessing progress.

To address these challenges, we present REBENCH, the first comprehensive dataset for evaluating LLMs on binary reverse engineering tasks. REBENCH is designed to be broad, standardized, and extensible. It consolidates datasets previously used in reverse engineering research [12, 18, 23], covering four major processor architectures (x86, x64, ARM 32-bit, MIPS 32-bit) and multiple compiler optimization levels (`O0`–`O3`). To ensure compatibility across analysis pipelines, we convert all binaries into a universal format using a general-purpose decompiler. Crucially, REBENCH introduces a knowledge-base-driven data normalization process that preserves task difficulty while ensuring consistent ground-truth labeling across datasets. After eliminating both duplicated and semantically trivial

Conference'17, Washington, DC, USA

ACM ISBN 978-x-xxxx-xxxx-x/YYYY/MM
https://doi.org/10.1145/nnnnnnn.nnnnnnn

functions, REBENCH contains 29.94 million lines of decompiled code on the x64 architecture alone.

Contributions. We make the following contributions.

- **Development of REBENCH:** We introduce REBENCH, a large-scale benchmark dataset normalized into a universal input format, enabling fair evaluation of LLMs in binary reverse engineering.
- **Reflection of Real-world Experiment Setup:** REBENCH preserves the inherent difficulty of reverse engineering by closely following real-world experimental settings, avoiding oversimplifications that could cause bias in evaluation.
- **Completeness of Ground Truth:** We construct a knowledge base by leveraging byte-level stack layout information, ensuring complete and reliable ground truth that overcomes limitations of existing decompilers.

2 Background & Motivation

2.1 Problem Definition

The field of binary analysis is rapidly evolving, driven in part by the growing application of LLMs. Establishing a standardized benchmark to evaluate LLMs in this domain is not only beneficial but essential for understanding their true capabilities. In this work, we focus on two fundamental reverse engineering tasks, name recovery and type inference, both of which are especially challenging in the context of stripped binaries, where symbol information has been removed. This scenario closely mirrors real-world reverse engineering tasks encountered in security-critical domains such as malware analysis and vulnerability research.

- **Name Recovery** seeks to predict human-readable, semantically meaningful names for functions, arguments, and variables, thereby improving code comprehension.
- **Type Inference** aims to identify the data types of variables, return values, and function arguments, which is critical for reasoning about program semantics.

2.2 Related Works

Binary reverse engineering involves analyzing and reconstructing machine code to understand program behavior, structure, and functionality. In recent years, research in this area has increasingly incorporated machine learning techniques, and more recently, LLMs. We provide an overview of related work, focusing on the two primary tasks addressed in this paper, name recovery and type inference, along with the datasets commonly used to evaluate them. We also briefly introduce additional reverse engineering tasks to illustrate the breadth of research in this field.

(I) Type Inference. Several works have explored automatic type inference in binaries:

- STATEFORMER [17] employs generative state modeling for type inference across 33 datasets, evaluating performance via exact name matching.
- DIRTY [3] uses a transformer-based model for type inference, restricting its dataset to binaries containing fewer than one million decompiled functions.
- OSPREY [27] applies probabilistic techniques to two benchmarks comprising 106 programs, with outcomes also measured by exact name matching.
- DEBIN [7] employs structured prediction algorithms for type inference, evaluating on 9,000 binaries, although the specific list of binaries is not detailed.

(II) Name Recovery. Recovering meaningful function and variable names has been advanced significantly by machine learning:

- SYMLM [12] leverages transformer models for function name recovery across 26 datasets and employs CodeWordNet to compute semantic distances between names.
- NERO [4], uses sequential models for name recovery across 99 datasets and assesses performance through exact name matching.
- VARBERT [2] performs exact name matching on the DIRE [14], which contains 3.1 million decompiled functions collected from GitHub.
- RESYM [22] and SYMGEN [10] fine-tune models to recover variable and function names via decompiled code, respectively.

(III) Others. Two recent works generate function summaries from decompiled code using LLM [11, 19]. Both operate directly on unmodified decompiled code and focus on a single task, making them comparatively simpler than multi-task approaches.

2.3 Observations

Our analysis of existing research underscores the critical need for a standardized dataset for evaluating LLMs on reverse engineering tasks. Without such a dataset, ensuring clear, consistent, and fair comparisons of model performance is difficult. We identify three major issues in current practices: variability in datasets, inconsistency in evaluation metrics, and incompleteness of decompiler output.

Observation 1: Variability in Datasets. A comprehensive review of recent studies reveals that the datasets used for evaluating LLMs in reverse engineering vary significantly in both scope and composition. Some works rely on small, narrowly defined datasets, while others evaluate across many datasets, reflecting considerable diversity in data sources. For example, DEEPBINDIFF [5] evaluates its framework on three well-known binary sets: coreutils, diffutils, and findutils. In contrast, SYMLM and SYMGEN incorporate 27 datasets, forming a superset of DEEPBINDIFF's collection but excluding ImageMagick, a dataset frequently used in other studies [18, 25].

This inconsistency creates a major barrier to progress. The lack of a standardized dataset makes it difficult to compare results across studies meaningfully, limiting collaborative research and slowing real-world applicability. Moreover, architectural coverage also varies across studies: DEBIN targets three architectures (x86, x64, ARM), while SYMLM extends this to include MIPS. Such discrepancies highlight the fragmented nature of current evaluation practices. A universal benchmark dataset would resolve these issues by providing a consistent foundation for evaluation, enabling objective model comparison, and supporting the training of future security researchers in binary analysis.

Observation 2: Inconsistency in Evaluation Metrics. Most existing studies evaluate LLM performance using traditional metrics such as precision, recall, and F1 score, the majority of which rely on exact

string matches between predicted outputs and ground-truth labels. However, this purely syntactic approach fails to capture semantic similarity, leading to systematic underestimation of LLM capabilities. For instance, Feitelson et al. [6] found that the probability of two developers independently selecting the same name for an identical function is only 6.9%. Moreover, developers frequently abbreviate or modify names for conciseness [8]. In such cases, string matching unfairly penalizes predictions that are semantically correct but syntactically different. For example, `compute` and `calculate` are semantically equivalent, yet exact-match criteria would label the prediction as incorrect.

These shortcomings are evident in DEBIN, which enforces strict string-based evaluation, overlooking meaningful semantic matches. By contrast, SYMLM and NERO adopt semantic-aware evaluation strategies, enabling a more accurate assessment of model performance. This disparity highlights the critical need for evaluation frameworks that incorporate semantic similarity, whether through embeddings, lexical resources, or domain-specific keywords, rather than relying solely on exact string comparison. Addressing this gap is essential for building reliable and realistic benchmarks for LLMs in reverse engineering.

Observation 3: Incompleteness of Decompiler Output. State-of-the-art decompilers still exhibit errors when generating decompiled code, particularly in reconstructing data structures, largely due to the absence of debug symbols. As a result, decompiled code generated with debug symbols often differs substantially from that produced without them. Prior work [2] has therefore relied on binaries containing debug symbols to generate decompiled code. While this approach improves the accuracy of the ground truth, it also simplifies the reverse engineering task, since the resulting code preserves richer semantic information than code derived from stripped binaries. For instance, developers can readily identify variables within a data structure and infer its components from debug-enriched output. However, this setup does not reflect real-world scenarios, where debug symbols are typically unavailable. Consequently, it is essential to select binaries that both mirror real-world conditions and enable the construction of complete, reliable ground truth.

3 REBENCH Design

Selecting appropriate datasets and compiling source code into executable binaries is a crucial step in preparing a usable benchmark. Since LLM-based methods require structured input such as assembly code or decompiled code, REBENCH abstracts symbols from these representations to construct a universal input format. This abstraction ensures flexibility, allowing the dataset to be customized for diverse downstream tasks while preserving the difficulty of real-world reverse engineering challenges.

3.1 Dataset Collection

To ensure the quality, representativeness, and usability of REBENCH, we adopt a principled dataset selection process guided by the following criteria:

- **Coverage of Existing Work.** REBENCH collects datasets from recent reverse engineering studies to maximize relevance and inclusivity. This ensures that the benchmark reflects the diversity of prior research and enables direct comparison with existing approaches.
- **Architectural and Compiler Optimization Diversity.** We include binaries compiled for different architectures and optimization levels, mimicking real-world scenarios where binary analysis must handle code from heterogeneous platforms and toolchains.
- **Accessibility.** We prioritize open, publicly available datasets, making REBENCH easily adoptable by both academic researchers and practitioners. Accessibility encourages collaboration and ensures reproducibility.
- **Popularity.** We select projects that are widely used and recognized in the community. Popular datasets frequently serve as baselines in reverse engineering studies, increasing the utility and credibility of REBENCH.
- **Ethical Sourcing and Licensing.** We prioritize ethical considerations in dataset acquisition to ensure responsible and transparent data usage, respecting privacy and intellectual property rights. We select datasets with permissive licenses that allow redistribution, promoting legal and ethical use and facilitating broader adoption of the benchmark.

Following these guidelines, we conducted a comprehensive review of existing work and surveyed open-source projects. Our focus is on programs implemented in C or C++, the most widely used languages in system-level programming and reverse engineering. To maintain openness and reproducibility, we exclude commercial or proprietary datasets, such as SPEC2017, that restrict redistribution. We also filter out binaries lacking debug symbols, such as firmware without source code, since ground-truth symbol information is necessary for constructing reliable evaluation datasets.

3.2 Dataset Compilation

Compiling the selected datasets requires manual effort due to diverse build requirements and dependencies. To ensure consistency, we adopt the standard `configure` and `make` workflow, which is widely used across open-source projects. The `configure` process generates files such as `Makefile` based on the given configurations, and the `make` process compiles the binaries. This approach provides the flexibility to compile each dataset for multiple target architectures (x64, x86, ARM 32-bit, and MIPS 32-bit) and optimization levels (`O0`–`O3`). For each dataset, we generate two sets of binaries that serve distinct purposes:

- **Debug-symbol Binaries.** These binaries preserve ground-truth information such as variable names, function names, and types. They are used to construct the knowledge base and to map predictions back to the ground truth.
- **Stripped Binaries.** These binaries have all symbol information removed, reflecting real-world scenarios where reverse engineers must infer missing information. They provide the decompiled or assembly code inputs supplied to LLMs in downstream tasks.

3.3 Dataset Normalization

The dataset normalization process is a non-trivial but essential step in building REBENCH. Our goal is to design a universal input format applicable to diverse reverse engineering tasks while preserving the inherent difficulty of stripped binary analysis. Since stripped binaries lack symbol information, we must systematically collect

and reconstruct ground truth for each symbol before normalizing the dataset. All examples described in this section are generated using the Ghidra [1] decompiler.

Input Format Selection. Using raw binary code (i.e., sequences of `0`s and `1`s) is impractical for LLMs, as it carries no explicit semantic information. Instead, following prior work, we adopt decompiled code and assembly code as input representations, since LLMs can leverage the richer vocabulary of keywords present in these formats, a key distinction from traditional machine learning techniques:

- **Assembly Code** provides low-level semantics (e.g., instruction mnemonics, registers) and is particularly useful for tasks such as primitive type inference and function name recovery [12]. However, identifying variables within assembly code requires additional analysis techniques. Its limited token vocabulary also makes it amenable to token-based machine learning methods.
- **Decompiled Code** offers high-level, human-readable pseudocode that exposes constructs such as functions, variables, and control flow [10, 22]. While imperfect, it provides richer semantic context, making it well suited for both name recovery and type inference.

Together, these representations enable REBENCH to support a wide range of reverse engineering tasks while maintaining fidelity to real-world workflows.

Decompiled Code Selection and Challenges. Decompiled code selection is a pivotal phase of our methodology, as it directly dictates the realism of the benchmark and the integrity of the ground truth. A primary challenge lies in ensuring that the normalization process does not inadvertently trivialize the reverse engineering task.

As illustrated in Figure 1, the original source code (Figure 1a) defines a user-defined structure comprising three variables. When decompiled from a binary containing debug symbols, the output preserves the original structure and variable names, offering a direct mapping to the ground truth. Conversely, stripped binaries lack these symbols, leading to significantly different decompiled outputs. As shown in Figure 1b and Figure 1c, the decompiler "flattens" the structure into three independent variables, discarding the high-level abstraction of the composite type. This creates a structural mismatch between the source-level ground truth and the decompiled representation.

Consequently, recovering the ground truth requires rigorous reconstruction. Our framework, REBENCH, must reconcile the original data structures and variable names by inspecting stack offsets and memory ranges rather than relying solely on decompiler-generated identifiers. Neglecting this step would underestimate the complexity of the task and risk introducing artificial "shortcuts" that inflate model performance.

Abstracting Symbols via a Knowledge Base. To mitigate these challenges, we employ a knowledge-base (KB) approach for symbol abstraction.

- **KB Construction.** The process begins with the construction of a comprehensive KB containing every symbol from an unstripped version of the binary, as detailed in algorithm 1. We initiate the process by decompiling the executable with debug symbols and systematically traversing each function from its entry point (line 1). For every function, we identify the initial symbols (line 6) and extract their associated stack offsets and memory footprints. These symbols are then mapped at the byte level within a dictionary. This granular mapping is essential because decompilers often fail to recognize complex data structures; instead flattening them into raw size allocations. By mapping symbols at the byte level, we preserve their semantic context for downstream analysis. This procedure is repeated for all symbols within a function and indexed by the function's entry address in the KB.
- **KB Utilization.** Once the KB is populated, we analyze the stripped binary to generate the corresponding decompiled and assembly code (algorithm 2). At this stage, we identify all detectable symbols and replace them with uniform placeholders (e.g., `VAR1`). Simultaneously, we query the KB to retrieve the original names and types. A similar replacement logic is applied to function names, which are substituted with generic identifiers (e.g., `FUNC1`). While algorithm 1 and algorithm 2 omit the granular details of function renaming for brevity, the process involves recording callee entry addresses to ensure that inter-procedural relationships and ground truth names remain recoverable during the analysis of stripped binaries.

Algorithm 1: Building Knowledge Base of Every Symbol from Binary

Input: Unstripped Binary (B)
Output: Knowledge-Base (KB)

```
1  f ← first_func(B)
2  KB ← {}
3  while True do
4      if f == NULL then
5          break
6      sym ← first_sym(f)
7      KB[f.entry_addr] ← {}
8      while True do
9          if sym == NULL then
10             break
11         for i = stack_offset(sym) to
             stack_offset(sym) + size(sym) do
12             KB[f.entry_addr][kb] = {i : [sym.name, sym.type]}
13         sym ← next_sym(f)
14     f ← next_func(B)
```

Figure 2 demonstrates this workflow using a basic addition function. While Figure 2a shows the original C code and Figure 2b shows the standard decompiled output, Figure 2c depicts the final version with abstracted symbols. In a real-world scenario, the intermediate step (Figure 2c) would not retain original identifiers. Instead, we use the raw decompiled output to build the KB, cross-referencing abstracted placeholders (`VAR, TYPE`) with their original forms in the database to provide a reliable, automated mapping for model evaluation.

Limitations of DWARF-only Ground Truth. A possible counter-argument is that DWARF information from unstripped binaries could serve as a direct ground truth. However, our methodology relies on the decompiler's interpretation of the code. Since the decompiler itself utilizes DWARF information to reconstruct the stack layout,

```c
typedef struct {
  int a;
  short b;
  unsigned int c;
} data;

int sum() {
  data val;
  scanf("%d %d %d",
    &val.a, &val.b,
    &val.c);
  return val.a +
    val.b + val.c;
}
```

(a) Original C Code

```c
int sum(void){
  data val;

  scanf("%d %d %d", &val,
    &val.b, &val.c);

  return val.c +
    val.b + val.a;
}
```

(b) Decompiled Code w Debug Symbols

```c
int FUNC(void){
  int local_1c;
  short local_18 [2];
  int local_14;

  scanf("%d %d %d",
    &local_1c, local_18,
    &local_14);

  return local_14 +
    local_18[0] +
    local_1c;
}
```

(c) Decompiled Code wo Debug Symbols

Figure 1: Differences of Decompiled Code Between With and Without Debug Symbols.

```c
int main(int argc) {
  int sum;
  int number1;
  int number2;

  printf("Enter two
    integers: ");
  scanf("%d %d",
    &number1,
    &number2);
  sum = number1 +
        number2;
  printf("%d + %d = %d",
    number1,
    number2,
    sum);
  return 0;
}
```

(a) Original C Code

```c
int main(int argc){
  int sum;
  int number1;
  int number2;

  printf("Enter two
    integers: ");
  __isoc99_scanf("%d %d",
    &number1, &number2);
  sum = num1 + num2;
  printf("%d + %d = %d",
    number1, number2,
    sum);

    return 0;
}
```

(b) Decompiled Code

```c
TYPE1 FUNC1(TYPE2 VAR1){
  TYPE3 VAR2;
  TYPE4 VAR3;
  TYPE5 VAR4;

  FUNC2("Enter two
    integers: ");
  FUNC3("%d %d",
    &VAR3, &VAR4);
  VAR2 = VAR3 + VAR4;
  FUNC2("%d + %d = %d",
    VAR3, VAR4, VAR2);

  return 0;
}
```

(c) Our Final Input

Figure 2: The Process of Normalization. The Example with Debug Symbols is Used For Better Understanding.

Algorithm 2: Utilizing Knowledge Base to Build Ground Truth

Input: Knowledge-Base (KB), Stripped binary (B)
Output: Decompiled Code

$f \leftarrow first\_func(B)$
while $True$ **do**
  if $f == NULL$ **then**
    $break$
  $sym \leftarrow first\_sym(f)$
  $idx \leftarrow 1$
  while $True$ **do**
    if $sym == NULL$ **then**
      $break$
    $sym.name, sym.type = VAR_{idx}, TYPE_{idx}$
    $sym.name\_gt, sym.type\_gt = KB[f.entry\_addr][kb][stack\_addr(sym)]$
    $idx++$
    $sym \leftarrow next\_sym(f)$
  $generate\_decompile\_code(f)$
  $f \leftarrow next\_func(B)$

using DWARF directly as the sole ground truth would be redundant and potentially overlook the idiosyncratic ways a decompiler reinterprets symbol relationships.

Moreover, relying strictly on DWARF may oversimplify the evaluation of variable name recovery. Decompilers often generate a different number of variables than those present in the original source code. Forcing an LLM to match the exact count and layout dictated by DWARF, rather than what is actually visible in the decompiled code, would fail to accurately measure the model's performance in a realistic reverse engineering context.

Filtering Decompiled Code. To ensure the integrity of our dataset, we implement a rigorous filtering pipeline. This process addresses two primary concerns: the removal of duplicate functions and the exclusion of external library wrappers.

- **Redundancy and Data Leakage:** Because our dataset aggregates multiple projects that may share common utility functions or dependencies, eliminating duplicates is essential. Failure to do so would result in data leakage, where functions encountered during fine-tuning reappear during evaluation, leading to inflated and biased performance metrics. We avoid relying solely on function names for de-duplication, as identical names often correspond to different implementations across different versions or projects. Instead, we compute code similarity across both assembly and decompiled representations to identify and discard redundant entries.
- **External and Trivial Functions:** A significant portion of a binary's decompiled code often consists of external library calls (e.g., PLT stubs). These functions are typically incomplete, containing only a few instructions to invoke a shared library, which provides insufficient context for meaningful evaluation. To address this, we exclude functions that fall below a specific line-count threshold (e.g., fewer than 10 lines of decompiled code). This heuristic offers two key advantages, 1) It effectively filters

out external wrappers that lack internal logic, and 2) It eliminates trivial functions (such as simple getters or counter increments) that offer limited semantic depth. By removing these, we ensure the benchmark focuses on complex code that meaningfully challenges the model's ability to infer original identifiers and types.

Adapting to Downstream Tasks. The REBENCH abstracted representation is designed for high versatility. By using a uniform, indexed placeholder system, our format can be easily customized for various binary analysis tasks. We illustrate this adaptability through four representative applications:

- **Single Function Name Recovery.** This task involves predicting a target function's name based on its decompiled or assembly context. To facilitate this, we restore all internal symbols while leaving the target function name abstracted. The placeholder acts as a functional equivalent to the `[MASK]` token in masked language models (MLMs). Task difficulty can be modulated by varying the ratio of abstracted-to-original symbols, effectively simulating different levels of stripping (from partially to fully stripped binaries).
- **Data Structure Reconstruction.** This task requires recovering user-defined types from "flattened" variables. While decompilers often break composite structures into individual stack variables, the REBENCH ground truth maintains the original associations and memory offsets. By analyzing variable usage patterns and inter-procedural interactions, models can be trained to re-aggregate these flattened elements into their original composite types.
- **Function Similarity.** REBENCH supports function-level similarity analysis directly. While our dataset minimizes overlaps, the presence of multiple versions of the same project allows for testing model robustness across compiler variations. This capability can be naturally extended to binary-level similarity through lightweight aggregation of function-level embeddings.
- **Control Flow Graph Recovery.** Since REBENCH preserves function boundaries and call relationships, it provides a foundational layer for CFG reconstruction. While static analysis alone may struggle with indirect jumps or calls, our enriched representation, which assigns unique, consistent identifiers to every function and variable, ensures that cross-references and dependencies are preserved for more advanced post-processing techniques.

Ultimately, REBENCH provides a robust and flexible framework that not only enhances the accuracy of standard recovery tasks but also broadens the scope of automated program understanding in reverse engineering.

3.4 Evaluation Metrics

To rigorously assess the performance of LLMs or other frameworks in reverse engineering, we employ distinct metrics for type inference and name recovery that account for both the strictness of C semantics and the flexibility of natural language.

Type Inference. The C programming language consists of a finite set of primitive types (e.g., `char, int, float`) and composite types (e.g., `struct, union`) constructed from those primitives. We employ exact type matching for our evaluation. This strictness is necessary because even closely related types, such as unsigned int and signed int, exhibit fundamentally different behaviors under C semantics. For instance, `unsigned int` overflow is well-defined and wraps around to zero, whereas `signed int` overflow is undefined and may lead to unpredictable compiler optimizations or vulnerabilities. Consequently, semantic equivalence cannot be assumed; exact matching ensures the most reliable measurement of type reconstruction accuracy.

```
// int main(int argc)
int main(char count)
{
    int index;  // int sum;
    int num1;   // int number1;
    int num2;   // int number2;

    //printf("Enter two integers: ");
    printf("Enter two integers: ");

    //scanf("%d %d",
    //      &number1, &number2);
    scanf("%d %d", &num1, &num2);

    // sum = number1 + number2;
    index = num1 + num2;

    // printf("%d + %d = %d",
    // number1, number2, sum);
    printf("%d + %d = %d",
    num1, num2, index);

    return 0;
}
```

Figure 3: Example of LLM Inference Results. Comments are the Original Code. Blue Boxes are Correctly Inferred and Red Boxes are not.

Name Recovery. As discussed in §2.3, names inferred by LLMs may be semantically correct even if they do not provide a character-for-character match with the ground truth. For example, `string` and `str` or `memory` and `mem` are functionally equivalent in a reverse engineering context. To address this, we move beyond exact string matching and adopt a semantic similarity metric based on the SYMLM method, which leverages CodeWordNet to compute the semantic distance between sub-tokens. The process involves:

(1) **Decomposition:** Breaking each identifier into constituent sub-names (e.g., `extract_trim_name` into `extract, trim, name`).
(2) **Semantic Mapping:** Measuring the similarity between corresponding sub-names using CodeWordNet.
(3) **Aggregate Scoring:** Calculating a similarity score that credits the model for synonyms while penalizing missing or incorrect semantic components.

Example. Suppose the ground truth is `extract_trim_name` and the inferred name is `get_file_name`.

- `name` is an exact match.
- `extract` and `get` are identified as semantically related synonyms.
- `trim`(ground truth) and `file`(inferred) are recognized as distinct, resulting in a mismatch.

In this case, the evaluation identifies 2 True Positives (TP) and 1 False Negative (FN), yielding an F1 score of 0.8 (1.0 Precision and 0.66 Recall).

Count (m)	x64				x86				ARM				MIPS			
	O0	O1	O2	O3	O0	O1	O2	O3	O0	O1	O2	O3	O0	O1	O2	O3
Before Filtering Process																
LoC	48.82	43.47	43.67	50.59	40.41	36.01	36.29	41.50	11.11	9.11	8.81	10.25	14.86	14.03	13.43	14.51
Func	4.64	3.99	3.97	4.18	5.20	4.30	4.27	4.46	0.95	0.72	0.66	0.66	0.93	0.63	0.59	0.56
Var	5.92	2.28	2.22	2.51	6.14	3.75	3.84	4.26	1.83	0.87	0.83	0.91	2.14	1.09	1.08	1.09
After Filtering Process																
LoC	29.94	28.29	29.11	36.88	26.19	24.37	25.31	30.88	7.99	6.87	6.85	8.34	9.77	10.10	9.75	11.02
Func	2.87	2.45	2.51	2.88	3.23	2.79	2.83	3.14	0.67	0.52	0.49	0.51	0.73	0.50	0.48	0.48
Var	3.65	1.65	1.61	1.94	4.06	2.49	2.66	3.16	1.40	0.70	0.67	0.75	1.61	0.88	0.90	0.93

Table 1: Statistics of REBENCH Before and After Filtering Process. The Unit Size is a Million.

	O0	O1	O2	O3	O0	O1	O2	O3
	CodeLlama				Llama2			
x64	0.0651	0.0695	0.0671	0.0650	0.0348	0.0362	0.0368	0.0354
x86	0.1187	0.1281	0.1292	0.1291	0.0542	0.0672	0.0611	0.0637
ARM	0.0429	0.064	0.0522	0.0533	0.0340	0.0330	0.0330	0.0383
MIPS	0.0230	0.0248	0.0211	0.0208	0.0250	0.0252	0.0302	0.0280

Table 2: F1 Scores on Function Name Recovery.

	O0	O1	O2	O3	O0	O1	O2	O3
	CodeLlama				Llama2			
x64	0.0243	0.0301	0.0352	0.0426	0.0157	0.0192	0.0256	0.0147
x86	0.0385	0.0571	0.0546	0.0561	0.0208	0.0346	0.0264	0.0310
ARM	0.0310	0.0308	0.0329	0.0416	0.0163	0.0185	0.0252	0.0158
MIPS	0.0214	0.0287	0.0176	0.0305	0.0156	0.0196	0.0213	0.0098

Table 3: F1 Scores on Variable Name Recovery.

	O0	O1	O2	O3	O0	O1	O2	O3
	CodeLlama				Llama2			
x64	0.0109	0.0121	0.0049	0.0081	0.0159	0.0145	0.0095	0.0082
x86	0.0141	0.0220	0.0149	0.0145	0.0202	0.0256	0.0253	0.0273
ARM	0.0091	0.0120	0.0094	0.0054	0.0301	0.0387	0.0470	0.0220
MIPS	0.0108	0.0079	0.0043	0.0063	0.0357	0.0102	0.0132	0.0103

Table 4: F1 Scores on Variable Type Inference.

Evaluation Metrics. We report Precision, Recall, and F1 score as our primary metrics to provide a comprehensive view of LLM performance. While our framework also supports Area Under the Curve (AUC) and Mean Average Precision (mAP), we omit them from the primary discussion as they reflect similar performance trends. The REBENCH architecture remains modular, allowing for the integration of additional metrics as needed.

Evaluation Example. Figure 3 illustrates the performance of a CodeLlama model when tasked with recovering the addition function previously described in Figure 2.

- **Type Inference:** The model correctly identifies four out of five type tokens, incorrectly inferring `char` instead of `int` for `TYPE2`. This results in a type F1 score of 0.8.
- **Function Name Recovery:** The model achieves a perfect F1 score of 1.0, correctly identifying the high-level intent.
- **Variable Name Recovery:** The model infers `count, index, num1`, and `num2`. While none are exact string matches for the original `number1` and `number2`, REBENCH utilizes CodeWordNet to validate `num1` and `num2` as semantically correct. With two semantic matches out of four predicted variables, the model achieves a Variable Name F1 score of 0.5.

4 Empirical Evaluation

Table 1 shows the total number of function and variable before and after filtering process. It illustrates the comprehensiveness of REBENCH, since it has more than millions of functions and variables in general. Moreover, to demonstrate the utility of REBENCH, we conduct an empirical study using the CodeLlama 13B and Llama2 13B models on two core reverse engineering tasks: name recovery and type inference.

Experimental Setup. Due to the vast number of functions in our dataset, we performed our evaluation on a randomly selected subset. For each combination of processor architecture and optimization level, we selected 2,500 functions for testing and 10,000 functions for fine-tuning the model.

Results and Analysis. The performance of the model, measured by F1 score, is summarized in Table 2, Table 3, and Table 4. Our findings reveal several key insights:

- **Task Complexity:** Across all architectures and optimization levels, F1 scores for name recovery were consistently higher than those for type inference. This disparity underscores the inherent difficulty of reconstructing precise C-type semantics compared to inferring semantic labels.
- **Performance Gaps:** Both models exhibit a noticeable performance gap when compared to specialized frameworks like RESYM and SYMGEN. We attribute this to two primary factors, 1) The cognitive load on the LLM when handling multiple, complex tasks simultaneously, and 2) The need for more sophisticated pre-processing of decompiled code to better expose low-level semantic relationships to the model.
- **Architectural Trend:** As shown in the tables, performance varies significantly across architectures, with x86 generally yielding higher recovery rates than MIPS or ARM.

5 Discussion & Future Work

In this section, we discuss the current REBENCH and outline potential directions for future development.

Expanding the Scope of Tasks. While our current evaluation focuses on name recovery and type inference, binary analysis encompasses a much broader array of challenges. Future iterations of REBENCH will integrate benchmarks for memory corruption detection, control-flow integrity (CFI) analysis, and binary similarity detection. By expanding the task set, we aim to provide a more holistic view of LLM capabilities in security-critical contexts.

Model Generalization and Data Leakage. A common concern in LLM-based studies is data leakage from the pre-training or fine-tuning phases. We argue that the REBENCH pipeline significantly mitigates this risk. By transforming source code through both decompilation and normalization, the resulting input format differs substantially from the original source found in public repositories. This makes it highly unlikely that the model relies on memorized code during evaluation. Moreover, we plan to extend our analysis to a wider variety of models to determine if these performance trends are universal across different architectures.

Decompiler Dependencies and Limitations. REBENCH relies on decompiler outputs for generating its universal representations. However, decompilers are inherently imperfect. We have observed instances where tools like Ghidra introduce artifacts not present in

the source, such as stack canaries, or omit vital type annotations. These errors can propagate through the preprocessing stage and affect symbol replacement. To improve robustness, we are investigating "decompiler-agnostic" input formats and plan to incorporate additional tools like IDA Pro and angr to identify which decompilers are most "LLM-friendly".

Advanced Deduplication and Maintenance. Current deduplication in REBENCH relies on direct comparison of assembly and decompiled code. However, functions that are semantically identical can appear different due to compiler optimizations or varying memory addresses. We intend to implement more sophisticated, semantics-driven deduplication techniques to further reduce the risk of data leakage. Finally, to ensure the long-term utility of the benchmark, we will maintain REBENCH with regular updates, including real-world firmware and binaries subjected to advanced obfuscation.

6 Conclusion

In recent years, numerous reverse engineering approaches have been proposed, leveraging techniques ranging from traditional machine learning to modern LLMs. In particular, LLM-based methods often rely on fine-tuning existing models for specialized tasks. However, the field currently lacks a standardized benchmark for evaluating both pre-trained and fine-tuned LLMs on reverse engineering tasks, largely due to the absence of comprehensive datasets. To address this gap, we introduce REBENCH, a large-scale benchmark dataset comprising 96 projects compiled across multiple architectures and optimization levels. REBENCH enables systematic measurement of LLM performance on a variety of reverse engineering tasks. A key feature of the benchmark is its transformation of executable binaries into a universal, human-readable format, supported by a knowledge base that preserves byte-level stack layout information. Importantly, REBENCH does not oversimplify reverse engineering challenges, ensuring that evaluation remains realistic and reflective of real-world conditions. We evaluate two LLMs with REBENCH, and it demonstrates the difficulties of LLM for given tasks.

7 Data Availability

The data normalization process can be reproduced using the provided scripts, which include implementations for both Ghidra and IDA Pro. The generated REBENCH dataset and the corresponding conversion scripts will be publicly released to facilitate reproducibility and future research.

References

[1] National Security Agency. 2019. Ghidra Decompiler. https://ghidra-sre.org/. Accessed: 2026-03-23.

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

[3] Qibin Chen, Jeremy Lacomis, Edward J Schwartz, Claire Le Goues, Graham Neubig, and Bogdan Vasilescu. 2022. Augmenting decompiler output with learned variable names and types. In *31st USENIX Security Symposium (USENIX Security 22)*. 4327–4343.

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

[5] Yue Duan, Xuezixiang Li, Jinghan Wang, and Heng Yin. 2020. Deepbindiff: Learning program-wide code representations for binary diffing. In *Network and distributed system security symposium*.

[6] Dror G Feitelson, Ayelet Mizrahi, Nofar Noy, Aviad Ben Shabat, Or Eliyahu, and Roy Sheffer. 2020. How developers choose names. *IEEE Transactions on Software Engineering* 48, 1 (2020), 37–52.

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

[8] Abram Hindle, Earl T Barr, Mark Gabel, Zhendong Su, and Premkumar Devanbu. 2016. On the naturalness of software. *Commun. ACM* 59, 5 (2016), 122–131.

[9] Kexin Huang, Jaan Altosaar, and Rajesh Ranganath. 2019. Clinicalbert: Modeling clinical notes and predicting hospital readmission. *arXiv preprint arXiv:1904.05342* (2019).

[10] Linxi Jiang, Xin Jin, and Zhiqiang Lin. 2025. Beyond Classification: Inferring Function Names in Stripped Binaries via Domain Adapted LLMs.. In *NDSS*.

[11] Xin Jin, Jonathan Larson, Weiwei Yang, and Zhiqiang Lin. 2023. Binary code summarization: Benchmarking chatgpt/gpt-4 and other large language models. *arXiv preprint arXiv:2312.09601* (2023).

[12] Xin Jin, Kexin Pei, Jun Yeon Won, and Zhiqiang Lin. 2022. 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*. 1631–1645.

[13] Hyunjin Kim, Jinyeong Bak, Kyunghyun Cho, and Hyungjoon Koo. 2023. 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*. 951–965.

[14] Jeremy Lacomis, Pengcheng Yin, Edward Schwartz, Miltiadis Allamanis, Claire Le Goues, Graham Neubig, and Bogdan Vasilescu. 2019. Dire: A neural approach to decompiled identifier naming. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 628–639.

[15] Jinhyuk Lee, Wonjin Yoon, Sungdong Kim, Donghyeon Kim, Sunkyu Kim, Chan Ho So, and Jaewoo Kang. 2020. BioBERT: a pre-trained biomedical language representation model for biomedical text mining. *Bioinformatics* 36, 4 (2020), 1234–1240.

[16] OpenAI. 2022. chatGPT. https://chat.openai.com/. Accessed: 2026-03-16.

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

[18] Kexin Pei, Zhou Xuan, Junfeng Yang, Suman Jana, and Baishakhi Ray. 2020. Trex: Learning execution semantics from micro-traces for binary similarity. *arXiv preprint arXiv:2012.08680* (2020).

[19] Xiuwei Shang, Shaoyin Cheng, Guoqiang Chen, Yanming Zhang, Li Hu, Xiao Yu, Gangyang Li, Weiming Zhang, and Nenghai Yu. 2024. How far have we gone in binary code understanding using large language models. In *2024 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 1–12.

[20] Lisa Torrey and Jude Shavlik. 2010. Transfer learning. In *Handbook of research on machine learning applications and trends: algorithms, methods, and techniques*. IGI global, 242–264.

[21] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. *Advances in neural information processing systems* 30 (2017).

[22] Danning Xie, Zhuo Zhang, Nan Jiang, Xiangzhe Xu, Lin Tan, and Xiangyu Zhang. 2024. 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*. 4554–4568.

[23] Xiangzhe Xu, Shiwei Feng, Yapeng Ye, Guangyu Shen, Zian Su, Siyuan Cheng, Guanhong Tao, Qingkai Shi, Zhuo Zhang, and Xiangyu Zhang. 2023. Improving Binary Code Similarity Transformer Models by Semantics-Driven Instruction Deemphasis. (2023).

[24] Xiangzhe Xu, Zhuo Zhang, Shiwei Feng, Yapeng Ye, Zian Su, Nan Jiang, Siyuan Cheng, Lin Tan, and Xiangyu Zhang. 2023. LmPa: Improving Decompilation by Synergy of Large Language Model and Program Analysis. *arXiv preprint arXiv:2306.02546* (2023).

[25] Jia Yang, Cai Fu, Xiao-Yang Liu, Heng Yin, and Pan Zhou. 2021. Codee: A tensor embedding scheme for binary code search. *IEEE Transactions on Software Engineering* 48, 7 (2021), 2224–2244.

[26] Xinze Yang, Chunkai Zhang, Yizhi Sun, Kairui Pang, Luru Jing, Shiyun Wa, and Chunli Lv. 2023. FinChain-BERT: A High-Accuracy Automatic Fraud Detection Model Based on NLP Methods for Financial Scenarios. *Information* 14, 9 (2023), 499.

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