BinRAG: An RAG-Based Decompilation Framework Fusing Name Prediction and Calling Context

WAI KIN WONG, Hong Kong University of Science and Technology, Hong Kong
DAOYUAN WU*, Lingnan University, Hong Kong
ZHIBO LIU*, Nanjing University, China
HUAIJIN WANG, Shandong University, China
ZONGJIE LI, Hong Kong University of Science and Technology, Hong Kong
SHUAI WANG, Hong Kong University of Science and Technology, Hong Kong

Decompiling stripped binaries to assist human reverse engineers remains a critical yet highly challenging task in software security. Prior work has developed various neural networks and even dedicated large language models (LLMs) to improve function and variable name recovery in decompiled code. Nonetheless, these approaches alone fall short of enabling a generic LLM-based decompilation pipeline that can reliably enhance the readability and semantic clarity of decompiled outputs.

In this paper, we propose BinRAG, a retrieval-augmented generation (RAG) based decompilation framework designed to enhance the decompilation of stripped binaries. Building upon name prediction models, BinRAG features three novel designs: (1) it first utilizes name prediction models to transform raw decompiled outputs into enriched, source-like queries for RAG retrieval; (2) it further enhances these queries using a fine-tuned specialized LLM conditioned on the predicted variable names, enabling more accurate retrieval from a curated example database; (3) it then integrates these retrieved examples with the target's calling context, employing a general-purpose LLM to synthesize high-fidelity, human-readable decompiled code. Evaluation on 3,200 functions from real-world software repositories demonstrates that BinRAG improves readability by 8.4% over standard RAG and semantic precision by 31.2% over the next-best prior baseline. Our results show that BinRAG effectively scales to large codebases and significantly reduces manual effort in reverse engineering tasks.

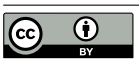
CCS Concepts: • **Security and privacy → Software reverse engineering**.

Additional Key Words and Phrases: Decompilation, Binary Analysis, Large Language Model

ACM Reference Format:
Wai Kin Wong, Daoyuan Wu, Zhibo Liu, Huaijin Wang, Zongjie Li, and Shuai Wang. 2026. BinRAG: An RAG-Based Decompilation Framework Fusing Name Prediction and Calling Context. *Proc. ACM Softw. Eng.* 3, ISSTA, Article ISSTA154 (October 2026), 26 pages. https://doi.org/10.1145/3832245

*Corresponding authors.

Authors' Contact Information: Wai Kin Wong, Hong Kong University of Science and Technology, Hong Kong, Hong Kong, wkwongal@cse.ust.hk; Daoyuan Wu, Lingnan University, Hong Kong, Hong Kong, daoyuanwu@ln.edu.hk; Zhibo Liu, Nanjing University, State Key Laboratory of Novel Software Technology, Nanjing, China, zhiboliu@nju.edu.cn; Huaijin Wang, Shandong University, Jinan, China, huaijinwang@sdu.edu.cn; Zongjie Li, Hong Kong University of Science and Technology, Hong Kong, Hong Kong, zligo@cse.ust.hk; Shuai Wang, Hong Kong University of Science and Technology, Hong Kong, Hong Kong, shuaiw@cse.ust.hk.



ACM 2994-970X/2026/10-ARTISSTA154
https://doi.org/10.1145/3832245

1 Introduction

Reverse engineering software binaries is fundamental to critical security applications, including vulnerability discovery [67, 119], malware analysis [26], and forensic investigation. Analysts often work exclusively with compiled binaries due to the unavailability of source code, whether dealing with proprietary software, legacy systems, or malicious programs. The compilation process, however, irreversibly removes high-level semantic information, such as meaningful variable names, function signatures, and structural abstractions, leaving analysts to interpret low-level assembly instructions.

Decompilers aim to bridge this semantic gap by reconstructing human-readable pseudocode from binary executables [27, 28]. Decompilation comprises three abstraction-raising steps: (1) *disassembly*, which converts binary machine code into architecture-specific assembly instructions; (2) *intermediate representation (IR) lifting*, which transforms assembly into an architecture-independent abstraction [64, 120] for program analysis; and (3) *pseudocode generation*, which converts the IR into high-level pseudocode using predefined control-flow and data-flow patterns [60, 77].

Semantic Poverty. While modern decompilers such as Hex-Rays [39] and NSA's Ghidra [70] have significantly advanced binary analysis, their outputs remain fundamentally limited by the information loss inherent to compilation [63, 77, 90, 106]. Decompiled code typically contains generic variable names (e.g., `v1`, `a2`), imprecise data types, reconstructed control flow that obscures the original logic, and decompiler-specific artifacts that deviate from conventional programming patterns. We refer to this lack of source-level semantic detail as *semantic poverty*. It limits downstream analysis and comprehension, forcing analysts to rely on substantial manual effort, pattern recognition, and domain expertise to reconstruct a program's intent.

Prior work has explored neural and LLM-based techniques to mitigate semantic poverty and improve decompiled-code readability. Early approaches framed the task as machine translation and used natural language processing techniques [24, 46, 48, 73] to predict identifiers such as function and variable names, but they suffer from the out-of-vocabulary problem caused by the long-tail distribution of source-code identifiers [73]. Recent LLM-based systems broaden the scope, including symbol-recovery models such as ReSym [99] and GenNm [104], and refinement models that aim to produce more compilable, source-like decompiled code [79]. However, these approaches often improve isolated aspects of decompiler output rather than jointly recovering semantic names, source-like structures, and contextual intent, as illustrated in §2.2. This suggests that scaling or fine-tuning models alone is insufficient: the missing ingredient is relevant semantic context.

RAG Opportunity and Bottleneck. RAG [50] offers a natural way to provide such context by retrieving relevant information from a large corpus and augmenting the LLM input at inference time. Context augmentation has improved LLM performance in many domains [23, 30, 47, 81], and it is particularly promising for decompilation because retrieved examples can demonstrate how low-level decompiled code may be transformed into high-level, human-readable source code. These examples guide the model's reasoning beyond what the target decompiled function alone provides.

However, applying standard RAG to decompiled code is challenging because existing retrieval methods assume that queries and retrieval corpora contain enough semantic signals for lexical or embedding similarity to be meaningful. Sparse retrieval algorithms such as BM25 [76] rely on term frequency, but such meaningful terms are largely absent from decompiled code. Similarly, embedding-based approaches capture semantic similarity but rely heavily on distinctive identifiers, such as variable and function names, for effective discrimination [44, 103, 118]. Stripped decompiled outputs violate this assumption: raw queries lack meaningful identifiers, precise types, and source-level abstractions. As a result, standard RAG may retrieve examples that look locally similar but implement different functionality, giving weak or even misleading context to the reasoning LLM.

Our Approach. Although decompiled code lacks explicit source-level semantics, semantic poverty does not mean semantic absence. Latent cues remain embedded in variable usage, control/data-flow patterns, and calling relationships. Specialized models trained on large corpora of human-written code [24, 40, 46, 48, 73, 99, 104] can recover cues such as meaningful identifier names, while LLMs can reshape non-idiomatic decompiler output into more source-like forms. Our key insight is that these recovered cues should support not only final code generation, but also retrieval itself: enriched queries make relevant examples easier to retrieve, structural correspondence helps filter misleading matches, and inter-procedural context helps the final LLM resolve ambiguities that cannot be settled from a single function alone.

We realize these insights in BinRAG, a novel framework that employs a multi-stage RAG process for improving decompiled-code readability. BinRAG first transforms semantically poor decompiled code into enriched retrieval queries, then retrieves and structurally re-ranks relevant examples, and finally uses these examples together with calling context to guide a general-purpose LLM in producing more human-readable decompiled code. By combining specialized semantic enrichment, structure-aware retrieval, and context-aware refinement, BinRAG addresses the retrieval bottleneck and harnesses the complementary strengths of specialized and general-purpose LLMs. We evaluate BinRAG on a dataset of 3,200 functions from real-world software and compare it against state-of-the-art (SoTA) approaches including DeGPT [41], LLM4Decompile-ref [79], VarDecoder [99], and GenNm [104]. Our results show that by addressing the difficulty of retrieval, BinRAG achieves significant improvements in both semantic accuracy and code readability.

Contributions. In sum, this work makes the following key contributions:

- We identify a cross-domain retrieval mismatch unique to decompilation: the query is formed from stripped decompiler output, yet the most useful reference examples are written in full source code. This semantic poverty of decompiled code creates a retrieval bottleneck that conventional RAG and model-centric approaches are poorly equipped to handle.
- We propose BinRAG, an RAG-based framework that addresses this bottleneck by converting semantically poor decompiled code into semantically enriched retrieval queries.
- We demonstrate BinRAG's effectiveness through a comprehensive evaluation on 3,200 real-world functions across 8 different domains and 4 optimization levels, showing significant improvements in code readability compared to state-of-the-art approaches.

2 Preliminaries and Key Insights

2.1 A Motivating Example

To illustrate the challenges of analyzing decompiled code, consider a digit-parsing function from a mail library [9], shown in both its original source (Figure 1d) and decompiled code (Figure 1a). The original implementation (Figure 1d) exemplifies clear software engineering practices: it uses descriptive identifiers (`parse_count`, `parser`) and employs structured data access (`parser->curp`), both of which aid program comprehension.

After compilation and decompilation, however, the output (Figure 1a) exhibits several properties that severely impair comprehensibility. Meaningful names are replaced with generic placeholders (`a1`, `v2`) that convey no semantic purpose. Typed structure access is transformed into low-level pointer arithmetic (`*(char**)(a1 + 8)`). Character constants appear as numeric ASCII values (45 instead of '-'), reducing immediate readability, and function names are replaced with function addresses (`sub_ED70`). Consequently, reverse engineers must invest considerable effort to infer the code's intent, often relying on substantial domain knowledge and context.

```
__int64 __fastcall sub_ED70(__int64 a1) {
  char *v2; char v3; ...
  v2 = *(char**)(a1 + 8);
  v3 = *v2;
  if ( *v2 ) {
    if (v3 == 45) {
    *(_QWORD *)(a1 + 8) = v2 + 1;
    ...
    }
  ...
  if (*(char*)v5[0])
    sub_ECF0(...);
  ...
}
```

(a) Raw decompiler output.

```
__int64 __fastcall parse_number(__int64 a1){
  char *input_ptr; char first_char; ...
  input_ptr = *(char **)(a1 + 8);
  first_char = *input_ptr;
  // Check if the first character is '-'
  if (first_char == 45)
  // Move the pointer to the next character
  *(_QWORD *)(a1 + 8) = input_ptr + 1;
  ...
  // Check a byte value from temp_arr
  if (*(char *)temp_arr[0])
    sub_ECF0(...); // Call sub_ECF0
  ...
}
```

(b) Output from DeGPT [41].

```
size_t sub_ED70(sub_C912_C912_struct *s)
{
  sub_C912_C912_struct s2;
  ...
  if (*s→f1 == '-' || *s→f1 == '+'){
  ...
  s→f1++;
  ...
  }
  ...
  if (*s2.f1)
  sub_ECF0(s→f1, &s2);
  ...
}
```

(c) Output from LLM4Decompile-ref.

```
static int parse_count
  (struct msgset_parser *parser){
  char *endp;
  ...
  if (*parser→curp == '-'){
    parser→sign = 1;
    parser→curp++;
  ...
  }
  if (*endp)
    msgset_abort (parser→curp);
  parser→curp = endp;
  return 1;
  }
```

(d) Original source code (ground truth).

Fig. 1. Comparison of a digit-parsing function refined by different methods. (a) Raw decompiler output. (b) DeGPT adds comments but fails to improve structure. (c) LLM4Decompile-ref improves structure but leaves semantic gaps. (d) The original source code, showing the semantic gap that persists in (b) and (c).

2.2 Why Existing Refinement Falls Short

Figure 1 illustrates why existing LLM-based refinement methods remain insufficient for analyst-oriented readability. VarDecoder [99], a specialized identifier-recovery model, can rename `a1` to `ParseOpt` and `v5` to `endptr`. Such names improve local comprehension, but they leave low-level pointer arithmetic, numeric character constants, and unresolved callees unchanged. Thus, identifier recovery alone does not reconstruct source-like data accesses or higher-level program intent.

Other LLM-based tools like DeGPT [41] and LLM4Decompile-ref [79] aim to improve the readability and executability of decompiled code, respectively. DeGPT renames some identifiers and adds explanatory comments, but retains decompiler-specific expressions, numeric character constants, and unresolved callees (Figure 1b). LLM4Decompile-ref improves code structure (Figure 1c), but still retains generic struct and field names such as `s` and `f1`, rather than recovering source-level abstractions such as `parser->curp`.

These outputs show that existing methods address isolated symptoms of decompiler output, such as names, comments, or syntactic structure, but do not jointly recover the semantic cues needed for source-like readability. This motivates an approach that first enriches decompiled code with recovered semantics and then uses additional context to guide refinement.

2.3 RAG for Decompiled Code

Augmenting LLMs with relevant context has been shown to significantly enhance their reasoning capabilities across various domains [47, 61, 81]. A typical RAG system consists of two main components: a *retriever* (R) and a *generator*. Given a user query, the retriever identifies a small set of relevant examples from a knowledge base. These retrieved examples are provided to the generator (typically an LLM) as additional context, thereby grounding its output with relevant contextual information.

In the context of decompilation, RAG offers considerable potential by retrieving relevant source-decompiled code pairs. These pairs can illuminate the transformation patterns required to reconstruct human-readable code. Such examples help the LLM to understand how specific, non-idiomatic low-level constructs map to high-level source code. They serve as contextual "hints" that are unavailable in other LLM-based decompilation approaches, grounding the model's reasoning in concrete examples rather than relying solely on its pre-trained general code knowledge. To facilitate further discussion, we formalize the retrieval task for decompilation as follows:

Definition 1 (RAG for Decompiled Code). *Given a target decompiled function, f_d, from a stripped binary, for which the original source code, f_s, is unknown, and a corpus $K = \{(f_{s_i}, f_{d_i})\}_{i=1}^{N}$ of corresponding source-decompiled function pairs, the goal of RAG retrieval is to select the most relevant examples $C \subset K$ for subsequent reasoning.*

2.4 Challenges of Standard RAG

Applying standard RAG to decompiled code poses three challenges in retrieval and generation.

C1: Semantically Poor Queries. Ideally, the retriever should select examples whose source functions are semantically close to the unknown source function f_s. In practice, however, retrieval can only query the decompiled function f_d, which lacks source-level cues such as meaningful identifiers, precise types, and high-level abstractions. Generic retrievers therefore have weak discriminative signals and often fall back on shallow syntactic patterns rather than true functional similarity.

```
1  __int64 __fastcall sub_ED70(__int64 a1) {
2    v2 = *(char**)(a1 + 8);
3    v3 = *v2;
4    if ( *v2 ) {
5      if (v3 == 45) {
6      *(_QWORD *)(a1 + 8) = v2 + 1;
7      ...
8    if (*(char*)v5[0])
9      sub_ECF0(...);
10   ...
```

(a) Decompiler Output

```
1  void __fastcall sub_AA80(...){
2    __int64 v2; char v3;
3    if ( (*(_BYTE *)(a2 + 49) & 2) != 0 ){
4      ...
5    if ( (unsigned int)sub_AA90(v2, a2) ){
6        v4 = *(_WORD *)(a2 + 20);
7        if ( v4 > *(__int16 *)(v2 + 20) )
8          v4 = *(_WORD *)(v2 + 20);
9    ...
10
```

(b) Retrieved Example from standard RAG

```
1  static void whereLoopAdjustCost(...){
2    for(; p; p=p->pNextLoop){
3      ...
4    if( ...){
5      pTemplate->rRun =
6        MIN(p->rRun, pTemplate->rRun);
7      pTemplate->nOut  =
8      MIN(p->nOut - 1, pTemplate->nOut);
9      ...
10 }
```

(c) sub_AA80's Source Code

Fig. 2. Simplified example retrieved by standard RAG.

C2: Structurally Misleading Retrieval. Decompiled code also exhibits a semantic-structural mismatch. The same source function may produce different decompiled outputs under different compilation settings, while unrelated source functions may yield superficially similar decompiled code. This challenge can persist even with enriched queries: functions that appear syntactically similar may still serve different purposes because functions do not exist in isolation, and their callees provide important evidence about what a function actually does. Figure 2 illustrates this problem, where CodeSplitter [12] with CodeT5+ embeddings [94] retrieves configuration adjustment

functions for the motivating digit-parsing example. These candidates resemble the target through local conditionals and struct-field assignments, but they implement different functionality.

Conditioning the LLM on such misleading examples can cause it to incorrectly refine the digit-parsing function toward a generic data-handling routine (Figure 3). Thus, inaccurate retrieval is not merely uninformative; it can actively degrade generation by supplying plausible but wrong context.

C3: Context-Limited Generation. Even with retrieved examples, function-local refinement can remain ambiguous. In Figure 1a, for example, the role of `sub_ECF0` cannot be reliably inferred from the target function alone. Without inter-procedural context, the LLM may assign plausible but incorrect names or behaviors to unresolved callees, limiting the accuracy of the final refinement.

```
1  __int64 process_config_entry(
       ConfigCtx *ctx) {
2    char *ptr = ctx→current_pos;
3    ...
4    if (*ptr == 45) {
5      ctx→current_pos = ptr + 1;
6    }
7    ...
8    if (*ctx→current_pos)
9      update_entry_state(ctx→
           current_pos);
10   ...
11   return 1;
12 }
```

Fig. 3. Output with Standard RAG.

2.5 Design Insights

These challenges suggest that effective RAG for decompilation must adapt both retrieval and generation to the semantics of stripped code. We derive three corresponding insights.

S1: Enrich Queries Before Retrieval. To address **C1**, the query should be enriched before retrieval. Rather than retrieving directly from raw decompiler output, a specialized model can recover likely identifiers and transform decompiled code into a source-like representation prior to retrieval, mirroring how human analysts reconstruct high-level abstractions from low-level code [66]. This projection shifts retrieval toward the source-code domain, where semantic similarity is better defined and existing code embedding models are more effective.

S2: Retrieve with Structural Correspondence. To address **C2**, retrieval should verify structural correspondence between the target and each candidate, rather than relying on local similarity. Analysts often infer a function's purpose from its callees: a function calling `malloc` and `strcpy` likely differs from one calling `socket` and `connect`. Comparing the callees of query and candidate functions therefore helps filter candidates that are syntactically similar but functionally irrelevant.

S3: Refine with Inter-Procedural Context. To address **C3**, generation should incorporate compact summaries of relevant callees as inter-procedural context. For example, if `sub_ECF0` handles error conditions, this context allows the LLM to interpret it as an error handler rather than a generic state-update routine. Such context-aware refinement helps resolve ambiguities that cannot be settled from the target function alone.

These insights guide the design of BinRAG, an RAG-based decompilation framework that systematically addresses the above challenges. First, BinRAG recovers semantic identifiers from stripped code and projects the result into a source-like representation via a fine-tuned LLM, denoted by Γ. Second, BinRAG filters irrelevant candidates with a structural correspondence mechanism that compares callees between query and candidate functions. Third, BinRAG recursively summarizes callee functions and incorporates this inter-procedural context into the generation prompt.

Figure 4 illustrates how these components work together using the motivating example from Figure 1a. Applying the Γ transformation yields the enriched query shown in Figure 4(a), which exhibits clear parsing logic with sign handling. With this source-like query and structural reranking, BinRAG retrieves relevant digit-parsing functions such as `p_count` (Figure 4(b)), whereas standard RAG retrieves irrelevant configuration handling functions for the same input (Figure 2). Given relevant examples as context and inter-procedural summaries revealing that `sub_ECF0` is

```c
static int Parse(const char *str,
struct strtolopt *result){
 ...
 char *ptr;
 char c;
 c = *str;
 ...
 if (c == '-')
   negative = 1, str++;
 else if (c == '+')
   base = 0, str++;
 n = (int)strtol(str, &ptr, base);
 ...
```

(a) Retrieval Query Enhanced by BinRAG

```c
static int p_count(struct parse *p) {
  int count = 0;
  int ndigits = 0;
  while (MORE() && isdigit((ut8)PEEK())
                && count <= DUPMAX) {
    count = count*10 + (GETNEXT() - '0');
    ndigits++;
  }
  REQUIRE(ndigits > 0 &&
                count <= DUPMAX,
R_REGEX_BADBR);
  return count;
}
```

(b) Retrieved Example by (a)

```c
__int64 parse_number(struct parser *parser_state){
  char *last;
  // Process sign prefix
  if (*parser_state -> cur == '-') {
    parser_state -> flag = 1;  // Set negative flag
    parser_state -> cur++;}
  if (*last) {
    overflow_handler(parser_state -> cur);
  }
  // Update parser position
  parser_state -> cur = last;
  return 1;
}
```

(c) Refined Example by BinRAG

Fig. 4. (a) shows the motivating example from Figure 1 after Γ transformations, alongside (b) the relevant retrieved source example and (c) the final refined output generated by BinRAG.

an error handler, the LLM correctly infers that the constant 45 represents the ASCII hyphen ('-'), producing the refined output shown in Figure 4(c).

3 BinRAG

This section presents the design of BinRAG. As illustrated in Figure 5, our framework implements a three-stage process that refines decompiled code into more human-readable representations.

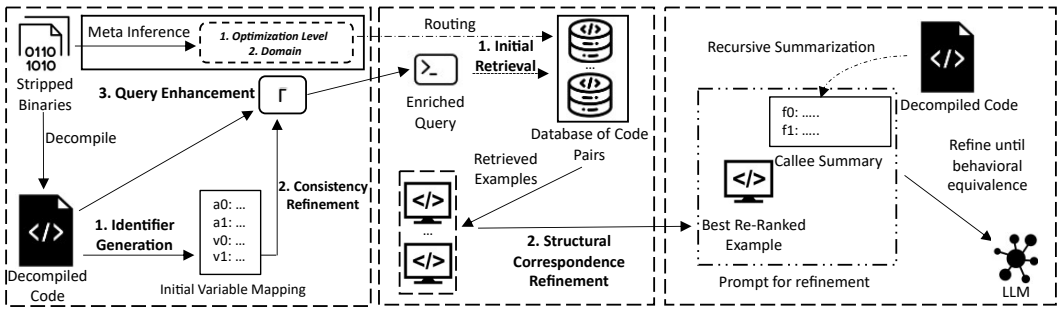


Fig. 5. The design of BinRAG.

3.1 Phase 1: Semantic Enrichment

This subsection outlines the first stage of BinRAG, which narrows the gap between retrieval in real-world stripped binaries and an *ideal* setting where source code is available. Rather than performing retrieval directly on the decompiled function f_d, we first transform it into a semantically enriched representation $f_g = \Gamma(f_d)$, where Γ is a model trained to approximate the semantic content of the original source code. Let K denote the corpus of source–decompiled function pairs. By operating on f_g instead of f_d, our retrieval process R_{BinRAG} can better approximate the ideal retrieval:

$$R_{\text{BinRAG}}(f_d, K) = R(f_g, K) \approx R(f_s, K) \tag{1}$$

3.1.1 Fine-Tuning Dataset Construction. We fine-tune Γ on binaries compiled from the Debian APT repository. We strip these binaries and decompile them to obtain function-level outputs d_i, using the corresponding source implementations s_i as supervision. Full details and leakage prevention are presented in §4.1.

3.1.2 Semantic Augmentation. Stripped decompiled code differs from human-written source along two dimensions: (1) generic placeholders replace meaningful identifiers, and (2) non-idiomatic control flow and expressions are reconstructed from machine instructions. As discussed in §2.4, the mapping between source and decompiled code is *many-to-many*. Therefore, directly fine-tuning an LLM on raw decompiled-source pairs (d_i, s_i) requires the model to simultaneously recover symbols and normalize structure, a compound task that introduces significant ambiguity. Prior work has shown that neural networks tend to memorize training data when facing such situations [19, 111].

To mitigate this risk, we draw inspiration from how human analysts approach decompiled code: rather than attempting to understand everything in a single pass, experts first infer meaningful identifiers from local context, then iteratively refine their understanding by examining cross-references throughout the program. We adopt a similar strategy by *decoupling* the problem: we first leverage specialized models to recover semantic identifiers, then train Γ to focus exclusively on structural normalization given these enriched inputs. This separation allows each component to specialize on a tractable subproblem, reducing ambiguity and improving generalization to unseen binaries. Concretely, we introduce a two-stage semantic augmentation pipeline:

① **Identifier Recovery.** For each decompiled function d_i, we first infer variable names that reflect their semantic roles. For this task, we select VarDecoder [99] because it generates high-quality semantically meaningful identifiers from decompiled code without multiple rounds of LLM interaction. After this step, VarDecoder transforms opaque identifiers (e.g., `v1`, `a2`) into semantically meaningful names (e.g., `user_buffer`, `packet_size`) by recognizing low-level code patterns and their semantic implications, a task that standard LLMs typically struggle with.

② **Identifier Consistency Refinement.** Even with meaningful identifiers, inconsistency can arise when the same program element is referenced across different contexts. For example, a variable might receive different names when analyzed as a function parameter versus within the function body. Inspired by cross-referencing techniques used by human analysts, we implement a similarity-based voting mechanism [99] to ensure that identifiers are used consistently throughout the binary. It aggregates the generated identifier names of each program element and selects the most representative name by token overlap, establishing a stable foundation for query enhancement.

3.1.3 Model Fine-Tuning. From the prior step, we construct training triples (d_i, V_i, s_i) for each decompiled function d_i, where V_i represents the recovered variable names and s_i denotes the ground-truth source code. These triples are used for fine-tuning Γ.

Training Paradigm. We employ Supervised Fine-Tuning (SFT) [55], which minimizes the negative log-likelihood of the target output s_i given the decompiled code d_i and variable names V_i as context. The model is trained using cross-entropy loss on datasets derived from real-world projects (§4.1):

$$\mathcal{L}_{\text{SFT}}(\theta) = -\sum_{i=1}^{N}\sum_{j=1}^{|s_i|}\log P(s_{i,j}|d_i, V_i; \theta) \tag{2}$$

where θ represents the model parameters, N is the number of training examples, $|s_i|$ is the token count of the source code, and $s_{i,j}$ is the j-th token of the i-th ground-truth source code. The query is used as conditioning context, while the loss is computed over the target response tokens. Compared to alternative fine-tuning methods such as RLHF [72] or DPO [74], SFT is more efficient, requiring no curated human preference dataset.

3.1.4 Query Enhancement via Learned Transformation. The training procedure described above yields a model Γ that has learned to map decompiled code, augmented with recovered identifiers, to source-like representations. At inference time, we leverage Γ to transform the input decompiled function into an enriched query suitable for retrieval.

Concretely, given a target decompiled function d_q from a stripped binary, we first apply the same identifier recovery and consistency refinement steps (①–②) used during training to obtain variable names V_q. We then invoke Γ to generate an enhanced representation $q_{gen} = \Gamma(d_q, V_q)$. Importantly, Γ is not intended to produce perfect, human-readable code. Instead, it generates an intermediate representation that is contextually rich enough for exemplar retrieval, to improve LLM reasoning.

3.2 Phase 2: Example Retrieval

While the semantically enriched query q_{gen} can serve as context for in-context learning [30], it lacks concrete examples for how source code maps to decompiled output, crucial for preventing hallucination and ensuring accurate reasoning. We therefore retrieve relevant examples from a curated knowledge base of decompiled-source code pairs. However, standard RAG pipelines that use an LLM as a second-pass filter [20] face critical limitations in this domain: function code lengths vary substantially, and processing all candidates incurs significant computational costs.

Leveraging the inter-procedural insight from § 2.5, we can filter candidates without consuming LLM context, ensuring that retrieved examples share not only semantic similarity but also comparable structural patterns. We achieve this through a two-step retrieval mechanism.

① **Initial Candidate Retrieval.** First, we search our function database, which is organized by application domain and compiler optimization level. Each retrieved candidate $c_j = (d_j, s_j)$ comprises a decompiled code snippet d_j paired with its corresponding source code s_j. We employ CodeT5+ [94] for embedding-based retrieval. [1] Using the embedding of q_{gen}, we retrieve the top-k candidates $C = \{c_1, c_2, \ldots, c_k\}$ based on maximizing cosine similarity with the source code embeddings of functions in our knowledge base: $\text{argmax}_{c_j \in K} \text{sim}(q_{gen}, s_j)$. We use q_{gen} for this step as its source-like nature is best suited for semantic matching against the source code in our database.

② **Structural Correspondence Refinement.** Even with semantically enriched queries, retrieval can be brittle if the retrieved candidates are semantically relevant but structurally different from the query. This can provide misleading context for the final code generation stage. To counter this, we re-rank the candidates to enforce structural correspondence by comparing the inter-procedural context of the original query d_q and each candidate d_j. By comparing their respective sets of called functions (callees), $C_q = \{f_{q,1}, \ldots\}$ and $C_{d_j} = \{f_{d,1}, \ldots\}$, we ensure that the retrieved examples have structural patterns that align with the query.

This alignment is formulated as a maximum weight bipartite matching problem [2], where the weight w_{ij} between a query callee $f_{q,i}$ and a candidate callee $f_{d,j}$ is defined as the cosine similarity of their code embeddings. We solve it with the Hungarian Algorithm [6], obtaining a globally optimal matching π^* that maximizes the total similarity weight between aligned callees. From this matching, we derive a normalized *Inter-Procedural Semantic Similarity* score (Sim_{call}):

$$\text{Sim}_{\text{call}}(d_q, d_j) = \frac{\sum_{(i,\ell) \in \pi^*} w_{i\ell}}{|C_q| + |C_{d_j}| - |\pi^*|}. \tag{3}$$

Here, $|\pi^*|$ denotes the number of callee pairs in the optimal matching, the numerator sums their cosine similarities, and the denominator normalizes this sum by the combined callee-set size after accounting for the matched pairs. This formulation rewards structural correspondence while penalizing architectural divergence, effectively reducing the retrieval brittleness that arises from structural mismatches between queries and candidates.

To support this retrieval process, we organize the database by application domain and compiler optimization level. These design choices aim to reduce the noise in the retrieval process, ensuring that the retrieved examples are relevant to the decompiled code under analysis.

[1] We also compared Pythia [22] and UniXcoder [36]; CodeT5+ achieved the best performance for our task.

Domain-Specific Database Organization. Domain awareness is crucial for retrieving semantically relevant examples for effective in-context learning, directly enhancing the quality of decompiler output reasoning. This approach also minimizes interference from matches that might be structurally similar but semantically distinct. To establish domain labels, two of the authors first manually classified a subset of 100 software repositories from the Debian APT repository (a diverse open-source collection spanning various domains) into eight distinct categories: System Utilities, Networking Software, Development Tools, Multimedia Software, Web and Data Services, System and Network Monitoring, Security Applications, and Scientific Computing. We then used an LLM to classify the remaining repositories across these eight software domains. The authors validated the LLM's classifications by independently reviewing a random sample and found a high level of agreement (Cohen's $\kappa = 0.825$, indicating substantial agreement). During inference, we use an LLM to classify the input binary's domain and retrieve examples from the corresponding database. We present our dataset construction in §4.1 and provide more details in [1].

Compiler Optimization Awareness. Modern compilers apply different optimization levels that drastically alter machine code structure and decompiler output [21]. Higher optimization levels involve aggressive transformations, producing decompiled code with little structural resemblance to unoptimized versions. To ensure retrieval relevance, BinRAG incorporates optimization awareness by determining the input binary's optimization level with o-glassesX [71] and retrieving examples compiled at matching levels, ensuring structural similarity between retrieved and target functions.

3.3 Phase 3: Context-Aware Refinement

The final stage generates refined decompiled code using retrieved context and inter-procedural information, then checks its behavioral consistency.

Context-Aware Summarization. Functions called within the target decompiled code provide critical contextual information, but including full implementations exceeds LLM context limits [59]. We address this by summarizing callees in topological order, starting from leaf functions and progressing upward, so that each function's summary incorporates its callees' summaries as context (see example in [15]). To handle cycles, we first identify strongly connected components using Tarjan's algorithm [80] and process them separately. We limit summarization to three levels of call depth for efficiency.

Refined Code Generation and Behavioral Consistency Checking. Equipped with retrieved examples and callee function summaries, we utilize an off-the-shelf LLM to generate a refined version d_{ref} of the original decompiled code d_q. The LLM leverages contextual examples to learn decompiled-to-source relations while incorporating inter-procedural context from function summaries. As verifying behavioral equivalence between two code snippets is undecidable in general [75], we employ MSSC [41] as a dynamic validation oracle to detect semantic inconsistencies between d_q and d_{ref} across diverse input scenarios. If MSSC detects that their executions diverge, the output is rejected, and the LLM is prompted to generate a new solution, with up to 20 retries. This verification step serves as a critical safeguard against potential hallucinations while preserving the semantic improvements achieved through our contextual restoration approach.

4 Experimental Setup

To evaluate BinRAG, we used the industry-standard decompiler, Hex-Rays IDA Pro 9.1 [4], which is widely adopted in the reverse engineering community. BinRAG is built with 3,427 lines of Python and C++ code. All experiments were conducted on a server equipped with a Ryzen 3970X 32-core processor and 256GB of memory. For the model that generates retrieval queries, we fine-tuned

the DeepSeek-R1-Distill-Qwen-14B [13] model using the LLaMA-Factory [7] framework. The fine-tuning runs for three epochs with the AdamW optimizer, using a batch size of 8 and a learning rate of 1e-5. This training was carried out on a server equipped with four NVIDIA A100 GPUs.

4.1 Datasets

Training Data. For fine-tuning our representation-enhancing model, we compile C packages from the Debian APT repository targeting the x86-64 architecture across four optimization levels (O0, O1, O2, O3). We randomly sample 100K functions with equal distribution across these optimization levels to prevent any bias towards a particular optimization level. We also use these binaries to train o-glassesX for compiler optimization prediction. Following the approach described in Section 3.1, we use VarDecoder to annotate variable names in each function. To prevent data leakage, we fine-tuned VarDecoder using the original dataset and scripts provided by its authors [10]. This annotation process enriches our training data with meaningful variable names, which is essential for our representation-enhancing model to learn effective mappings between decompiled and source code representations.

Evaluation Data. For evaluation, we compiled binaries from 5,000 libraries in the Gentoo package repository targeting the x86-64 architecture across four optimization levels (O0-O3). This produces approximately 57,000 binaries (roughly 14K per optimization level). To ensure a comprehensive and balanced assessment, we construct a diverse dataset spanning eight software domains (detailed in §3.2), randomly sampling 100 functions from each optimization level per domain. This yields a total evaluation set of 3,200 functions (4 optimization levels $\times$ 8 domains $\times$ 100 functions). We employ an 8:2 data split, allocating 80% of the compiled libraries exclusively to populate our RAG database, while reserving the remaining 20% for evaluation to ensure a clear separation between retrieval sources and test cases.

Deduplication and Data Leakage Prevention. To prevent data leakage and ensure the integrity of our evaluation, we exclude any libraries used in training from our evaluation set. We also follow established practices [104] by excluding any binary that shares more than 70% of its functions with binaries in another set (training, evaluation, or RAG database). This ensures that if a binary has significant function overlap with one set, it will not be included in the others. Function similarity is measured by comparing both function names and hashes of normalized decompiled code [104]. During hash computation, we replace all user-defined function calls with a standardized placeholder (`call_to_user_function`), ensuring a consistent comparison across different binaries.

4.2 Evaluation Metrics

We employ two metrics to assess the effectiveness of BinRAG in enhancing decompiler outputs:

Semantic accuracy. We evaluate variable name quality using semantic clustering rather than exact string matching, as semantically equivalent names may have different lexical forms [46]. Following the methodology in [46], we first tokenize and preprocess the words in names, then cluster semantically similar terms (including synonyms and abbreviations) based on their embedding similarity from CodeWordNet [46] and string distance metrics. A generated token is considered a True Positive (TP) if it belongs to the same semantic cluster as a ground-truth token. From this, we compute the standard Precision, Recall, and F1 scores.

LLM-based Semantic Assessment and Readability Evaluation. Evaluating improvements in decompiled code readability presents unique methodological challenges that existing approaches only partially address. While qualitative frameworks like [31] offer valuable insights through expert assessment, their inherent reliance on manual review limits scalability for large-scale analysis. On the other hand, automated metrics that measure structural fidelity of decompiled code primarily focus on syntax rather than semantic readability [33]. Since BinRAG aims to enhance both semantic

understanding and structural readability of decompiled code, we require an evaluation approach that can effectively assess these dual aspects while maintaining scalability.

To bridge the gap between detailed expert assessment and automated analysis, inspired by recent work that demonstrates LLMs can serve as reliable evaluation oracles for software engineering artifacts, achieving agreement comparable to human annotators [37, 42, 92, 121], we designed an LLM-based evaluation framework to assess the semantic and qualitative improvements in BinRAG's output. Our approach leverages a two-stage evaluation process with an independent LLM serving as an evaluation oracle, completely separate from BinRAG's pipeline.

Stage 1: Per-instance criteria generation. In the first stage, for each test case, we prompt the LLM to act as an expert reverse engineer, analyzing the specific original source code and its corresponding raw decompiler output. This analysis generates a set of fine-grained, customized evaluation criteria for each unique source-decompiler pair. These criteria address the particular challenges present in each case, such as logical structure issues, variable naming clarity, and semantic fidelity concerns specific to that code sample. To ensure the validity and relevance of these criteria, two authors with reverse engineering experience randomly sampled 100 instances and manually validated the generated criteria, confirming their effectiveness in assessing essential aspects of code readability (Cohen's $\kappa = 0.793$, indicating substantial agreement). Due to space limitations, we provide an example of the evaluation criteria for the motivating example (Figure 1) in [8].

Stage 2: Scoring against the generated criteria. In the second stage, the LLM oracle evaluates both the BinRAG-refined and original decompiler outputs against the established criteria. The LLM is instructed to provide detailed assessments, assigning ratings on a 4-point scale (1: Poor, 4: Excellent) for each criterion. To account for varying numbers of criteria across different samples, we calculate a normalized total score by dividing the sum of all ratings by the maximum possible score ($4 \times$ number of criteria). This normalization ensures fair comparisons across samples with differing complexity and modification requirements.

Oracle reliability. To validate that the LLM oracle is capable of capturing human readability judgments, two authors with reverse engineering experience independently applied the LLM-generated criteria to the same 100 instances used for criteria validation. We report that the two human annotators achieved weighted Cohen's $\kappa = 0.820$ (85.0% exact agreement). The Cohen's κ values between the LLM and the two human annotators are 0.740 and 0.735, with exact agreement of 78.0% and 77.5%, respectively. Overall, the LLM oracle can be considered a reliable and scalable proxy for readability judgment.

4.3 Model Setting

We employ DeepSeek-V3 [57] as the primary reasoning model for Stage 3 of BinRAG's pipeline (Section 3.3) and for inter-procedural context summarization. The model uses a temperature of 0.7 and top-p sampling of 0.9 to balance output diversity and consistency. Following VarDecoder [99], we set the maximum context length to 7,168 tokens, reserving 1,024 tokens for variable name generation to handle token limitations effectively. For example retrieval, we leverage CodeT5+ [94] embeddings, which demonstrated strong performance while maintaining high efficiency when accelerated with GPUs for call-graph matching. We set the top-k parameter to 100 to strike a balance between retrieval performance and quality. Readability assessment is performed with Gemini-2.5 Flash [3] at a temperature of 0.2, in line with prior work [92, 109] to avoid model-family bias [51]. To reduce run-to-run variability, we repeat the LLM-based evaluation five times and report the mean and standard deviation of the readability scores.

Table 1. Comparison of approach performance metrics. Readability scores are reported as mean and standard deviation.

Method	Precision	Recall	F1	Readability Score
BinRAG	0.383	0.425	0.403	**0.814 ± 0.002**
Zero-Shot	0.215	0.237	0.225	0.691 ± 0.001
RAG-baseline	0.247	0.268	0.257	0.751 ± 0.002

5 Evaluation

To comprehensively evaluate the capabilities of BinRAG, this section addresses the following research questions (RQs):

- **RQ1** (§5.1): How effectively does BinRAG enhance the readability and semantic accuracy of decompiled code compared to LLM-baselines?
- **RQ2** (§5.2): To what extent does BinRAG improve over SoTA approaches?
- **RQ3** (§5.3): What is the impact of BinRAG's specific design choices on its overall performance in code refinement?
- **RQ4** (§5.4): What are the qualitative characteristics of BinRAG's generation?
- **RQ5** (§5.5): Does BinRAG assist real-world reverse engineering?

5.1 RQ1: Effectiveness of BinRAG

In this RQ, we evaluate the effectiveness of BinRAG in enhancing the readability of stripped decompiled code. We benchmark BinRAG against two baseline settings:

- **Zero-Shot**: This baseline prompts the LLM directly with stripped decompiled code, providing no additional context. This setup evaluates the out-of-the-box performance of the LLM.
- **RAG-baseline**: This baseline represents a generic RAG system implemented using CodeSplitter [12] from LlamaIndex [17]. To enable more fine-grained retrieval, CodeSplitter chunks code into smaller segments. Notably, BinRAG does not employ this specific optimization technique.

Results. Table 1 presents the comparative performance metrics across all approaches. BinRAG achieves a precision of 0.383 and recall of 0.425, outperforming both RAG-baseline (0.247 precision, 0.268 recall) and Zero-Shot (0.215 precision, 0.237 recall) by clear margins. Specifically, BinRAG demonstrates a 55% improvement in precision and 59% improvement in recall over RAG-baseline. For readability assessment, BinRAG attains a score of 0.814, substantially higher than both Zero-Shot (0.691) and RAG-baseline (0.751); this corresponds to an 8.4% improvement over RAG-baseline ($(0.814 - 0.751)/0.751$). These results empirically validate the effectiveness of our multi-stage RAG design. We attribute these improvements to the following factors:

Limitations of Zero-Shot LLM Approaches. The Zero-Shot baseline demonstrates limited effectiveness when applied to stripped binaries, achieving only 0.215 precision and 0.237 recall. This reflects the fundamental distributional gap between LLMs' training data and decompiled code: while LLMs excel at processing human-written code with meaningful identifiers, they struggle with the characteristic features of decompiled output. In our experiments, when processing stripped code with generic identifiers (e.g., `v1`, `a2`), the model often produces outputs with minimal semantic enhancement, such as replacing generic names with equally non-specific alternatives like `buffer` or `data`. These results empirically validate our motivation for developing a specialized approach that can bridge this semantic gap through RAG.

Limitations of Generic RAG Systems. RAG-baseline demonstrates moderate improvements over zero-shot methods with 0.247 precision, 0.268 recall, and 0.751 readability score. These improvements

stem from RAG's ability to provide LLMs with examples of transformations between source and decompiled code. Nevertheless, RAG-baseline's performance remains constrained compared to BinRAG. This limitation stems primarily from the semantic gap between decompiled code and source code. Without specialized design in BinRAG, standard RAG systems cannot effectively reason about decompiled code, leading to modest improvements in semantic clarity and readability.

Performance Improvements Through Specialized Design. BinRAG's improvement over RAG-baseline stems from three design elements that address limitations in generic RAG approaches:

Better Retrieval. BinRAG's semantic enhancement improves retrieval quality. Our evaluation shows that our query transformation process consistently retrieves more relevant examples than using raw decompiled code. Specifically, the enhanced queries demonstrate closer alignment with source-level semantics. In 60% of cases, at least one-fifth of the top-50 results retrieved using enhanced queries also appear in the top-50 results when using original source code as queries. This improved retrieval provides the LLM with more relevant context, directly contributing to the precision and recall improvements observed in Table 1.

Inter-procedural Context. Our inter-procedural summaries enable the LLM to reason beyond local context, understanding the function's role within the broader binary. This approach effectively extends the context window by providing essential dependency information without including full function implementations, allowing more informed reasoning about the target function's purpose in the program structure. This contextual enhancement significantly improves precision and accuracy in variable name generation, as the LLM can leverage functional relationships to infer more appropriate semantic identifiers based on the function's actual role in the program.

Optimization-Aware Corpus Organization. Our approach organizes the retrieval corpus by both software domain and compiler optimization level, significantly improving retrieval precision. This dual categorization creates meaningful semantic boundaries that ensure retrieved examples share both functional similarity and structural characteristics with the target code. Our compiler optimization classifier, which achieves 89% accuracy, enables matching of examples with similar decompilation patterns, providing more appropriate transformation guidance while reducing semantic noise in the retrieval process.

Consistency Checking. Beyond the three design elements above, BinRAG uses MSSC [41] as a safety check against potentially semantics-altering edits. Within the configured retry budget, 3,005 of the 3,200 evaluated functions (93.9%) produced at least one refinement that passed the MSSC check. For the remaining 195 functions (6.1%), BinRAG returned the original decompiled code.

Efficiency Analysis. We evaluate both the runtime and monetary cost of BinRAG. With batch size 1, BinRAG refines an individual function in 29.66s on average. Most of the time is spent in LLM-based stages: summary generation takes 8.69s, code refinement takes 8.40s, and query enhancement takes 5.65s. Consistency checking adds 5.40s on average, while retrieval and re-ranking together add only 1.52s. To estimate deployment cost at scale, we run BinRAG end to end on nine real-world binaries totaling more than 41,000 functions, ranging in size from `jq` (781 functions) to `openssl` (9,245 functions): `htop` (851), `yara` (957), `sudo` (3,587), `re2` (4,646), `protobuf` (4,929), `krb5` (7,114), and `htslib` (9,052). We group these binaries by function count into three size buckets. Table 2 reports the input/output token usage and the average monetary cost per bucket. Across binaries of different sizes, the per-function cost remains stable at around $0.002. Overall, this cost is small compared to manual reverse engineering, which takes minutes to hours of expert effort per function [67].

Table 2. Per-bucket efficiency of BinRAG on nine real-world binaries, grouped by function count. Cost is computed using DeepSeek-V3 pricing ($0.20 and $0.77 per 1M input and output tokens, respectively).

Size bucket	Funcs.	Input	Output	Cost	Cost/Func.
Small (≤1,000)	2,589	4.5M	5.7M	$5.29	$0.002
Medium (1,001–5,000)	13,162	18.8M	26.4M	$24.09	$0.002
Large (>5,000)	25,411	40.4M	54.8M	$50.27	$0.002
Total	**41,162**	**63.7M**	**86.9M**	**$79.65**	**$0.002**

Table 3. Comparison of BinRAG against baseline approaches on variable name recovery precision, recall, and overall readability score. Readability scores are reported as mean and standard deviation. N/A indicates that the metric is inapplicable to the tool: GenNm and VarDecoder only generate variable names (not full-function rewrites), and LLM4Decompile-ref rewrites whole functions without exposing variable-name predictions amenable to our P/R/F1 evaluation.

Approach	Precision	Recall	F1	Readability Score
BinRAG	**0.383**	**0.425**	**0.403**	**0.814 ± 0.002**
DeGPT	0.197	0.208	0.202	0.563 ± 0.001
GenNm	0.292	0.282	0.287	N/A
VarDecoder	0.278	0.273	0.275	N/A
LLM4Decompile-ref	N/A	N/A	N/A	0.696 ± 0.002

5.2 RQ2: Comparison with Prior Work

In this section, we present a comparative analysis of BinRAG against state-of-the-art LLM-based approaches for decompiler output refinement, focusing on the key areas of variable name recovery and holistic code readability enhancement.

Evaluation Setup. For variable name quality assessment, we compare BinRAG with VarDecoder [99] and GenNm [104]: two specialized LLMs fine-tuned for variable name generation from stripped decompiled code. We also include DeGPT [41], a recent multi-agent framework designed for decompiler output refinement. For code readability enhancements, we benchmark against DeGPT and LLM4Decompile-ref [79], a fine-tuned LLM that specializes in improving decompiler output re-executability. We exclude traditional prediction-based approaches, as recent studies [41, 99, 104] have consistently demonstrated that LLM-based techniques achieve substantially better performance for decompiler output refinement. We also exclude DecLLM [97], a recent re-executability approach that requires executing the target function with comprehensive test cases, a requirement difficult to satisfy even with source code access [34, 35, 54, 98, 100, 101], let alone binaries.

Table 3 presents the evaluation results comparing BinRAG against different SoTA approaches for enhancing stripped decompiler output. Overall, we can see that BinRAG outperforms existing approaches in refining decompiler output.

Comparison with fine-tuned LLMs. Our analysis shows that BinRAG achieves superior variable name recovery with a precision of 0.383 and recall of 0.425, outperforming VarDecoder (precision 0.278, recall 0.273) and GenNm (precision 0.292, recall 0.282). Compared with GenNm, the next-best precision baseline, BinRAG improves precision by 31.2% ($(0.383 - 0.292)/0.292$). These results show that augmenting an off-the-shelf LLM with retrieved source-decompiled examples and inter-procedural summaries can recover better variable names than a specialized fine-tuned generator produces in isolation. The fine-tuning effort in BinRAG targets the query-enhancement model Γ, not the final-stage generator, so the refinement LLM itself remains general-purpose and replaceable (see §5.3).

Table 4. Ablation study results on different configurations of BinRAG. Readability scores are reported as mean and standard deviation.

Configuration	Precision	Recall	F1	Readability
BinRAG	**0.383**	**0.425**	**0.403**	**0.814 ± 0.002**
BinRAG-novar	0.347	0.376	0.361	0.782 ± 0.003
BinRAG-llama	0.372	0.413	0.391	0.791 ± 0.001
BinRAG-nosummary	0.322	0.357	0.339	0.776 ± 0.001
BinRAG-no-rerank	0.365	0.394	0.379	0.772 ± 0.001
BinRAG-no-Opt	0.367	0.399	0.382	0.734 ± 0.002
BinRAG-no-Domain	0.353	0.384	0.368	0.784 ± 0.002

Comparison with DeGPT and LLM4Decompile-ref. When comparing BinRAG with other LLM-based approaches, we observe notable performance differences reflecting their distinct design goals. DeGPT, with its multi-agent framework, focuses on adding explanatory comments and performing simple variable renaming without extensive code restructuring. This achieves a readability score of 0.563 and variable name recovery metrics of 0.197 (precision) and 0.208 (recall). LLM4Decompile-ref adopts a fundamentally different approach, with re-executability as its primary objective. To maintain compilation integrity, it typically makes conservative syntax modifications while preserving many identifiers from the original stripped inputs (refer to Figure 1c). Consequently, it achieves a readability score of 0.696, which is higher than DeGPT. It is important to note that LLM4Decompile-ref optimizes for re-executability rather than readability, so these results align with the expected performance characteristics of each approach.

5.3 RQ3: Ablation Study

To understand the impact of different design choices that contribute to the overall performance of BinRAG, we evaluate several major factors that could affect its effectiveness.

Semantic Enrichment. First, the impact of semantic identifier generation was assessed by fine-tuning Γ without the semantic identifiers produced by VarDecoder. In this configuration, denoted as "BinRAG-novar," the model directly processed stripped decompiler output. Table 4 shows marked decreases in both semantic accuracy and readability, confirming that identifier generation enriches decompiler output with vital semantic information, enabling representations that better mirror source code and improving retrieval of relevant examples.

LLM Backend. Second, to test whether BinRAG's effectiveness depends on a specific LLM, we substituted DeepSeek-V3 with Llama-4 Maverick [11] (BinRAG-llama). Despite a slight dip in semantic accuracy (precision 0.372, recall 0.413), this configuration remains close to the full BinRAG result (Table 4), indicating BinRAG generalizes across LLMs with comparable capabilities.

Inter-Procedural Components. Third, we evaluated the summary generation and example re-ranking components by omitting each (BinRAG-nosummary and BinRAG-no-rerank). Both configurations showed decreased performance (Table 4), with BinRAG-nosummary declining more substantially, demonstrating that local function context alone is insufficient for semantically accurate generation. Meanwhile, the drop in BinRAG-no-rerank suggests that the quality of the selected examples also affects the final refinement.

Database Organization. Finally, we evaluated domain-specific database organization (BinRAG-no-Domain) and optimization-aware matching (BinRAG-no-Opt). Both configurations showed performance decreases (Table 4), with BinRAG-no-Domain showing a greater F1 drop (0.368 vs. 0.382). These results confirm that domain-specific organization reduces noise from irrelevant retrievals, while optimization-aware matching enhances example quality.

Table 5. Representative Examples of Variable Name Predictions by BinRAG.

Ground Truth	BinRAG Prediction	Baseline
file_hdr	file_info	statbuf
stringlen	str_length	length
attrs	attribute	objects
bytes_per_sector	sector_size	length
vertices	vertex_set	facetlist
context	device_context	context
total	resource_count	manifest
certificate_value	certificate_attr	obj
data	data_ptr	cpu
signo	signal_num	sig
setting	setting_ptr	setting
is_first	is_first_iteration	do_free
streamPos	current_position	ptr

5.4 RQ4: Qualitative Analysis

Variable Name Generation Analysis. This section presents a qualitative analysis of BinRAG's output. As shown in Table 5, which samples from our evaluation, BinRAG accurately generates semantically meaningful variable names that closely match the originals. In contrast, baseline approaches often produce generic or less relevant names. BinRAG is also capable of deriving more descriptive names based on a variable's role in the function. For instance, in the `write_out_new_ascii_header` function, BinRAG generates the name `file_info` for the variable whose original name is `file_hdr`, while a baseline yields the generic `statbuf`. Examining the intermediate result reveals that for this example, BinRAG successfully retrieved similar data serialization implementations and accessed summaries of callee functions that manipulate file fields. This contextual information enabled BinRAG to infer the more descriptive name (`file_info`) that accurately reflects the variable's purpose within the function.

Failure Analysis. While BinRAG generally outperforms baseline models, insufficient information can still lead to suboptimal variable name generation. First, BinRAG often produces generic names for functions with minimal inter-procedural interactions or limited semantic cues. For example, pointer variables involved in simple buffer manipulation are frequently named `buffer` rather than a more descriptive identifier that captures their specific purpose.

Second, BinRAG struggles with domain-specific terminology, even when it successfully retrieves well-annotated context. For instance, in the `wpa_init` function from Hostapd [5], error messages clearly indicated that a pointer variable was related to the WPA protocol and passed to multiple deserialization functions. Yet, BinRAG generated the generic name `sec_ctx` rather than using protocol-specific terminology. Baseline methods also exhibit similar behaviors, producing equally generic names like `buf` and `state`. While the first type of challenge represents an inherent limitation of any identifier generation approach, the second issue could likely be addressed through domain-specific fine-tuning. However, creating such specialized datasets requires domain expertise and is beyond our current scope, presenting a promising direction for future enhancement of BinRAG.

Readability Enhancement Analysis. To assess BinRAG's effectiveness in enhancing code readability, we analyzed the LLM evaluator's comments from the readability evaluation. We categorized these comments into two groups based on their scores: a high-score set (scores 3-4, representing successful improvements) and a low-score set (scores 1-2, indicating areas for improvement). To ensure a comprehensive yet manageable analysis, we randomly sampled 200 evaluation instances from each set. Two authors independently categorized each comment by the readability aspect it

targets. Inter-rater agreement, measured by Cohen's κ, is 0.854 on the high-score set and 0.839 on the low-score set, both indicating substantial agreement. Remaining disagreements are resolved through discussion to produce the final categorization used below.

Analysis of the high-score set reveals that BinRAG successfully addresses several key readability challenges. The most significant improvements (52%) involve transforming decompiler outputs into more idiomatic C code patterns like converting `goto`-based loops into `while` or `for` loops. Another substantial category of improvements (28.5%) stems from BinRAG's effectiveness in generating semantically meaningful names for variables and data structures. Notably, even without specialized prompting, BinRAG often reconstructs structure definitions and replaces decompiler-generated types (e.g., `_DWORD`) with standard C types. BinRAG also effectively handles compiler artifacts like `__fprintf_chk` and `__stack_chk_fail` by automatically inferring their purpose and removing them. This demonstrates the effectiveness of the RAG paradigm in improving the reasoning capabilities of LLMs in decompilation outputs. The remaining improvements (19.5%) include adding clarifying comments at key decision points and extracting magic numbers into named constants, both of which enhance code comprehensibility.

Our analysis of the low-score set highlights key challenges for future work. The main issue (45%) is complex data structure recovery, especially with unresolved pointer arithmetic. Although BinRAG often reconstructs data structures, automated recovery remains an open problem [69, 116], and current methods still struggle with complex data structures and compiler-optimized memory access patterns, an inherent challenge in binary analysis that extends beyond the scope of our current work. Other challenges include semantic inaccuracies (26.5%) and miscellaneous issues (28.5%), such as insufficient comments or redundant variables. These often stem from LLM limitations, which can produce incorrect or irrelevant code even with context. Improved training and advanced LLMs may help address these problems in the future.

5.5 RQ5: User Study

Previous sections evaluated BinRAG quantitatively and qualitatively. To assess its practical utility, we conducted a user study on how effectively BinRAG assists human reverse engineers.

Participants. We recruited 12 participants for the user study. Six have at least 3 years of reverse engineering experience in CTF, industry or academia. The remaining six are CS PhD students with programming experience but limited reverse engineering backgrounds.

Study Design. We randomly selected 8 decompiled functions from the BinRAG test set. Each function was prepared under four conditions: (1) raw decompiler output, (2) DeGPT's output, (3) LLM4Decompile-ref's output, and (4) BinRAG-refined output. For every task, the participant was shown the binary together with one condition of the decompiled code, and selected from four multiple-choice descriptions of the function's intended behavior. We score each answer on a 1–4 scale, where a higher score indicates more complete understanding of the function behavior. To reduce participant fatigue while preserving balanced coverage, each participant was assigned 8 function comprehension tasks, with two tasks per condition. We randomized the assignment so that no participant saw the same function under more than one condition, while each condition received 12 ratings from experienced participants, 12 ratings from less experienced participants, and 24 ratings overall. Question samples are shown in [16].

Results. Table 6 presents the average comprehension score and task completion time across all four study conditions. All three LLM-based approaches improve over raw decompiler output in the user study. The average score rises from 2.46 on raw output to 2.79 for DeGPT, 2.83 for LLM4Decompile-ref, and 3.17 for BinRAG. The average completion time also decreases from 9.9 minutes to 8.9, 8.5,

Table 6. User study results across four study conditions.

Condition	Avg. Score (1–4)	Avg. Time (min)
BinRAG	**3.17**	**6.9**
Raw Decompiler Output	2.46	9.9
DeGPT	2.79	8.9
LLM4Decompile-ref	2.83	8.5

and 6.9 minutes, respectively. This result shows that refinement itself provides practical value for binary comprehension before we further compare the differences among refinement methods.

Among the LLM-based approaches, BinRAG shows better results in this study. It achieves a comprehension score (3.17) and shorter task completion time (6.9 minutes), scoring 0.71 points higher and finishing 30.3% faster than raw decompiler output. The gaps between BinRAG and the other two LLM-based approaches are smaller than the gap to raw output, but BinRAG remains better in both score and completion time. Participants' feedback helps explain this result. They noted that BinRAG's outputs were easier to read than stripped decompilation outputs, especially because the refined code included more meaningful variable names and more idiomatic control structures. Some participants also reported that the refined outputs reduced the amount of cross-referencing needed to confirm a function's behavior, allowing them to focus more on semantic reasoning.

Breakdown by Analyst Experience. For experienced participants, BinRAG reduced completion time by 32.1% (from 8.1 to 5.5 minutes) and raised the average score from 2.83 to 3.33. For less experienced participants, BinRAG reduced completion time by 29.1% (from 11.7 to 8.3 minutes), and the average score rose from 2.08 to 3.00. The score improvement is larger for less experienced participants (+0.92) than for experienced participants (+0.50), suggesting that the refinements are especially helpful when users have less background knowledge to recover missing semantics from raw decompiler output. Overall, BinRAG benefits participants across different experience levels and narrows the gap between less experienced reverse engineers and professionals.

6 Discussion

Threats to Validity. A primary consideration for the validity of our study is the influence of the chosen LLMs on BinRAG's performance. To address this, we use separate LLMs from different model families for refinement (DeepSeek-V3) and evaluation (Gemini-2.5 Flash), so that the readability oracle is not biased toward outputs produced by the same model used for refinement [51]. We also conducted an ablation study (see §5.3), which confirmed BinRAG's effectiveness across different language models. We further validated the oracle empirically: against two reverse-engineering-experienced human annotators on 100 sampled instances, the LLM judge achieved Cohen's $\kappa = 0.740$ and 0.735 (78.0% and 77.5% exact agreement; see §4.2), comparable to the inter-human $\kappa = 0.820$.

Another potential threat is the representativeness of our dataset, which could limit the generalizability of our findings. We mitigated this by implementing a rigorous dataset separation protocol with strict deduplication. This process prevents contamination between the training, evaluation, and retrieval sets, ensuring a robust measure of true generalization. While real-world scenarios often involve retrieving exact or near-duplicate examples, our evaluation prioritizes this strict separation to rigorously assess the model's ability to generalize to unseen data.

Decompiler Choice. Our study employs Hex-Rays, the industry-standard decompiler, to generate the decompiled code processed by BinRAG. We acknowledge that different decompilers produce varying outputs, potentially introducing bias. However, BinRAG is designed to be decompiler-agnostic. Consequently, adapting BinRAG to accommodate other decompilers is a straightforward extension of our current framework and is not expected to present significant new challenges.

Future Work. As noted in §5.4, BinRAG occasionally fails to recover names in specialized contexts. To alleviate this, one may explore building finer-grained sub-corpora keyed by library or protocol signatures, or applying domain-specific fine-tuning on top of BinRAG's existing domain-aware retrieval. Second, as mentioned in §5.4, the readability improvements of BinRAG decrease in cases tied to data-structure recovery. We envision that augmenting BinRAG with data structure recovery techniques [69, 99, 116] could further enhance readability in these cases.

7 Related Work

Decompiler Output Enhancement. A significant body of research has been dedicated to improving the quality of decompiler outputs through various paradigms. Traditional program analysis approaches have concentrated on enhancing type inference and data structure recovery using static and dynamic analysis techniques [49, 56, 65, 69, 78, 116, 117]. Meanwhile, machine learning methods have been employed to predict meaningful variable and function names from stripped decompiler outputs to improve code readability [24, 29, 46, 48, 73]. While effective, these methods often face challenges with out-of-vocabulary words and limited generalization to novel code patterns.

More recently, LLMs have found broad application in software security [25, 43, 58, 91, 113–115]. Within decompiler output enhancement specifically, studies have focused on generating variable names from stripped output [99, 104], while others have leveraged LLMs to achieve re-executability of the decompiled code [52, 62, 79, 97]. Another line of research has utilized LLMs for summarizing the functionality of binary functions [18, 41, 45, 82, 102]. Our work is orthogonal to these approaches. Rather than targeting a specific enhancement task, BinRAG enables off-the-shelf LLMs to improve the overall readability of stripped decompiled code via retrieval-augmented contextual grounding.

Binary–Source Code Matching. Matching binary functions to their source-level counterparts is a long-standing challenge in reverse engineering, with applications in software composition analysis [32, 44, 84, 85, 107, 110], patch presence detection [105, 108, 112], and binary code similarity detection [38, 53, 83, 86–89, 93, 95, 96]. Most work in this area assumes the binary is compiled from a codebase already in the candidate database, possibly with a different compiler, version, or build configuration; the goal is to identify which source produced each binary function [68]. BinaryAI [44] is a representative state-of-the-art system. It combines a transformer-based embedding model with locality-driven matching: it uses the link-time contiguity of functions compiled from the same translation unit, which gives it high precision for software composition analysis. BinRAG targets a related but distinct task. Instead of identifying the source a binary function was compiled from, BinRAG retrieves analogous implementations from independent codebases as reference context for LLM-based code refinement. In this cross-project setting, no compilation relationship holds between query and candidate, so link-time signals do not apply. BinRAG instead uses callee-set similarity via bipartite matching as a structural signal: it compares functions by what they call rather than where they sit in memory.

8 Conclusion

To address the semantic poverty problem of stripped decompiled code and the retrieval bottleneck it creates for standard RAG, we introduce BinRAG, a novel three-stage framework that enhances decompiler output readability. BinRAG utilizes specialized LLMs to generate enriched code representations, which then query an RAG system to find relevant examples from a curated knowledge base. A general-purpose LLM integrates these examples to produce more readable and semantically accurate code. Our evaluation shows that BinRAG improves readability by 8.4% over standard RAG and semantic precision by 31.2% over the next-best prior baseline, demonstrating its effectiveness in reducing the manual effort required in reverse engineering.

Data Availability

We maintain a website at [14] hosting the research artifacts.

Acknowledgements

The authors would like to thank the anonymous reviewers for their valuable comments and the support of the Collaborative Innovation Center of Novel Software Technology and Industrialization, Jiangsu, China. This work was supported in part by the Fundamental and Interdisciplinary Disciplines Breakthrough Plan of the Ministry of Education of China (No. JYB2025XDXM118), the 111 Center (No. B26023), a grant from the Research Grants Council of the Hong Kong Special Administrative Region, China (HKUST No. C6004-25G), an ITF grant under the contract ITS/161/24FP, and a grant from CMHK.

References

[1] [n. d.]. BinRAG Database Classification. https://sites.google.com/view/binrag/database.
[2] [n. d.]. CS 598CSC: Combinatorial Optimization - Lecture 10. https://courses.grainger.illinois.edu/cs598csc/sp2010/lectures/lecture10.pdf.
[3] [n. d.]. Gemini 2.5: Our most intelligent AI model. https://blog.google/innovation-and-ai/models-and-research/google-deepmind/gemini-model-thinking-updates-march-2025.
[4] [n. d.]. Hex-Rays Decompiler: Manual. https://www.hex-rays.com/products/decompiler/manual/failures.shtml.
[5] [n. d.]. Hostapd. https://wiki.gentoo.org/wiki/Hostapd.
[6] [n. d.]. Hungarian algorithm. https://en.wikipedia.org/wiki/Hungarian_algorithm.
[7] [n. d.]. LLaMA-Factory. https://github.com/hiyouga/LLaMA-Factory.
[8] [n. d.]. LLM-generated Evaluation Criteria. https://sites.google.com/view/binrag/criteria.
[9] [n. d.]. mailutils. https://mailutils.org/.
[10] [n. d.]. ReSym Artifact. https://zenodo.org/records/13923982.
[11] [n. d.]. The Llama 4 herd: The beginning of a new era of natively multimodal AI innovation. https://ai.meta.com/blog/llama-4-multimodal-intelligence/.
[12] 2025. CodeSplitter. https://developers.llamaindex.ai/python/examples/node_parsers/code_splitter_chunking/.
[13] 2025. DeepSeek-R1-Distill-Qwen-14B. https://huggingface.co/deepseek-ai/DeepSeek-R1-Distill-Qwen-14B.
[14] 2026. Artifact. https://sites.google.com/view/binrag.
[15] 2026. BinRAG Summary Example. https://sites.google.com/view/binrag/summary.
[16] 2026. BinRAG User Study. https://sites.google.com/view/binrag/user-study.
[17] 2026. Building RAG from Scratch. https://docs.llamaindex.ai/en/stable/examples/low_level/oss_ingestion_retrieval/.
[18] Ali Al-Kaswan, Toufique Ahmed, Maliheh Izadi, Anand Ashok Sawant, Premkumar Devanbu, and Arie van Deursen. 2023. Extending source code pre-trained language models to summarise decompiled binaries. In *2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 260–271. https://doi.org/10.1109/saner56733.2023.00033
[19] Devansh Arpit, Stanisław Jastrzębski, Nicolas Ballas, David Krueger, Emmanuel Bengio, Maxinder S Kanwal, Tegan Maharaj, Asja Fischer, Aaron Courville, Yoshua Bengio, et al. 2017. A closer look at memorization in deep networks. In *International conference on machine learning*. PMLR, 233–242. https://doi.org/10.48550/arxiv.1706.05394
[20] Akari Asai, Zeqiu Wu, Yizhong Wang, Avi Sil, and Hannaneh Hajishirzi. 2024. Self-rag: Learning to retrieve, generate, and critique through self-reflection. In *International conference on learning representations*. https://doi.org/10.48550/arxiv.2310.11511
[21] Zion Leonahenahe Basque, Ati Priya Bajaj, Wil Gibbs, Jude O'Kain, Derron Miao, Tiffany Bao, Adam Doupé, Yan Shoshitaishvili, and Ruoyu Wang. 2024. Ahoy sailr! there is no need to dream of c: A compiler-aware structuring algorithm for binary decompilation. In *33rd USENIX Security Symposium (USENIX Security 24)*. 361–378.
[22] Stella Biderman, Hailey Schoelkopf, Quentin Gregory Anthony, Herbie Bradley, Kyle O'Brien, Eric Hallahan, Mohammad Aflah Khan, Shivanshu Purohit, USVSN Sai Prashanth, Edward Raff, et al. 2023. Pythia: A suite for analyzing large language models across training and scaling. In *International Conference on Machine Learning*. PMLR, 2397–2430. https://doi.org/10.48550/arxiv.2304.01373
[23] Jingyi Chen, Songqiang Chen, Jialun Cao, Jiasi Shen, and Shing-Chi Cheung. 2026. When Retrieval Augmentation Meets API Documentation: Can LLMs Code with Less-Common Libraries? *ACM Trans. Softw. Eng. Methodol.* (June 2026). https://doi.org/10.1145/3821424

[24] 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. https://doi.org/10.48550/arxiv.2108.06363

[25] Xiang Chen, Anshunkang Zhou, Chengfeng Ye, and Charles Zhang. 2025. ClearAgent: Agentic Binary Analysis for Effective Vulnerability Detection. In *Proceedings of the 1st ACM SIGPLAN International Workshop on Language Models and Programming Languages*. 130–137. https://doi.org/10.1145/3759425.3763397

[26] Yufan Chen, Daoyuan Wu, Juantao Zhong, Zicheng Zhang, Debin Gao, Shuai Wang, Yingjiu Li, Ning Liu, Jiachi Chen, and Rocky KC Chang. 2025. Rethinking and Exploring String-Based Malware Family Classification in the Era of LLMs and RAG. *arXiv preprint arXiv:2507.04055* (2025).

[27] Cristina Cifuentes. 1994. *Reverse compilation techniques*. Queensland University of Technology, Brisbane.

[28] Cristina Cifuentes and K. John Gough. 1995. Decompilation of Binary Programs. *Softw. Pract. Exper.* 25, 7 (July 1995), 811–829. https://doi.org/10.1002/spe.4380250706

[29] 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. https://doi.org/10.1145/3428293

[30] Qingxiu Dong, Lei Li, Damai Dai, Ce Zheng, Zhiyong Wu, Baobao Chang, Xu Sun, Jingjing Xu, and Zhifang Sui. 2022. A survey on in-context learning. *arXiv preprint arXiv:2301.00234* (2022).

[31] Luke Dramko, Jeremy Lacomis, Edward J Schwartz, Bogdan Vasilescu, and Claire Le Goues. 2024. A taxonomy of C decompiler fidelity issues. In *33rd USENIX Security Symposium (USENIX Security 24)*. 379–396.

[32] Ruian Duan, Ashish Bijlani, Meng Xu, Taesoo Kim, and Wenke Lee. 2017. Identifying open-source license violation and 1-day security risk at large scale. In *Proceedings of the 2017 ACM SIGSAC Conference on computer and communications security*. 2169–2185. https://doi.org/10.1145/3133956.3134048

[33] Haeun Eom, Dohee Kim, Sori Lim, Hyungjoon Koo, and Sungjae Hwang. 2024. R2i: A relative readability metric for decompiled code. *Proceedings of the ACM on Software Engineering* 1, FSE (2024), 383–405. https://doi.org/10.1145/3643744

[34] Gordon Fraser and Andrea Arcuri. 2014. A large-scale evaluation of automated unit test generation using evosuite. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 24, 2 (2014), 1–42. https://doi.org/10.1145/2685612

[35] Gordon Fraser and Andreas Zeller. 2010. Mutation-driven generation of unit tests and oracles. In *Proceedings of the 19th international symposium on Software testing and analysis*. 147–158. https://doi.org/10.1145/1831708.1831728

[36] Daya Guo, Shuai Lu, Nan Duan, Yanlin Wang, Ming Zhou, and Jian Yin. 2022. Unixcoder: Unified cross-modal pre-training for code representation. *arXiv preprint arXiv:2203.03850* (2022).

[37] Junda He, Jieke Shi, Terry Yue Zhuo, Christoph Treude, Jiamou Sun, Zhenchang Xing, Xiaoning Du, and David Lo. 2025. From code to courtroom: Llms as the new software judges. *arXiv preprint arXiv:2503.02246* (2025). https://doi.org/10.48550/arxiv.2503.02246

[38] Armijn Hemel, Karl Trygve Kalleberg, Rob Vermaas, and Eelco Dolstra. 2011. Finding software license violations through binary code clone detection. In *Proceedings of the 8th Working Conference on Mining Software Repositories*. 63–72. https://doi.org/10.1145/1985441.1985453

[39] SA Hex-Rays. 2014. IDA Pro: a cross-platform multi-processor disassembler and debugger.

[40] 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. https://doi.org/10.1145/2902362

[41] Peiwei Hu, Ruigang Liang, and Kai Chen. 2024. DeGPT: Optimizing Decompiler Output with LLM. In *NDSS*. https://doi.org/10.14722/ndss.2024.24401

[42] Zhenlan Ji, Pingchuan Ma, Zongjie Li, Zhaoyu Wang, and Shuai Wang. 2025. Causality-Aided Evaluation and Explanation of Large Language Model-Based Code Generation. *Proc. ACM Softw. Eng.* 2, ISSTA, Article ISSTA061 (June 2025), 24 pages. https://doi.org/10.1145/3728938

[43] Zimo Ji, Daoyuan Wu, Wenyuan Jiang, Pingchuan Ma, Zongjie Li, and Shuai Wang. 2025. Measuring and Augmenting Large Language Models for Solving Capture-the-Flag Challenges. In *Proceedings of the 2025 ACM SIGSAC Conference on Computer and Communications Security* (Taipei, Taiwan) *(CCS '25)*. Association for Computing Machinery, New York, NY, USA, 603–617. https://doi.org/10.1145/3719027.3744855

[44] Ling Jiang, Junwen An, Huihui Huang, Qiyi Tang, Sen Nie, Shi Wu, and Yuqun Zhang. 2024. Binaryai: binary software composition analysis via intelligent binary source code matching. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*. 1–13. https://doi.org/10.1145/3597503.3639100

[45] Linxi Jiang, Xin Jin, and Zhiqiang Lin. 2025. Beyond Classification: Inferring Function Names in Stripped Binaries via Domain Adapted LLMs.. In *NDSS*. https://doi.org/10.14722/ndss.2025.240797

[46] 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 *CCS*. https://doi.org/10.1145/3548606.3560612

[47] Vladimir Karpukhin, Barlas Oguz, Sewon Min, Patrick SH Lewis, Ledell Wu, Sergey Edunov, Danqi Chen, and Wen-tau Yih. 2020. Dense Passage Retrieval for Open-Domain Question Answering.. In *EMNLP (1)*. 6769–6781. https://doi.org/10.18653/v1/2020.emnlp-main.550

[48] 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 *ASE*. https://doi.org/10.1109/ase.2019.00064

[49] JongHyup Lee, Thanassis Avgerinos, and David Brumley. 2011. TIE: Principled Reverse Engineering of Types in Binary Programs. In *NDSS*.

[50] Patrick Lewis, Ethan Perez, Aleksandra Piktus, Fabio Petroni, Vladimir Karpukhin, Naman Goyal, Heinrich Küttler, Mike Lewis, Wen-tau Yih, Tim Rocktäschel, et al. 2020. Retrieval-augmented generation for knowledge-intensive nlp tasks. *Advances in Neural Information Processing Systems* 33 (2020), 9459–9474. https://doi.org/10.48550/arxiv.2005.11401

[51] Qingquan Li, Shaoyu Dou, Kailai Shao, Chao Chen, and Haixiang Hu. 2026. Evaluating Scoring Bias in LLM-as-a-Judge. (2026), 19–34. https://doi.org/10.1007/978-981-92-0372-7_2

[52] Weichen Li, Albert Jan, Baishakhi Ray, Chengzhi Mao, Junfeng Yang, and Kexin Pei. 2025. EditLord: Learning Code Transformation Rules for Code Editing. *ICML* (2025).

[53] Zongjie Li, Pingchuan Ma, Huaijin Wang, Shuai Wang, Qiyi Tang, Sen Nie, and Shi Wu. 2022. Unleashing the power of compiler intermediate representation to enhance neural program embeddings. In *ICSE*. https://doi.org/10.1145/3510003.3510217

[54] Zongjie Li, Chaozheng Wang, Zhibo Liu, Haoxuan Wang, Dong Chen, Shuai Wang, and Cuiyun Gao. 2023. CCTEST: Testing and Repairing Code Completion Systems. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1238–1250. https://doi.org/10.1109/icse48619.2023.00110

[55] Zongjie Li, Daoyuan Wu, Shuai Wang, and Zhendong Su. 2025. API-Guided Dataset Synthesis to Finetune Large Code Models. *Proc. ACM Program. Lang.* 9, OOPSLA1, Article 108 (April 2025), 30 pages. https://doi.org/10.1145/3720449

[56] Zhiqiang Lin, Xiangyu Zhang, and Dongyan Xu. 2010. Automatic reverse engineering of data structures from binary execution. In *Proceedings of the 11th Annual Information Security Symposium*. 1–1.

[57] Aixin Liu, Bei Feng, Bing Xue, Bingxuan Wang, Bochao Wu, Chengda Lu, Chenggang Zhao, Chengqi Deng, Chenyu Zhang, Chong Ruan, et al. 2024. Deepseek-v3 technical report. *arXiv preprint arXiv:2412.19437* (2024). https://doi.org/10.48550/arxiv.2412.19437

[58] Han Liu, Daoyuan Wu, Yuqiang Sun, Shuai Wang, and Yang Liu. 2025. Have We Solved Access Control Vulnerability Detection in Smart Contracts? A Benchmark Study. In *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE Press. https://doi.org/10.1109/ASE63991.2025.00166

[59] Nelson F Liu, Kevin Lin, John Hewitt, Ashwin Paranjape, Michele Bevilacqua, Fabio Petroni, and Percy Liang. 2023. Lost in the middle: How language models use long contexts. *arXiv preprint arXiv:2307.03172* (2023). https://doi.org/10.48550/arxiv.2307.03172

[60] Yibo Liu, Zion Leonahenahe Basque, Arvind S Raj, Chavin Udomwongsa, Chang Zhu, Jie Hu, Changyu Zhao, Fangzhou Dong, Adam Doupé, Tiffany Bao, et al. 2026. Oxidizer: Toward Concise and High-fidelity Rust Decompilation. In *2026 IEEE Symposium on Security and Privacy (SP)*. IEEE, 3433–3451. https://doi.org/10.1109/sp63933.2026.00257

[61] Ye Liu, Yue Xue, Daoyuan Wu, Yuqiang Sun, Yi Li, Miaolei Shi, and Yang Liu. 2025. Propertygpt: Llm-driven formal verification of smart contracts through retrieval-augmented property generation. In *NDSS*. https://doi.org/10.14722/ndss.2025.241357

[62] Zhibo Liu, Huaijin Wang, Wai Kin Wong, Daoyuan Wu, and Shuai Wang. 2026. No More Translation at Runtime: LLM-Empowered Static Binary Translation. In *Proceedings of the 21st European Conference on Computer Systems, EuroSys 2026, McEwan Hall/The University of Edinburgh, Edinburgh, Scotland, UK, April 27-30, 2026*. ACM, 1023–1040. https://doi.org/10.1145/3767295.3803600

[63] Zhibo Liu and Shuai Wang. 2020. How far we have come: testing decompilation correctness of C decompilers. *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis* (2020). https://doi.org/10.1145/3395363.3397370

[64] Zhibo Liu, Yuanyuan Yuan, Shuai Wang, and Yuyan Bao. 2022. Sok: Demystifying binary lifters through the lens of downstream applications. In *2022 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1100–1119. https://doi.org/10.1109/sp46214.2022.9833799

[65] Zhibo Liu, Yuanyuan Yuan, Shuai Wang, Xiaofei Xie, and Lei Ma. 2023. Decompiling x86 deep neural network executables. In *32nd USENIX Security Symposium (USENIX Security 23)*. 7357–7374. https://doi.org/10.48550/arxiv.2210.01075

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

[67] Alessandro Mantovani, Luca Compagna, Yan Shoshitaishvili, and Davide Balzarotti. 2022. The Convergence of Source Code and Binary Vulnerability Discovery–A Case Study *(AsiaCCS)*. https://doi.org/10.1145/3488932.3497764

[68] Andrea Marcelli, Mariano Graziano, Xabier Ugarte-Pedrero, Yanick Fratantonio, Mohamad Mansouri, and Davide Balzarotti. 2022. How Machine Learning Is Solving the Binary Function Similarity Problem. In *31st USENIX Security Symposium (USENIX Security 22)*. USENIX Association, Boston, MA, 2099–2116. https://www.usenix.org/conference/usenixsecurity22/presentation/marcelli

[69] Matt Noonan, Alexey Loginov, and David Cok. 2016. Polymorphic Type Inference for Machine Code. In *PLDI*. https://doi.org/10.1145/2980983.2908119

[70] National Security Agency (NSA). 2018. Ghidra. https://www.nsa.gov/resources/everyone/ghidra/.

[71] Yuhei Otsubo, Akira Otsuka, Mamoru Mimura, Takeshi Sakaki, and Hiroshi Ukegawa. 2020. o-glassesX: Compiler provenance recovery with attention mechanism from a short code fragment. In *Proc. Workshop Binary Anal. Res.* 1–12. https://doi.org/10.14722/bar.2020.23001

[72] Long Ouyang, Jeffrey Wu, Xu Jiang, Diogo Almeida, Carroll Wainwright, Pamela Mishkin, Chong Zhang, Sandhini Agarwal, Katarina Slama, Alex Ray, et al. 2022. Training language models to follow instructions with human feedback. *Advances in neural information processing systems* 35 (2022), 27730–27744. https://doi.org/10.52202/068431-2011

[73] Kuntal Kumar Pal, Ati Priya Bajaj, Pratyay Banerjee, Audrey Dutcher, Mutsumi Nakamura, Zion Leonahenahe Basque, Himanshu Gupta, Saurabh Arjun Sawant, Ujjwala Anantheswaran, Yan Shoshitaishvili, et al. 2024. "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)*. https://doi.org/10.1109/sp54263.2024.00152

[74] Rafael Rafailov, Archit Sharma, Eric Mitchell, Stefano Ermon, Christopher D Manning, and Chelsea Finn. 2023. Direct preference optimization: Your language model is secretly a reward model. *arXiv preprint arXiv:2305.18290* (2023).

[75] Henry Gordon Rice. 1953. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical society* 74, 2 (1953), 358–366. https://doi.org/10.1090/s0002-9947-1953-0053041-6

[76] Stephen Robertson, Hugo Zaragoza, et al. 2009. The probabilistic relevance framework: BM25 and beyond. *Foundations and Trends® in Information Retrieval* 3, 4 (2009), 333–389. https://doi.org/10.1561/1500000019

[77] Edward J Schwartz, J Lee, Maverick Woo, and David Brumley. 2013. Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring. In *Proceedings of the 22nd USENIX Security Symposium (Washington DC, USA, 2013), USENIX Association*.

[78] Asia Slowinska, Traian Stancescu, and Herbert Bos. 2011. Howard: A Dynamic Excavator for Reverse Engineering Data Structures.. In *NDSS*.

[79] Hanzhuo Tan, Qi Luo, Jing Li, and Yuqun Zhang. 2024. Llm4decompile: Decompiling binary code with large language models. *arXiv preprint arXiv:2403.05286* (2024).

[80] Robert Tarjan. 1972. Depth-first search and linear graph algorithms. *SIAM journal on computing* 1, 2 (1972), 146–160.

[81] Chaozheng Wang, Zezhou Yang, Shuzheng Gao, Cuiyun Gao, Ting Peng, Hailiang Huang, Yuetang Deng, and Michael Lyu. 2025. RAG or Fine-tuning? A Comparative Study on LCMs-based Code Completion in Industry. *arXiv preprint arXiv:2505.15179* (2025). https://doi.org/10.48550/arxiv.2505.15179

[82] Hao Wang, Zeyu Gao, Chao Zhang, Zihan Sha, Mingyang Sun, Yuchen Zhou, Wenyu Zhu, Wenju Sun, Han Qiu, and Xi Xiao. 2024. Clap: Learning transferable binary code representations with natural language supervision. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 503–515. https://doi.org/10.1145/3650212.3652145

[83] Huaijin Wang and Zhiqiang Lin. [n. d.]. vSim: Semantics-Aware Value Extraction for Efficient Binary Code Similarity Analysis. In *33rd Annual Network and Distributed System Security Symposium, NDSS 2026, San Diego, California, USA*. https://doi.org/10.14722/ndss.2026.240213

[84] Huaijin Wang, Zhibo Liu, Yanbo Dai, Shuai Wang, Qiyi Tang, Sen Nie, and Shi Wu. 2025. Preserving Privacy in Software Composition Analysis: A Study of Technical Solutions and Enhancements. In *ICSE*. https://doi.org/10.1109/icse55347.2025.00055

[85] Huaijin Wang, Zhibo Liu, Shuai Wang, Ying Wang, Qiyi Tang, Sen Nie, and Shi Wu. 2024. Are We There Yet? Filling the Gap Between Binary Similarity Analysis and Binary Software Composition Analysis. In *IEEE EuroS&P '24*. https://doi.org/10.1109/eurosp60621.2024.00034

[86] Huaijin Wang, Pingchuan Ma, Shuai Wang, Qiyi Tang, Sen Nie, and Shi Wu. 2023. sem2vec: Semantics-aware assembly tracelet embedding. *ACM Transactions on Software Engineering and Methodology* (2023). https://doi.org/10.1145/3569933

[87] Huaijin Wang, Pingchuan Ma, Yuanyuan Yuan, Zhibo Liu, Shuai Wang, Qiyi Tang, Sen Nie, and Shi Wu. 2022. Enhancing dnn-based binary code function search with low-cost equivalence checking. *IEEE TSE* (2022). https://doi.org/10.1109/tse.2022.3149240

[88] Hao Wang, Wenjie Qu, Gilad Katz, Wenyu Zhu, Zeyu Gao, Han Qiu, Jianwei Zhuge, and Chao Zhang. 2022. jTrans: Jump-Aware Transformer for Binary Code Similarity. *arXiv preprint arXiv:2205.12713* (2022). https://doi.org/10.48550/

arxiv.2205.12713

[89] Huaijin Wang and Shuai Wang. 2026. Instruction Alignment for Binary Code Representation Learning. In *ASE*. https://doi.org/10.1145/3832783.3837516

[90] Huaijin Wang, Shuai Wang, Dongpeng Xu, Xiangyu Zhang, and Xiao Liu. 2020. Generating effective software obfuscation sequences with reinforcement learning. *IEEE Transactions on Dependable and Secure Computing* (2020). https://doi.org/10.1109/tdsc.2020.3041655

[91] Liwen Wang, Wenxuan Wang, Shuai Wang, Zongjie Li, Zhenlan Ji, Zongyi Lyu, Daoyuan Wu, and Shing-Chi Cheung. 2025. IP Leakage Attacks Targeting LLM-Based Multi-Agent Systems. *arXiv preprint arXiv:2505.12442* (2025). https://doi.org/10.48550/arxiv.2505.12442

[92] Ruiqi Wang, Jiyu Guo, Cuiyun Gao, Guodong Fan, Chun Yong Chong, and Xin Xia. 2025. Can llms replace human evaluators? an empirical study of llm-as-a-judge in software engineering. *Proceedings of the ACM on Software Engineering* 2, ISSTA (2025), 1955–1977. https://doi.org/10.1145/3728963

[93] Shuai Wang and Dinghao Wu. 2017. In-memory fuzzing for binary code similarity analysis. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 319–330. https://doi.org/10.1109/ase.2017.8115645

[94] Yue Wang, Hung Le, Akhilesh Deepak Gotmare, Nghi DQ Bui, Junnan Li, and Steven CH Hoi. 2023. Codet5+: Open code large language models for code understanding and generation. *arXiv preprint arXiv:2305.07922* (2023).

[95] Wai Kin Wong, Huaijin Wang, Zongjie Li, and Shuai Wang. 2024. Binaug: Enhancing binary similarity analysis with low-cost input repairing. In *Proceedings of the 46th IEEE/ACM International Conference on Software Engineering*. https://doi.org/10.1145/3597503.3623328

[96] Wai Kin Wong, Huaijin Wang, Pingchuan Ma, Shuai Wang, Mingyue Jiang, Tsong Yueh Chen, Qiyi Tang, Sen Nie, and Shi Wu. 2022. Deceiving deep neural networks-based binary code matching with adversarial programs. In *2022 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 117–128. https://doi.org/10.1109/ICSME55016.2022.00019

[97] Wai Kin Wong, Daoyuan Wu, Huaijin Wang, Zongjie Li, Zhibo Liu, Shuai Wang, Qiyi Tang, Sen Nie, and Shi Wu. 2025. DecLLM: LLM-Augmented Recompilable Decompilation for Enabling Programmatic Use of Decompiled Code. *Proceedings of the ACM on Software Engineering* 2, ISSTA (2025), 1841–1864. https://doi.org/10.1145/3728958

[98] Wai Kin Wong, Dongwei Xiao, Cheuk Tung Lai, Yiteng Peng, Daoyuan Wu, and Shuai Wang. 2025. Extraction and Mutation at a High Level: Template-Based Fuzzing for JavaScript Engines. *Proc. ACM Program. Lang.* 9, OOPSLA2, Article 376 (Oct. 2025), 29 pages. https://doi.org/10.1145/3763154

[99] 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. https://doi.org/10.1145/3658644.3670340

[100] Yuchong Xie, Kaikai Zhang, Yu Liu, Rundong Yang, Ping Chen, Shuai Wang, and Dongdong She. 2026. RandSet: Randomized Corpus Reduction for Fuzzing Seed Scheduling. *Proceedings of the ACM on Programming Languages* 10, OOPSLA1 (2026), 1570–1598. https://doi.org/10.1145/3798257

[101] Yuchong Xie, Wenhui Zhang, and Dongdong She. 2025. ZTaint-Havoc: From Havoc Mode to Zero-Execution Fuzzing-Driven Taint Inference. *Proc. ACM Softw. Eng.* 2, ISSTA, Article ISSTA041 (June 2025), 23 pages. https://doi.org/10.1145/3728916

[102] Jiaqi Xiong, Guoqiang Chen, Kejiang Chen, Han Gao, Shaoyin Cheng, and Weiming Zhang. 2023. Hext5: Unified pre-training for stripped binary code information inference. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 774–786. https://doi.org/10.1109/ase56229.2023.00099

[103] Ling Xu, Huanhuan Yang, Chao Liu, Jianhang Shuai, Meng Yan, Yan Lei, and Zhou Xu. 2021. Two-stage attention-based model for code search with textual and structural features. In *2021 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 342–353. https://doi.org/10.1109/saner50967.2021.00039

[104] Xiangzhe Xu, Zhuo Zhang, Zian Su, Ziyang Huang, Shiwei Feng, Yapeng Ye, Nan Jiang, Danning Xie, Siyuan Cheng, Lin Tan, and Xiangyu Zhang. 2025. Unleashing the Power of Generative Model in Recovering Variable Names from Stripped Binary. In *NDSS*. https://doi.org/10.14722/ndss.2025.240276

[105] Yifei Xu, Zhengzi Xu, Bihuan Chen, Fu Song, Yang Liu, and Ting Liu. 2020. Patch based vulnerability matching for binary programs. In *ISSTA '20: 29th ACM SIGSOFT International Symposium on Software Testing and Analysis, Virtual Event, USA, July 18-22, 2020*. ACM. https://doi.org/10.1145/3395363.3397361

[106] Khaled Yakdan, Sebastian Eschweiler, Elmar Gerhards-Padilla, and Matthew Smith. 2015. No More Gotos: Decompilation Using Pattern-Independent Control-Flow Structuring and Semantic-Preserving Transformations.. In *NDSS*. https://doi.org/10.14722/ndss.2015.23185

[107] Can Yang, Zhengzi Xu, Hongxu Chen, Yang Liu, Xiaorui Gong, and Baoxu Liu. 2022. ModX: binary level partially imported third-party library detection via program modularization and semantic matching. In *Proceedings of the 44th International Conference on Software Engineering*. 1393–1405. https://doi.org/10.1145/3510003.3510627

[108] Chengfeng Ye, Anshunkang Zhou, and Charles Zhang. [n. d.]. Enhancing Semantic-Aware Binary Diffing with High-Confidence Dynamic Instruction Alignment. In *33rd Annual Network and Distributed System Security Symposium, NDSS 2026, San Diego, California, USA*. https://doi.org/10.14722/ndss.2026.240663

[109] Peiwen Yuan, Shaoxiong Feng, Yiwei Li, Xinglin Wang, Boyuan Pan, Heda Wang, Yao Hu, and Kan Li. 2024. Batcheval: Towards human-like text evaluation. In *ACL*. https://doi.org/10.18653/v1/2024.acl-long.846

[110] Zimu Yuan, Muyue Feng, Feng Li, Gu Ban, Yang Xiao, Shiyang Wang, Qian Tang, He Su, Chendong Yu, Jiahuan Xu, et al. 2019. B2SFinder: Detecting open-source software reuse in COTS software. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1038–1049. https://doi.org/10.1109/ase.2019.00100

[111] Chiyuan Zhang, Samy Bengio, Moritz Hardt, Benjamin Recht, and Oriol Vinyals. 2016. Understanding deep learning requires rethinking generalization. *arXiv preprint arXiv:1611.03530* (2016). https://doi.org/10.48550/arxiv.1611.03530

[112] Hang Zhang and Zhiyun Qian. 2018. Precise and accurate patch presence test for binaries. In *USENIX Security*.

[113] Kunpeng Zhang, Zongjie Li, Daoyuan Wu, Shuai Wang, and Xin Xia. 2025. Low-Cost and Comprehensive Non-textual Input Fuzzing with LLM-Synthesized Input Generators. In *34th USENIX Security Symposium, USENIX Security 2025, Seattle, WA, USA, August 13-15, 2025*. USENIX Association, 6999–7018.

[114] Kunpeng Zhang, Shuai Wang, Jitao Han, Xiaogang Zhu, Xian Li, Shaohua Wang, and Sheng Wen. 2025. Your Fix Is My Exploit: Enabling Comprehensive DL Library API Fuzzing with Large Language Models. In *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering* (Ottawa, Ontario, Canada) *(ICSE '25)*. 13 pages. https://doi.org/10.1109/icse55347.2025.00041

[115] Kunpeng Zhang, Dongwei Xiao, Daoyuan Wu, Shuai Wang, Jiali Zhao, Yuanyi Lin, Tongtong Xu, and Shaohua Wang. 2026. LLM-Powered Silent Bug Fuzzing in Deep Learning Libraries via Versatile and Controlled Bug Transfer. *Proceedings of the ACM on Programming Languages* 10, OOPSLA1 (2026), 1599–1626. https://doi.org/10.1145/3798258

[116] 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. https://doi.org/10.1109/sp40001.2021.00051

[117] Zhuo Zhang, Wei You, Guanhong Tao, Guannan Wei, Yonghwi Kwon, and Xiangyu Zhang. 2019. BDA: practical dependence analysis for binary executables by unbiased whole-program path sampling and per-path abstract interpretation. *Proceedings of the ACM on Programming Languages* 3, OOPSLA (2019), 1–31. https://doi.org/10.1145/3360563

[118] Zhaowei Zhang, Hongyu Zhang, Beijun Shen, and Xiaodong Gu. 2022. Diet code is healthy: Simplifying programs for pre-trained models of code. In *FSE*. https://doi.org/10.1145/3540250.3549094

[119] Anshunkang Zhou, Chengfeng Ye, Heqing Huang, Yuandao Cai, and Charles Zhang. 2024. Plankton: Reconciling Binary Code and Debug Information. In *ASPLOS*. https://doi.org/10.1145/3620665.3640382

[120] Anshunkang Zhou and Charles Zhang. 2026. Diatom: Polylithic Binary Lifting with Data-Flow Summaries and Type-Aware IR Linking. *Proceedings of the ACM on Programming Languages* 10, OOPSLA1 (2026), 170–200. https://doi.org/10.1145/3798206

[121] Xin Zhou, Kisub Kim, Ting Zhang, Martin Weyssow, Luís F. Gomes, Guang Yang, Kui Liu, Xin Xia, and David Lo. 2025. SE-Jury: An LLM-as-Ensemble-Judge Metric for Narrowing the Gap with Human Evaluation in SE. In *2025 40th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. https://doi.org/10.1109/ase63991.2025.00214

Received 2026-01-30; accepted 2026-06-25