R^3-Bench: Reproducible Real-world Reverse Engineering Dataset for Symbol Recovery

Muzhi Yu*†, Zhengran Zeng*, Wei Ye*, Jinan Sun*, Xiaolong Bai†, Shikun Zhang*
*Peking University †Alibaba Group
muzhi.yu@pku.edu.cn, zhengranzeng@stu.pku.edu.cn, wye@pku.edu.cn,
sjn@pku.edu.cn, bxl1989@gmail.com, zhangsk@pku.edu.cn

***Abstract*—Symbol recovery in reverse engineering is crucial for restoring variable and data structure information in compiled binaries. While learning-based methods have shown promise in recovering both semantic information (names and types) and syntactic information (shapes), they require comprehensive datasets where expressions in binary code are precisely aligned with their source code equivalents. Current techniques for generating such alignments struggle with complex data access patterns, resulting in incomplete training data and consequently hampering model performance and recovery accuracy. We present AST-Align, a novel technique unifying alignment of variables and struct access expressions across multiple architectures (x86 and ARM) and languages (C/C++/Rust). AST-Align significantly improves the number of generated ground truths, capturing four times more struct fields than previous methods. Using this algorithm, we develop R3-Bench, a metadata-rich, extensible dataset with explicit project inclusion criteria and reproducible processing pipeline, comprising over 10 million functions across multiple architectures. Our evaluation establishes baseline performance by testing various approaches from n-gram models to Large Language Models. The results show that while general LLMs initially perform poorly, their effectiveness dramatically improves with proper demonstration. R3-Bench provides a robust foundation for assessing model capabilities and serves as a valuable reference for future symbol recovery research.**

I. Introduction

Reverse engineering (RE) of compiled software is fundamental to vulnerability discovery [1], [2], malware analysis [3], [4], and software supply chain security [5], [6], [7], [8]. A central challenge in RE is recovering high-level semantic information—descriptive variable names, types, and data structure layouts—that gets stripped during compilation. This process, known as symbol recovery, significantly enhances code comprehension and enables more effective downstream analyses.

Modern decompilers like IDA [9] and Ghidra [10] can effectively reconstruct control flow structures from binaries, producing functionally equivalent code with accurate branching and loop patterns. However, they typically fail to recover semantic elements that make code truly comprehensible to humans. Variables are assigned generic identifiers (e.g., "v1", "a2") rather than meaningful names, and complex data structures are reduced to cryptic memory offsets. For example, readable expressions like `ctxa->pk_info->verify_func` become `*(QWORD *)(*v12 + 32LL)`, significantly impeding code comprehension.

Machine learning techniques have advanced from basic syntactic recovery [11], [12] toward semantic recovery of meaningful names and types [13], [14], [15], [16], evolving from classification-based models to transformer architectures and generative models that can capture complex relationships between program elements.

Despite significant progress, existing research faces fundamental limitations in dataset quality that constrain further advancement. Training effective symbol recovery models requires establishing learnable fine-grained alignment between stripped binaries and their semantic equivalents. Function-level labeling proves inadequate because compiler optimizations blur function boundaries through inlining, while providing no fine-grained symbol alignment. Symbol-level labeling struggles with struct pointers, as the semantic gap between source-level field accesses and their compiled memory offset representations breaks traditional variable tracking mechanisms. Expression-level labeling appears promising but faces technical challenges in DWARF-based resolution of complex indirect dereferences (e.g., `tab_array->tabs[tab_index].decimal_point`), where existing academic DWARF parsers support limited specification subsets and exhibit poor cross-architecture compatibility, potentially missing substantial recoverable information that dominates real-world data structure usage patterns.

Additionally, the field suffers from fragmented dataset construction methodologies, with each study developing project-specific approaches. A fundamental contributor to this fragmentation is the inherently uncontrollable nature of building binaries from source code [17] —researchers often adopt an opportunistic corpus selection strategy, including whatever projects successfully compile rather than following systematic inclusion criteria. This ad-hoc approach compromises evaluation rigor and reproducibility. The resulting fragmentation prevents comprehensive comparison with previous work, and most datasets lack critical metadata including source code versions and build configurations, making extension or reproduction extremely challenging.

To address these challenges, we introduce R3-Bench, a comprehensive dataset with novel alignment techniques and reproducible construction methodology. Our core contributions are:

This work is done during internship at Alibaba Group.
Wei Ye and Jinan Sun are co-corresponding authors.

2643-1572/25/$31.00 ©2025 IEEE
DOI 10.1109/ASE63991.2025.00026

- **AST-Align: Unified Expression Ground Truth Alignment:** A novel algorithm that unifies ground truth generation for variables and field accesses through expression-level alignment comparing ASTs from decompiled code with and without debug information. AST-Align comprehensively aligns complex symbol manipulation expressions, including complex indirect dereferences (e.g., `tab_array->tabs[tab_index].decimal_point`), significantly increasing recoverable ground truth for struct members (recovering 4x more information than prior work).
- **Large-Scale, Reproducible Benchmark Dataset:** R3-Bench[1] is built using the Nix [18] package manager for automated build of complex projects and high reproducibility, containing over 10 million functions across x86 and ARM architectures from thousands of real-world C/C++/Rust projects, with rich metadata facilitating extensions and future research.
- **Comprehensive Baseline Evaluation:** We establish baseline performance by evaluating methods from n-gram models to Large Language Models, demonstrating the dataset's utility in assessing model capabilities, and providing a valuable reference point for future symbol recovery research, indicating that generalization performance scales with model size.

The remainder of this paper is organized as follows: Section II provides background information. Section III details our AST-Align algorithm. Section IV describes the dataset construction methodology and statistics. Section V presents both the effectiveness of our AST-Align algorithm and baseline model performance. Finally, Section VI discusses limitations and future work, and Section VII concludes the paper.

II. Background

Symbol recovery in reverse engineering aims to restore variable names, types, and data structure information lost during compilation. This process encompasses variable name recovery, type recovery, data structure recovery, and expression recovery—tasks that are interconnected yet present distinct challenges, with data structure recovery typically requiring analysis across multiple functions.

Creating datasets for symbol recovery research requires establishing ground truth by aligning symbols between source code and compiled binaries. Current datasets employ various alignment strategies but struggle with comprehensive expression-level alignment. DIRE and DIRTY [19], [13] track memory addresses for basic variable matching but struggle with complex data structures. VarCorpus [20] modifies DWARF information to preserve names while sacrificing type information and lacks public source code access. ReSym [14] uses stack frame analysis and recursive field resolution but handles only limited struct access patterns and also omits source code metadata. These limitations result in incomplete training data, particularly for complex multi-level dereferences essential for data structure recovery.

[1]Artifact: https://github.com/aur3l14no/R3-Bench

III. AST-Align: Expression Alignment Algorithm

Expression alignment forms the foundation for effectively training and evaluating symbol recovery models. Our approach aims to align all data-related expressions, not merely variable names or specific expression types, thereby providing comprehensive ground truth for symbol recovery techniques.

Previous approaches track variables and dereference expressions through DWARF debug information to resolve base types and field information. However, these methods face several challenges when handling complex expressions:

Multiple indirections: Some fields are accessed via multiple levels of pointer dereferencing. For example, the expression `*(QWORD *)(*v12 + 32LL)` in stripped binary code actually corresponds to the semantic access path `ctxa->pk_info->verify_func` in source code. This creates substantial challenges for resolution-based alignment techniques. The omission of such complex alignments is problematic for two reasons: (1) it significantly reduces available training data, and (2) more critically, certain data structures are exclusively accessed through these multi-level indirections. For instance, in the `mbedtls_pk_info_t` structure depicted in Figure 1, member fields are consistently accessed through double dereference patterns (e.g., `ptr->structure->field`). Without successful recovery of these complex expressions, reconstructing the field composition of structures like `pk_info` becomes infeasible, as no direct field references exist throughout the codebase.

Implementation limitations: Existing approaches typically support a restricted subset of DWARF specifications and primarily target specific architectures (predominantly x86), which significantly constrains their cross-platform applicability. Furthermore, the comprehensive interpretation of various DWARF features presents non-trivial engineering challenges, requiring substantial development effort to implement robust support across different compiler versions, optimization levels, and architectural variants.

The AST-Align method is based on two key observations: (1) Modern decompilers already integrate robust DWARF parsing capabilities, (2) After importing DWARF information, decompiler output typically exhibits predictable, systematic transformations while preserving clear structural correspondence, providing an opportunity to align code before and after importing debug information.

Therefore, rather than developing yet another DWARF parser, AST-Align leverages existing debug information parsing capabilities in decompilers. This design choice may appear to introduce additional dependencies, but our dependency profile is actually identical to approaches like VarCorpus, which similarly aligns two versions of decompiled functions. Even methods that appear to parse DWARF independently, such as ReSym, fundamentally rely on decompiler capabilities for tracking variable locations in stack frames and registers. Moreover, implementing a comprehensive, production-

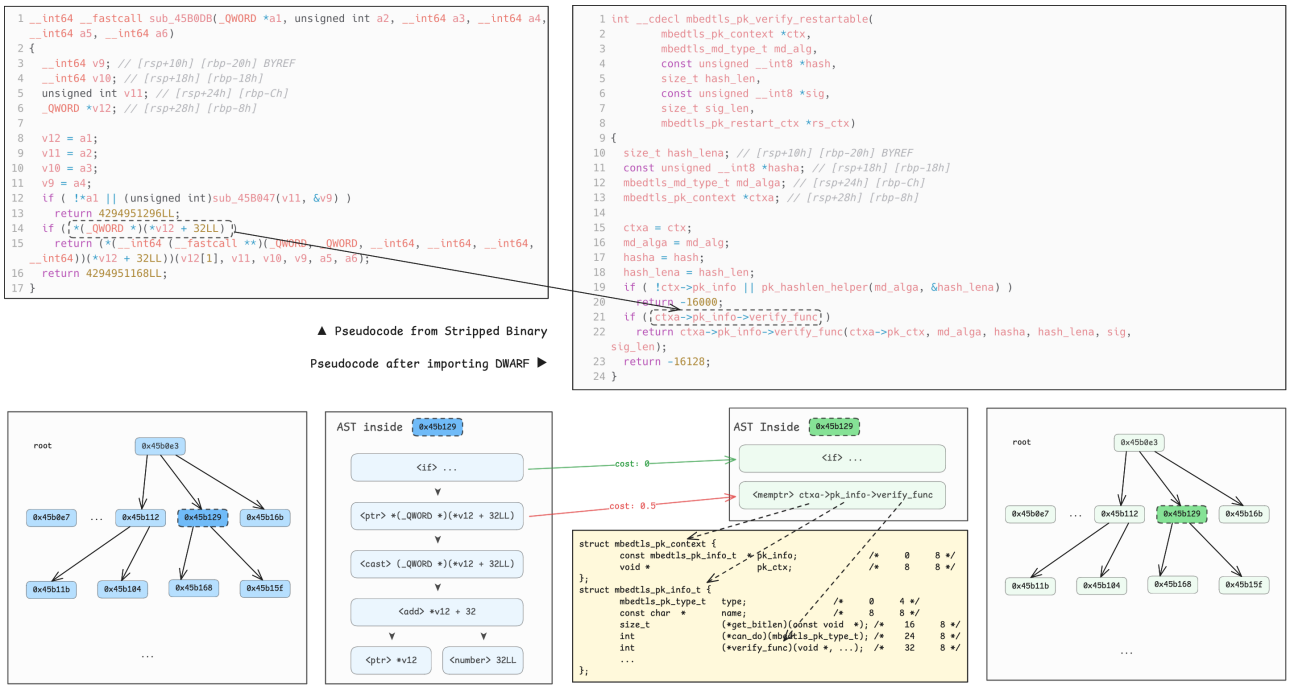

Fig. 1. Overview of the AST-Align algorithm for expression alignment in decompiled code.

quality DWARF parser represents a substantial engineering undertaking across multiple compiler versions and architectures. We align parallel binaries (with and without debug information) by minimizing the tree edit distance between their Abstract Syntax Trees (ASTs). This approach handles complex expressions more effectively and evolves naturally with decompiler ecosystems as they implement support for newer debug formats (e.g., DWARF v5) and architectures. This architectural robustness contrasts favorably with academic tools that implement custom parsers, which often support only limited DWARF versions or specific architectures—for instance, existing academic approaches like ReSym cannot handle DWARF v5 or ARM architectures that are increasingly prevalent in modern software development.

We acknowledge that importing DWARF information does introduce variations in decompiler output, as noted in previous work like VarCorpus. However, unlike VarCorpus, which addresses this challenge by stripping type information from DWARF data, our approach requires preserving complete type information—the decompiler relies on this information to reconstruct meaningful field access patterns that are essential for expression recovery. Our experimental analysis demonstrates that these variations, while present, are manageable within a carefully designed alignment framework. The sophisticated three-phase methodology we develop is specifically intended to handle these controlled variations while maintaining alignment accuracy.

Our alignment process employs a three-phase approach:

Phase 1: Clustered Tree Mapping. We preprocess both ASTs by grouping nodes based on their memory addresses. Nodes with identical addresses are merged into single clusters representing memory regions (e.g., expressions `i`, `j`, and `i + j` might all share address `0xdeadbeef` and will be merged accordingly). The adjacent nodes without effective addresses are also merged in the cluster. We call the resulting structure a clustered tree, where each node represents a subtree. We then apply tree edit distance algorithms to map these clustered trees.

This clustering step effectively utilizes address information from AST nodes. While most alignments occur at identical addresses, we must handle exceptions: (1) some AST nodes lack address information entirely, and (2) node addresses occasionally shift after importing debug information, requiring us to track these changes. (3) Considering the computational complexity of Tree Edit Distance algorithms (typically $O(n^3)$), this address-based node clustering transforms the global tree edit distance calculation into multiple independent cluster-level TED calculations, significantly reducing computational overhead.

Phase 2: Sub Tree Mapping. We calculate the minimum tree edit distance between corresponding subtrees from the previous phase and establish node mappings between the ASTs. This creates alignments between various expressions (from data access expressions, to function calling expressions) in the stripped and debug-informed versions of the decompiled code.

The tree edit distance quantifies the minimal number of edit operations—additions, deletions, and renamings—required to transform one tree structure into another. We formalize our

approach by defining a cost function $\mathcal{C}(op, n_1, n_2)$ for AST node transformations, where $op \in \{add, delete, rename\}$ represents the operation type and n_1, n_2 represent source and target nodes respectively.

Our cost function design is guided by modeling known decompiler behavior patterns. By default, all operations are assigned a unit cost: $\mathcal{C}(op, n_1, n_2) = 1$. However, we introduce specialized cost reductions for transformations that commonly occur when importing debug information:

For data access transformations between semantically equivalent forms, we apply a reduced cost: $\mathcal{C}(rename, n_1, n_2) = 0.5$ when transforming between equivalent access patterns, such as array indexing to field access (e.g., `a1[1]` $\rightarrow$ `s->f`) or pointer arithmetic to field access (e.g., `(a1+8)` $\rightarrow$ `s.f`). These represent identical memory access semantics expressed through different syntactic forms.

For AST nodes representing type conversion operations, we assign minimal costs: $\mathcal{C}(add, null, n_{cast}) = \mathcal{C}(delete, n_{cast}, null) = 0.1$. Type casting operations are frequently inserted or removed during debug information integration to maintain type consistency, but are not useful for label generation, as the types are already reflected on the variables themselves.

These specialized cost assignments enable our algorithm to prioritize structurally meaningful transformations while accommodating common variations in decompiler output.

Phase 3: Expression Filtering. After obtaining mapped expressions, we filter for data access expressions by examining AST node types.

Beyond the data access expressions that constitute our primary focus, AST-Align's generality yields mappings for various program elements including string literals and function identifiers. We implement a dedicated filtering stage to isolate mappings relevant to our expression recovery task. This filtering is critical because indiscriminate inclusion of all mappings would introduce samples requiring cross-function analysis and global program context—dimensions beyond the scope of our function-level analysis framework.

In addition to filtering out non-data access expressions, we eliminate trivial mappings (e.g., `a1` $\rightarrow$ `a1`) that contribute no semantic enrichment, thereby enhancing dataset quality through the removal of uninformative samples.

We implement AST-Align using IDA Pro as the decompilation platform, selected for its superior DWARF processing capabilities and stability. Comparative analysis across multiple decompilers revealed that IDA Pro exhibits the most controlled variations in pseudocode generation when importing debug information. Alternative tools like Ghidra frequently exhibit more significant structural changes, such as if-else statement reordering and control flow reorganization, which substantially complicate the alignment process and reduce the reliability of expression mappings.

This three-phase alignment methodology substantially enhances the precision of expression correspondence compared to conventional direct tree mapping techniques, as validated through systematic human evaluation of the alignment results.

TABLE I
COMPARISON OF DATASET AVAILABILITY AND REPRODUCIBILITY

Dataset	Source	Binary	Variable Label	Expression Label	Reproducible	Arch.	Lang.
DIRT/DIRTY	✗	✓	✓	✗	✗	x86	C/C++
VarCorpus	✗	✗	✓	✗	✗	x86	C/C++
ReSym	✗	✓	✓	✓	✗	x86	C/C++
R3-Bench	✓	✓	✓	✓	✓	x86, ARM	C/C++, Rust

To facilitate this evaluation, we developed a specialized tree mapping visualization tool that enabled direct inspection and verification of the generated alignments. The address-based clustering phase (Phase 1) mitigates alignment errors induced by variations in decompiler output, while the context-sensitive cost function in Phase 2 optimizes structural transformation detection across different representation forms. Additionally, the selective filtering mechanism in Phase 3 ensures that the resulting alignment corpus maintains high information density by excluding trivial or irrelevant mappings.

IV. BUILDING R3-BENCH

A key contribution of R3-Bench is its focus on reproducibility, clarity, and extensibility compared to prior datasets. Table I presents a comparison of the availability of various information across datasets.

Without source code access and version pinning, researchers face significant reproducibility issues, as projects can drastically evolve (e.g., fish-shell rewrote from C to Rust in v4.0). The fragility of build pipelines means researchers cannot generate identical binaries, limiting studies to merely "rebuilding models" rather than extending previous datasets.

Our dataset addresses these challenges through several innovative design decisions:

- **Reproducibility:** We use pinned software repositories and deterministic build systems via the Nix package manager
- **Clarity:** We employ explicit filtering procedures with well-defined inclusion criteria
- **Extensibility:** Our dataset includes comprehensive metadata, offering direct access to source code, project details (e.g., licenses), and precise revision information. This enables researchers to easily extend R3-Bench or retrieve specific project artifacts for their studies.

Instead of relying on GitHub, we leverage Nixpkgs, the package collection for the Nix package manager. Nixpkgs represents the most comprehensive collection of packaged projects according to Repology [21], with explicit build instructions for each package encoded as Nix functions. This approach eliminates the complex problem of determining how to build diverse projects consistently.

A. Dataset Construction Pipeline

Our dataset construction follows a three-step pipeline designed to ensure quality, diversity, and reproducibility:

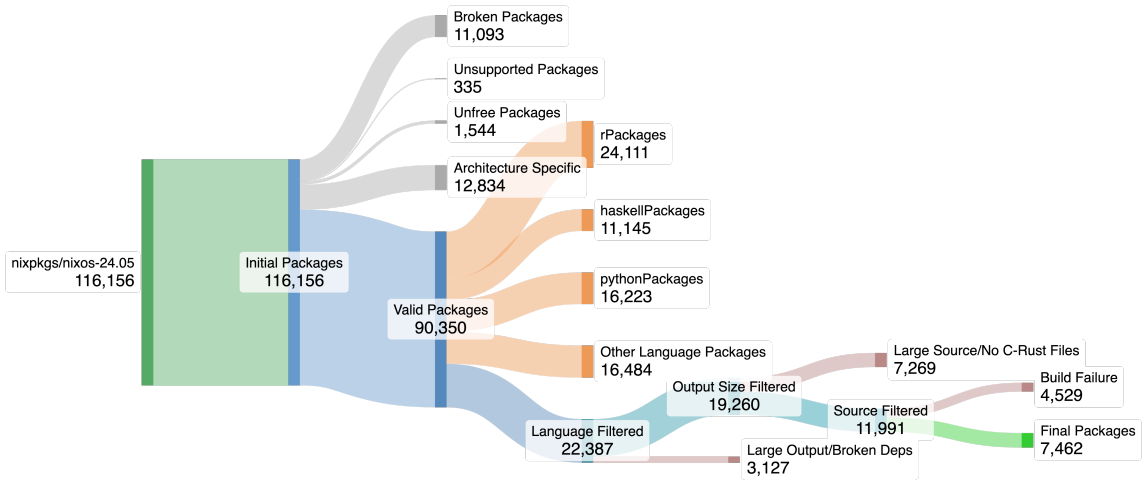

Fig. 2. Corpora generation pipeline. Systematic filtering with clear criteria is adopted to produce the final relevant packages from the original 116,156 packages.

a) Systematic Project Filtering: Starting from 116,156 packages in NixOS/nixpkgs repository (nixos-24.05), we applied systematic filtering: removing broken/unsupported packages, enforcing x86/ARM compatibility (90,350 packages), focusing on C/C++/Rust projects (22,387 packages), imposing 100MB size limits and dependency requirements (19,260 packages), verifying source code presence (11,991 packages), and ensuring meaningful binary differences between O0/O2 optimization levels, yielding 7,462 final projects. The entire procedure took approximately 14 days on a 40-core x86 server and ARM-based cloud services.

b) Building Parallel Corpus: We leverage the Nix package manager to build all filtered projects with debug flags, facilitating high reproducibility. This approach creates deterministic builds where the exact same binary can be reproduced by other researchers using the same Nix expression, eliminating a major challenge in binary analysis research. For each project, we generate both stripped binaries and binaries with DWARF debug information, creating the parallel corpus necessary for ground truth establishment. We include both x86_64 and aarch64 architectures to ensure the dataset's applicability across different platforms.

c) Expression Alignment: We decompile all binaries using IDA Pro, chosen for its established pseudocode generation capabilities that offer consistent representation of control flow and pseudocode styles. We then apply our AST-Align algorithm to identify corresponding expressions between the stripped and debug-informed decompiled code. This alignment creates a comprehensive mapping between raw memory access patterns and their semantically meaningful counterparts, serving as ground truth for our recovery tasks.

To ensure efficient processing and prevent resource exhaustion, we enforce a strict timeout of 600 seconds for each step (build, decompilation, and alignment) per project.

B. Dataset Characteristics

Our dataset demonstrates significant scale and diversity compared to previous datasets. Table II presents a comparison between R3-Bench and ReSym:

TABLE II
COMPARISON BETWEEN R3-BENCH AND RESYM DATASETS

	R3-Bench	ReSym	Multiplier
Projects	7,462	-	3.3×
Aligned Projects	4,785	2,054	2.3×
Binaries in Aligned Projects	24,941	6,721	3.7×
Functions in Aligned Projects	6,447,194	345,412	18.7×
Average Functions per Binary	258.5	51.4	5.0×

As shown in the table, our dataset contains 3.3 times more projects, 3.7 times more binaries, and approximately 18.7 times more functions than the ReSym dataset. This substantial increase provides a more comprehensive test of generalization capabilities, as models must handle a wider variety of coding styles, project domains, and complexity levels.

An additional advantage of our Nixpkgs-based approach is that the dataset naturally evolves as Nixpkgs grows over time. This aspect contributes to maintaining the dataset's relevance, providing a constantly refreshing benchmark for binary analysis techniques. Nixpkgs adds 9,000+ new packages last year, and over 70% of packages are in the latest versions, per Repology [21], reducing contamination risk.

The combination of our novel alignment technique and the scale of the Nixpkgs ecosystem has resulted in a dataset that provides substantial improvements over existing benchmarks. This dataset serves as both a challenging benchmark for existing approaches and a comprehensive dataset for developing symbol recovery models.

V. Evaluation

Our evaluation consists of two complementary analyses designed to assess both the effectiveness of our proposed dataset construction methods and establish performance baselines for future research. First, we evaluate the quality and coverage of our AST-Align algorithm compared to previous alignment techniques, quantifying the impact on downstream recovery tasks. Second, we conduct a comprehensive baseline evaluation, testing various approaches from statistical n-gram models to demonstration-augmented large language models on our R3-Bench dataset.

A. Alignment Algorithm Effectiveness

To evaluate the effectiveness of our AST-Align algorithm, we conducted a series of experiments using ReSym's corpus as a baseline for fair comparison. Our evaluation focuses on two key aspects: (1) the quality and coverage of alignments generated by each algorithm, and (2) the impact of these alignments on downstream recovery tasks.

1) Alignment Quality and Coverage: We first compared the overlap between AST-Align and ReSym by using the quadruple $\langle p, b, f, e \rangle$ (representing project, binary, function and expression respectively) as a key to track overlapping items. Our analysis revealed that approximately 65% of the keys are shared between the two methods, with AST-Align identifying 2.3x more alignments overall.

Among the shared keys, both methods agree on 88% of the alignments. The differences primarily stem from two sources: (1) IDA naming conventions, where occasionally a suffix like `_0` is added to variable names, and (2) representation differences for nested fields, such as `slot->attr.type` (AST-Align) versus `slot->attr` (ReSym), where `type` is the first element of `attr`. These differences reflect implementation choices rather than fundamental disagreements, suggesting high consistency where both methods identify alignments.

To evaluate comprehensiveness, we measured the field information recovered by each alignment method. We selected field information as the evaluation metric because it effectively captures the core functionality of alignment algorithms — exposing structural access patterns within binaries. For methodological consistency, we applied binary normalization to field name occurrences, where each field is counted at most once per binary (value of 1) regardless of its frequency within that binary.

We formalize our evaluation methodology of alignment algorithm using the following notation:

Let $\mathcal{B}$ denote the set of all binaries, $\mathcal{F}$ the set of all fields, and $Gt_{\text{universe}} \subseteq \mathcal{B} \times \mathcal{F}$ represent the universe of (binary, field) pairs where a field appears in a binary. Let $\mathcal{A}_{\text{AST-Align}}$ and $\mathcal{A}_{\text{ReSym}}$ denote the sets of (binary, field) pairs identified by AST-Align and ReSym, respectively.

The recall for an alignment algorithm is:

$$\text{Recall}(\mathcal{A}) = \frac{|Gt_{\text{universe}} \cap \mathcal{A}|}{|Gt_{\text{universe}}|} \tag{1}$$

TABLE III
Field Name Distribution Comparison Between AST-Align and ReSym

Name	Truth (count)	AST-Align (count)	AST-Align Recall (%)	ReSym (count)	ReSym Recall (%)
next	68	64	94.12	47	69.12
name	66	55	83.33	36	54.55
type	57	51	89.47	39	68.42
len	66	44	66.67	35	53.03
size	51	47	92.16	33	64.71
flags	58	39	67.24	32	55.17
data	56	35	62.50	26	46.43
prev	37	33	89.19	23	62.16
length	51	29	56.86	22	43.14
fd	38	29	76.32	10	26.32
Average	–	–	**77.79**	–	**54.30**

This binary normalization approach counts each field exactly once per binary regardless of its frequency, ensuring that alignment algorithms are not repetitively rewarded or penalized for each missed field instance across binaries. Our evaluation framework assesses AST-Align's performance across three dimensions: (1) overall field recall, (2) distributional consistency with ground truth, and (3) per-binary recovery completeness.

It is important to note that the recall metric herein evaluates the alignment algorithm's effectiveness during dataset construction, rather than the recall of the downstream struct recovery task. Nevertheless, alignment algorithms with lower recall inevitably produce insufficient annotated data, which adversely impacts subsequent tasks—a critical relationship we will demonstrate in the struct recovery evaluation (Section V.1.3).

We define our evaluation set $I = (\mathcal{A}_{\text{AST-Align}} \cup \mathcal{A}_{\text{ReSym-Align}}) \cap Gt_{\text{universe}}$ as our subject of investigation. This selection criterion is motivated by the observation that Gt_{universe} contains numerous fields that are not user-defined and are not utilized in binaries. This methodological approach allows us to filter out fields that are challenging to capture and offer limited value for downstream tasks.

Table III presents the recall rates for the ten most frequently occurring field names in I. AST-Align consistently outperforms ReSym across all field names, achieving an average recall rate of 77.79% compared to ReSym's 54.30%.

For distributional analysis, we employed Kendall's τ coefficient to examine the recall patterns of both alignment algorithms. AST-Align demonstrates a τ value of 0.71, whereas ReSym-Align exhibits a τ value of 0.52. This quantitative difference indicates that the field names recovered by AST-Align more accurately reflect the true distribution.

Finally, we conducted a granular analysis of performance on individual programs. Examining a representative binary (provided as sample binary) from the ReSym dataset, we identified 116 unique fields through DWARF debug information, of which AST-Align successfully recovered 56 fields (48.3%), while ReSym recovered only 13 fields (11.2%). This differential performance is visualized in Figure 3 using

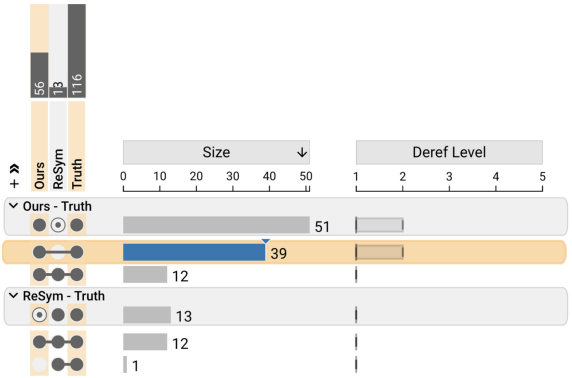

Fig. 3. Upset diagram of aligned fields in sample binary 6df0f62e. Our alignment uncovers 4x more unique fields compared to ReSym.

TABLE IV
LLM-AS-A-JUDGE EVALUATION RESULTS FOR LABEL QUALITY

Metric	AST-Align	ReSym
Recall (%)	75.1 ±30.2	66.1 ±33.7
Precision (%)	91.7 ±18.1	83.6 ±29.9
Confidence (%)	85.5 ±6.6	83.9 ±7.0

an upset diagram. Additionally, we computed the minimum dereference depth (d_{min}) at which each field was detected. For AST-Align, the mean minimum dereference depth was $\bar{d}min = 1.3$, indicating that a substantial proportion of fields are only exposed through multiple levels of indirection. This finding underscores the critical importance of comprehensive alignment algorithms in generating high-quality ground truth for data structure recovery tasks.

2) Label Quality Assessment via LLM-as-a-Judge: While the previous analysis demonstrates AST-Align's superior coverage, we conducted an additional assessment to evaluate the actual quality of generated labels beyond pure quantity metrics. We employed LLM-as-a-Judge methodology for this quality assessment. Our evaluation corpus was constructed from the ReSym test dataset, but due to ReSym's lack of precise revision information—a common limitation across existing datasets that undermines reproducibility—we had to conservatively select only projects whose last git commit occurred at least 5 years ago. This defensive approach was necessary to mitigate potential inconsistencies arising from unknown code evolution in the original ReSym dataset construction. We employed Tree-sitter [22] to parse source code and extract function implementations corresponding to aligned functions.

Our evaluation focused on 696 functions where both AST-Align and ReSym generated alignments. For each function, we presented the LLM judge (Gemini-2.5-Flash) with source code, AST-Align labels, and ReSym labels, evaluating along three dimensions: recall, precision, and confidence.

Table IV presents the evaluation results. AST-Align demonstrates superior performance across all metrics, with notable improvements in both recall (+9.0 percentage points) and precision (+8.1 percentage points). We also ran ranking-based assessments across GPT-4.1, Gemini-2.5-Flash, and Claude-4-Sonnet; AST-Align had equal-or-higher recall 79% and precision 94%, with inter-model agreement 0.857 (recall) and 0.587 (precision).

To understand the source of these quality improvements, we conducted detailed analysis of cases where the two methods exhibited significant disagreement (differences exceeding 0.5 in recall or precision scores). Our analysis reveals two main categories where AST-Align significantly outperforms ReSym: complex multi-level pointer dereferences such as `**(_QWORD **)(a1 + 1328)` corresponding to source expressions like `s1->dynsym->data_offset`, and expressions involving large offsets that cause ReSym's tracking mechanisms to lose correspondence with original field definitions during DWARF parsing.

A representative example from the `nxp-archive/openil_linuxptp` project illustrates these underlying patterns. The source code ground truth shows a function `clock_get_tstamp` that accesses struct fields `clock->is_ts_available` and `clock->last_ts`. AST-Align correctly identified all field access expressions with accurate field names and types, while ReSym identified only one expression with incorrect field attribution.

The superior precision of AST-Align can be attributed to its context-aware validation mechanism. During tree alignment, AST-Align evaluates alignment quality holistically by considering overall tree edit distance. Incorrect alignments typically result in higher edit distances, encouraging contextually consistent alignments and filtering out spurious matches that might arise from local pattern matching.

3) Impact on Struct Recovery Performance: The effectiveness of our alignment methodology must ultimately be validated through its impact on downstream tasks. For struct recovery evaluation, we use ReSym's precision and recall metrics where Precision $= \frac{\sum_{u \in U} |Gt^u_{Attr} \cap Pred^u_{Attr}|}{\sum_{u \in U} |Pred^u_{Attr}|}$ measures field attribute recovery quality, with U representing variables pointing to structures and Gt^u_{Attr} denoting ground truth field attributes.

We evaluate this impact through three critical dimensions:

First, we investigate whether models can effectively learn expression recovery patterns despite the increased volume and diversity of aligned data. Second, we assess whether models trained on our enriched dataset demonstrate improved performance in both expression recovery and subsequent struct recovery tasks. Finally, we examine the upper bound of struct recovery performance achievable with perfect expression recovery.

We utilized the fine-tuning pipeline established by ReSym for building our model ExprDecoder.

For our implementation, we utilized the fine-tuning pipeline established by ReSym to develop ExprDecoder. We selected pre-trained StarCoder 3B as the base model, maintaining complete consistency with ReSym's experimental setup. To construct our training dataset, we specifically collected sam-

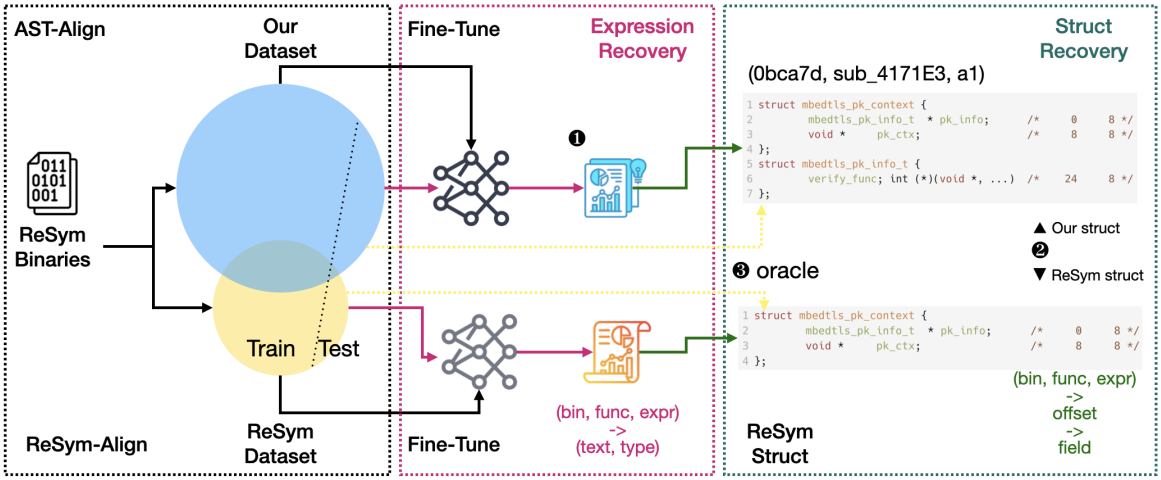

Fig. 4. Evaluation of AST-Align performance compared to existing approaches. The figure illustrates three experimental configurations: (1) expression recovery performance evaluation of our expression recovery models trained on AST-Align data, (2) struct recovery performance using models trained on different alignment datasets, and (3) theoretical upper bound analysis showing maximum achievable struct recovery performance with perfect expression recovery ("oracle" setting). Results demonstrate that models trained with AST-Align's more comprehensive alignments achieve significant improvements in struct recovery tasks compared to previous alignment methods.

ples containing pointer dereference operators (->) to focus on data structure recovery, discarding pure variable samples to concentrate on field access expressions.

Following ReSym's training methodology, we applied identical filtering criteria: inputs exceeding 4096 tokens were discarded to avoid truncating field access expressions that could lead to model misinterpretation. We maintained hardware parity using 8x A100 GPUs, training for 3 epochs with a total training time of 30 hours. The model weights were updated using an AdamW optimizer with identical hyperparameters: batch size of 16, learning rate of 5e-5, adjusted by 500 warm-up steps followed by cosine decay.

Our approach differs from ReSym's in the model's input-output design. While ReSym's model is structured to recover multiple attributes (base name, struct type, field name, and field type) for expressions like a1, our ExprDecoder directly recovers the complete target expression. This approach accommodates more complex cases including multi-level dereferences where assigning a single struct label is often infeasible.

To evaluate our model's learning effectiveness, we assess performance across five distinct evaluation metrics consistent with standard benchmarking approaches in expression recovery: text accuracy, type accuracy, base name accuracy, field name accuracy, and field type accuracy. Each accuracy metric is computed using exact match criteria, where a prediction is considered correct if and only if it precisely matches the ground truth.

For the latter three structural metrics (base name accuracy, field name accuracy, and field type accuracy), we constructed a specialized evaluation subset containing only expressions conforming to the simplest pattern $\alpha \rightarrow \beta$, where α represents the base variable and β represents the accessed field. These expressions represent the most straightforward structural relationships and provide unambiguous evaluation criteria. This evaluation framework enables us to systematically assess the model's capacity to learn various aspects of expression recovery despite the heightened complexity and heterogeneity present in our expanded dataset.

TABLE V
PERFORMANCE OF EXPRDECODER ON EXPRESSION RECOVERY TASKS

	Text	Type	Base name	Field name	Field type
Accuracy (%)	53.9	62.8	57.4	57.7	62.8

For the latter two investigations about struct recovery, we employed Large Language Model (LLM) as a substitute for the reasoning system in ReSym, since ReSym's struct recovery reasoning system which aggregates expression information across functions is not open-sourced. This substitution is justified on two grounds: (1) empirical testing confirmed that LLMs can effectively leverage expression recovery information, achieving accuracy comparable to or slightly exceeding ReSym's approach; (2) our dataset contains substantially more complex expressions beyond simple base+offset patterns, necessitating a more sophisticated struct recovery system than the one described in ReSym's methodology—an extension beyond our current scope of investigation.

In our implementation, we selected Gemini-2.5-flash as our LLM platform for two primary reasons: its large context window enables processing of complex binaries, and its relatively low API cost permits evaluation across numerous binaries (over 100) within reasonable costs. To mitigate the inherent variability in LLM performance, we conducted three separate experimental trials and reported averaged results, ensuring more reliable performance metrics.

TABLE VI
COMPARATIVE ANALYSIS OF MODEL PERFORMANCE FOR STRUCT RECOVERY

Precision (%)				
Attribute	**Ours**	**ReSym**	**Ours (Oracle)**	**ReSym (Oracle)**
Size	52.0	37.0	62.6	63.9
Name	39.7	16.8	58.9	51.8
Type	37.4	21.2	54.8	56.2
Recall (%)				
Attribute	**Ours**	**ReSym**	**Ours (Oracle)**	**ReSym (Oracle)**
Size	49.7	27.9	69.6	42.9
Name	37.8	14.9	68.3	37.9
Type	37.3	18.1	66.8	39.8
F1 Score (%)				
Attribute	**Ours**	**ReSym**	**Ours (Oracle)**	**ReSym (Oracle)**
Size	50.5	31.8	65.9	51.4
Name	34.8	15.8	63.3	43.8
Type	34.1	19.5	60.2	46.6

The initial experimental results as is shown in Table VI indicate that our ExprDecoder+LLM approach, trained on the enhanced dataset, outperforms ReSym's FieldDecoder+LLM configuration across both precision and recall metrics. To isolate the causal factors behind this performance differential, we conducted an oracle experiment wherein ground truth expression labels were provided as input to the struct recovery system.

The results demonstrate a fundamental limitation in ReSym's approach: even with perfect expression recovery (oracle configuration), the ReSym system achieves suboptimal recall rates ($< 50\%$) for structural attributes. This performance ceiling is attributable to insufficient field alignment in the ReSym dataset, where a significant proportion of structure fields remain unaligned, imposing an inherent upper bound on recovery performance regardless of model quality.

B. Baseline Performance

For establishing baseline performance on R3-Bench, we employed two representative methodological approaches: statistical n-gram models [23] and demonstration-augmented large language models. We partitioned the R3-Bench dataset using a 95:5 train-test split following ReSym's practice and constructed R3-Bench-Lite (100 projects, 5,000 functions, with similar quantity as the ReSym's test set) for cost-effective LLM evaluation.

In our baseline evaluation, we focus exclusively on expression recovery accuracy rather than struct recovery for several reasons: (1) expression recovery has greater practical utility as traditional tools already implement type propagation, (2) struct recovery requires prohibitively expensive binary-wide analysis, (3) expression recovery employs simpler, more stable evaluation metrics, and (4) our experiments suggest that expression recovery performance directly influences struct recovery outcomes.

TABLE VII
BASELINE PERFORMANCE COMPARISON ACROSS METHODS AND DATASETS

Method	**R3-Bench-Lite**	**ReSym Test**
LLM (with demo, best model)	36%	55%
Fine-tuned Models	-	56%
N-gram Models	40%	55%

To establish validity, we conducted parallel assessments on both R3-Bench and ReSym datasets, converting ReSym's native labels to our expression recovery formulation for consistency.

1) Statistical and Neural Approaches: We selected STRIDE [23] as a representative statistical approach based on n-gram modeling. While STRIDE was originally designed for variable recovery, which does not require consideration of the pseudocode itself (as variables are generically labeled as `v1`, `a2`, etc.), our task encompasses both variables and expressions. Consequently, we extended STRIDE to incorporate the expressions themselves into the n-gram calculation process.

The values in Table VII represent the average of expression text accuracy and expression type accuracy for each method and dataset. Empirical analysis reveals that n-gram performance is unexpectedly effective for elementary recovery tasks. On the ReSym dataset, n-gram models attained 55% accuracy, comparable to ReSym's fine-tuned model at 56%. However, when evaluated on our more comprehensive R3-Bench dataset, n-gram performance decreased significantly to 40%, which nevertheless surpassed the performance of more computationally intensive LLM-based methods. This outcome suggests that at the expression recovery level, even with the augmentation of demonstration, LLM-based approaches may not provide substantial advantages over simpler statistical methods.

2) Large Language Models: We also evaluated the performance of large language models (LLMs) on our dataset. While general LLMs alone struggled with expression recovery tasks, their performance improved significantly when provided with demonstrations of similar examples.

For model selection, we employed a dual-criteria approach that balanced performance capabilities with computational efficiency constraints. Our primary selection criteria were: (1) demonstrated performance on established programming benchmarks, and (2) inference cost considerations for large-scale evaluations. For flagship models with higher computational requirements, we selected Claude-3.7-Sonnet [24], which demonstrated superior performance on code-related tasks at the time of experimentation. For more economical alternatives, we evaluated models from the Qwen family [25], DeepSeek-Reasoner [26], and Phi-4 [27], which have shown competitive performance on programming tasks while maintaining lower inference costs.

For our retrieval-augmented approach, we employed CodeSage [28] as the embedding model and use the Hierarchical Navigable Small World (HNSW) algorithm [29] implemented in PGVector [30] to construct an efficient index

TABLE VIII
LLM PERFORMANCE ON R3-BENCH-LITE

Model	Text Accuracy	Type Accuracy
Claude-3.7-Sonnet	35.0%	37.5%
Qwen2.5-32B-Instruct	30.5%	38.7%
Qwen2.5-Coder-32B-Instruct	29.2%	31.0%
DeepSeek-Reasoner	27.6%	25.1%
Phi-4	18.4%	24.9%
Qwen2.5-7B-Instruct	14.9%	19.5%
Qwen2.5-Coder-7B-Instruct	13.6%	18.7%

TABLE IX
LLM PERFORMANCE ON RESYM

Model	Text Accuracy	Type Accuracy
Claude-3.7-Sonnet	51.6%	59.5%
Qwen2.5-32B-Instruct	50.3%	57.4%
DeepSeek-Reasoner	45.7%	53.3%
Qwen2.5-Coder-32B-Instruct	43.4%	53.0%
Phi-4	36.3%	46.8%
Qwen2.5-Coder-7B-Instruct	31.4%	42.5%
Qwen2.5-7B-Instruct	34.4%	35.3%

for the training sets of both ReSym and R3-Bench. This methodology facilitated the retrieval of semantically similar training examples for each test sample, enabling the LLMs to better understand the task context and significantly enhance prediction quality.

Our empirical analysis demonstrates a clear correlation between model scale and performance efficacy across both datasets. Claude-3.7-Sonnet achieved the highest text accuracy on both evaluation corpora (35.0% on R3-Bench-Lite and 51.6% on ReSym), while Qwen2.5-32B-Instruct exhibited superior type accuracy on R3-Bench-Lite (38.7%) and Claude-3.7-Sonnet maintained dominance on ReSym (59.5%).

A notable observation emerges from the inter-dataset performance comparison: all models consistently demonstrated higher accuracy metrics on the ReSym dataset relative to R3-Bench-Lite. This performance differential suggests that R3-Bench-Lite encompasses more challenging and diverse expression patterns that necessitate more sophisticated reasoning capabilities. Additionally, cross-dataset evaluation of ReSym's pre-trained model on R3-Bench yielded substantially lower performance (only 8%), further confirming the increased complexity of our benchmark.

Our findings reveal three key insights:

1) AST-Align's comprehensive expression alignment significantly outperforms existing methods, and enabling more effective downstream recovery tasks.
2) Statistical n-gram models demonstrate unexpectedly strong performance in expression recovery, achieving comparable results to fine-tuned neural models and often outperforming large language models, suggesting that simpler approaches remain competitive for this task.
3) The performance gap between ReSym and R3-Bench datasets reveals that existing benchmarks may not capture the full complexity of real-world expression patterns, highlighting the need for more comprehensive evaluation frameworks.

VI. DISCUSSION

Our research reveals several important insights about expression alignment and symbol recovery that warrant further discussion.

A. Expression Recovery as a Primary Metric

Our analysis demonstrates that certain data structures, such as mbedtls_pk_info_t in Figure 1, never appear as independent variables but are exclusively accessed through multi-level indirections. Current evaluation frameworks that examine each pointer type individually systematically exclude these structures from evaluation corpus, despite their prevalence in operational code. This suggests that expression recovery should be considered a fundamental metric with independent evaluative significance, rather than merely a subordinate component of data structure recovery.

Current data structure evaluation metrics face a dilemma: should structures referenced by multiple pointers receive proportionally higher weight in evaluation? Edit distance metrics operating solely on structure definitions fail to capture whether pointer-structure relationships are accurately recovered, while combined approaches introduce complex balancing challenges without theoretical justification. Given these methodological constraints, expression recovery performance may provide a more theoretically sound terminal evaluation metric.

B. Limitations and Future Work

While R3-Bench represents a significant advancement, several limitations remain. First, the dataset primarily covers open-source software, which may not fully represent proprietary software patterns. Second, our alignment algorithm, while more comprehensive than previous approaches, still misses approximately 52% of available struct field information, indicating room for improvement. Third, the substantial scale of R3-Bench presents computational challenges for comprehensive model training. We did not conduct full-scale fine-tuning experiments on the complete R3-Bench dataset due to the prohibitive computational resource requirements. With over 10 million functions, training state-of-the-art models on the full dataset would require significant computational infrastructure beyond our current capacity. This limitation necessitated our focus on lighter methods for baseline evaluations, though the complete dataset remains available for various learning methods.

VII. CONCLUSION

This paper presents R3-Bench, a comprehensive dataset for variable and data structure recovery that addresses key limitations in existing datasets. By introducing AST-Align, a novel expression alignment algorithm that unifies variable and field access expression recovery, we enable more comprehensive ground truth generation for training and evaluating symbol recovery techniques.

Our evaluation demonstrates that R3-Bench significantly advances the state of the art in several ways: First, by capturing more struct field information than previous approaches, R3-Bench enables more comprehensive training data and more accurate evaluation. Second, the scale of R3-Bench (18.7 times more functions than previous datasets) provides a more robust benchmark. Third, the reproducible construction methodology ensures that researchers can extend and build upon our work.

Our baseline experiments also establish a comprehensive foundation for future research by providing reference performance across diverse approaches from statistical n-gram models to large language models.

REFERENCES

[1] J. Vadayath, M. Eckert, K. Zeng, N. Weideman, G. P. Menon, Y. Fratantonio, D. Balzarotti, A. Doupé, T. Bao, R. Wang *et al.*, "Arbiter: Bridging the static and dynamic divide in vulnerability discovery on binary programs," in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 413–430.

[2] J. Gao, X. Yang, Y. Fu, Y. Jiang, and J. Sun, "VulSeeker: A semantic learning based vulnerability seeker for cross-platform binary," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ser. ASE '18. New York, NY, USA: Association for Computing Machinery, Sep. 2018, pp. 896–899.

[3] O. Mirzaei, R. Vasilenko, E. Kirda, L. Lu, and A. Kharraz, "SCRUTINIZER: Detecting code reuse in malware via decompilation and machine learning," in *Detection of Intrusions and Malware, and Vulnerability Assessment: 18th International Conference, DIMVA 2021, Virtual Event, July 14–16, 2021, Proceedings*. Berlin, Heidelberg: Springer-Verlag, Jul. 2021, pp. 130–150.

[4] I. Angelakopoulos, G. Stringhini, and M. Egele, "{FirmSolo}: Enabling dynamic analysis of binary linux-based {IoT} kernel modules," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 5021–5038.

[5] D. Kim, E. Kim, S. K. Cha, S. Son, and Y. Kim, "Revisiting binary code similarity analysis using interpretable feature engineering and lessons learned," *IEEE Trans. Softw. Eng.*, vol. 49, no. 4, pp. 1661–1682, Apr. 2023.

[6] K. Pei, Z. Xuan, J. Yang, S. Jana, and B. Ray, "Learning approximate execution semantics from traces for binary function similarity," *IEEE Transactions on Software Engineering*, vol. 49, no. 4, pp. 2776–2790, Apr. 2023.

[7] J. Wang, M. Sharp, C. Wu, Q. Zeng, and L. Luo, "Can a deep learning model for one architecture Be used for others? {retargeted-Architecture} binary code analysis," in *32nd USENIX Security Symposium (USENIX Security 23)*, 2023, pp. 7339–7356.

[8] S. Chen, Z. Lin, and Y. Zhang, "{SelectiveTaint}: Efficient data flow tracking with static binary rewriting," in *30th USENIX Security Symposium (USENIX Security 21)*, 2021, pp. 1665–1682.

[9] "IDA pro," https://hex-rays.com/ida-pro, 2025.

[10] "Ghidra," https://ghidra-sre.org/, 2025.

[11] A. Slowinska, T. Stancescu, and H. Bos, "Howard: A dynamic excavator for reverse engineering data structures."

[12] E. J. Schwartz, C. F. Cohen, M. Duggan, J. Gennari, J. S. Havrilla, and C. Hines, "Using Logic Programming to Recover C++ Classes and Methods from Compiled Executables," in *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, ser. CCS '18. New York, NY, USA: Association for Computing Machinery, Oct. 2018, pp. 426–441.

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

[14] D. Xie, Z. Zhang, N. Jiang, X. Xu, L. Tan, and X. Zhang, "ReSym: Harnessing LLMs to Recover Variable and Data Structure Symbols from Stripped Binaries," 2024.

[15] Z. Sha, H. Shu, H. Wang, Z. Gao, Y. Lan, and C. Zhang, "HyRES: Recovering data structures in binaries via semantic enhanced hybrid reasoning," *ACM Trans. Softw. Eng. Methodol.*, May 2025.

[16] X. Xu, Z. Zhang, Z. Su, Z. Huang, S. Feng, Y. Ye, N. Jiang, D. Xie, S. Cheng, L. Tan *et al.*, "Unleashing the power of generative model in recovering variable names from stripped binary," in *Proceedings 2025 Network and Distributed System Security Symposium*. San Diego, CA, USA: Internet Society, 2025.

[17] Z. Gao, H. Wang, Y. Wang, and C. Zhang, "Virtual compiler is all you need for assembly code search," in *Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, L.-W. Ku, A. Martins, and V. Srikumar, Eds. Bangkok, Thailand: Association for Computational Linguistics, Aug. 2024, pp. 3040–3051.

[18] E. Dolstra, "The purely functional software deployment model," Ph.D. dissertation, Utrecht University, Utrecht, Netherlands, 2006.

[19] J. Lacomis, P. Yin, E. J. Schwartz, M. Allamanis, C. L. Goues, G. Neubig, and B. Vasilescu, "DIRE: A neural approach to decompiled identifier naming," in *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*, ser. ASE '19. San Diego, California: IEEE Press, Feb. 2020, pp. 628–639.

[20] K. K. Pal, A. P. Bajaj, P. Banerjee, A. Dutcher, M. Nakamura, Z. L. Basque, H. Gupta, S. A. Sawant, U. Anantheswaran, Y. Shoshitaishvili *et al.*, ""len or index or count, anything but v1": Predicting variable names in decompilation output with transfer learning," in *2024 IEEE Symposium on Security and Privacy (SP)*. San Francisco, CA, USA: IEEE, May 2024, pp. 4069–4087.

[21] "Repology," https://repology.org/, 2025.

[22] "Tree-sitter/tree-sitter," tree-sitter, May 2025.

[23] H. Green, E. J. Schwartz, C. L. Goues, and B. Vasilescu, "STRIDE: Simple Type Recognition In Decompiled Executables," Jul. 2024.

[24] "Claude 3.7 sonnet," https://www.anthropic.com/claude/sonnet.

[25] Qwen, A. Yang, B. Yang, B. Zhang, B. Hui, B. Zheng, B. Yu, C. Li, D. Liu, F. Huang *et al.*, "Qwen2.5 technical report," Jan. 2025.

[26] DeepSeek-AI, D. Guo, D. Yang, H. Zhang, J. Song, R. Zhang, R. Xu, Q. Zhu, S. Ma, P. Wang *et al.*, "DeepSeek-R1: Incentivizing reasoning capability in LLMs via reinforcement learning," Jan. 2025.

[27] M. Abdin, J. Aneja, H. Behl, S. Bubeck, R. Eldan, S. Gunasekar, M. Harrison, R. J. Hewett, M. Javaheripi, P. Kauffmann *et al.*, "Phi-4 technical report," Dec. 2024.

[28] D. Zhang, W. U. Ahmad, M. Tan, H. Ding, R. Nallapati, D. Roth, X. Ma, and B. Xiang, "Code representation learning at scale," in *The Twelfth International Conference on Learning Representations*, Oct. 2023.

[29] Y. A. Malkov and D. A. Yashunin, "Efficient and robust approximate nearest neighbor search using hierarchical navigable small world graphs," *IEEE Trans. Pattern Anal. Mach. Intell.*, vol. 42, no. 4, pp. 824–836, Apr. 2020.

[30] "Pgvector/pgvector," pgvector, Apr. 2025.