ReSym: Harnessing LLMs to Recover Variable and Data Structure Symbols from Stripped Binaries

Danning Xie
Purdue University
West Lafayette, IN, USA
xie342@purdue.edu

Zhuo Zhang
Purdue University
West Lafayette, IN, USA
zhan3299@purdue.edu

Nan Jiang
Purdue University
West Lafayette, IN, USA
jiang719@purdue.edu

Xiangzhe Xu
Purdue University
West Lafayette, IN, USA
xu1415@purdue.edu

Lin Tan
Purdue University
West Lafayette, IN, USA
lintan@purdue.edu

Xiangyu Zhang
Purdue University
West Lafayette, IN, USA
xyzhang@cs.purdue.edu

ABSTRACT

Decompilation aims to recover a binary executable to the source code form and hence has a wide range of applications in cyber security, such as malware analysis and legacy code hardening. A prominent challenge is to recover variable symbols, including both primitive and complex types such as user-defined data structures, along with their symbol information such as names and types. Existing efforts focus on solving parts of the problem, e.g., recovering only types (without names) or only local variables (without user-defined structures). In this paper, we propose ReSym, a novel hybrid technique that combines Large Language Models (LLMs) and program analysis to recover both names and types for local variables and user-defined data structures. Our method encompasses fine-tuning two LLMs to handle local variables and structures, respectively. To overcome the inherent token limitations in current LLMs, we devise a novel Prolog-based algorithm to aggregate and cross-check results from multiple LLM queries, suppressing uncertainty and hallucinations. Our experiments show that ReSym is effective in recovering variable information and user-defined data structures, substantially outperforming the state-of-the-art methods.

CCS CONCEPTS

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

KEYWORDS

Reverse Engineering, Large Language Models, Program Analysis

ACM Reference Format:
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 ACM SIGSAC Conference on Computer and Communications Security (CCS '24), October 14–18, 2024, Salt Lake City, UT, USA.* ACM, New York, NY, USA, 15 pages. https://doi.org/10.1145/3658644.3670340

CCS '24, October 14–18, 2024, Salt Lake City, UT, USA

ACM ISBN 979-8-4007-0636-3/24/10
https://doi.org/10.1145/3658644.3670340

1 INTRODUCTION

Decompilation aims to recover the source code form of a binary executable, especially stripped binaries. This entails recovering variables, including both primitive and complex variables such as data structures and arrays, which are in the form of register and memory accesses after compilation. Additionally, symbol information for these variables, which is completely discarded in stripped binaries, must also be restored. Decompilation plays a pivotal role in many security applications, such as malware analysis [22, 24, 85] and vulnerability detection [30, 55, 71].

Existing decompilers such as IDA [32] and Ghidra [47] are able to recover the source code control structure with good precision, and in some cases, a subset of local variables. However, they usually cannot recover meaningful names for variables and simply name them as v1, v2, etc. The recovered types are often incorrect, and custom data structures are out of their scope too. There has been significant research [11, 38, 39, 49, 50, 88] focused on recovering the missing information, e.g., variable names and types, from decompiled code to enhance its readability. While these efforts have demonstrated substantial progress, they have limitations. For example, OSPREY [88] utilizes probabilistic inference based on program properties such as memory access patterns and data-flow to recover variables and data structures. However, it falls short of recovering high-level, human-readable symbol information, such as names and type names. DIRTY [11] employs machine learning methods to recover variable names and types, but it has limitations in recovering user-defined data structures. Recovering descriptive and insightful variable names, types, and user-defined data structures is crucial for comprehending the code, yet it remains a challenging and unfulfilled task due to its inherent complexities. More discussion of related work is in Section 6.1.

Large Language Models (LLMs) have exhibited inspiring capabilities in understanding code, rivaling human experts. Extensive research has demonstrated the superior performance of LLMs over traditional methods in tasks that require code understanding, such as unit test generation [9, 17, 35], program repair [34, 80], and code completion [40, 70]. Considering LLMs' powerful ability to comprehend and generate code, applying them to symbol recovery appears promising. In particular, similar to DIRTY, we aim to build on existing decompilers' outputs and recover variables and symbols from decompiled code by those tools, instead of recovering directly from

stripped binaries. However, our exploration revealed challenges in effectively utilizing LLMs for this task.

First, directly prompting a pre-trained LLM for general-purpose to produce symbols from decompiled code hardly works due to the unique patterns in the decompiled code and the inherent uncertainty in symbol recovery, meaning that the source code snippets with various symbols and high-level structures may be compiled into highly similar binary code patterns. Second, the accurate recovery of variable names and types requires a comprehensive, global view of the program, but the intrinsic token limitations of LLMs, e.g., 4,096 tokens, make providing the full context for thorough analysis impossible.

Inspired by the reverse engineering processes employed by human engineers [46, 73], we introduce ReSym, a novel automatic reverse engineering technique that mimics the manual approach by combining fine-tuned LLMs, which provide initial results, with a lightweight Prolog-based reasoning system [14], designed to aggregate results from various sources. To adapt pre-trained LLMs for the particular symbol recovery task, we divide the problem into two manageable sub-problems: 1) recovering local variables, and 2) recovering (custom) structures and their accesses. This resembles how human engineers divide the problem. We fine-tune two LLMs for the sub-problems, respectively. To create the training dataset, we first compile source files with debug symbols and make a copy of these symbols before stripping them from the compiled binaries. Next, we decompile these fully stripped binaries to obtain the decompiled code. The decompiled code is often quite low-level, in which a source variable or structure can be broken down into multiple decompiled variables. We design novel algorithms that can accurately project a variable or expression in the decompiled code to a source variable or structure field. During inference, the individual functions of a program are passed to the two fine-tuned LLMs to generate symbols for local variables and structures, respectively. To suppress the inherent uncertainty and hallucinations in LLMs, these per-function results are rigorously aggregated and cross-checked by a Prolog-based reasoning algorithm.

Our contributions are as follows:

- We divide the difficult symbol recovery problem into two manageable sub-problems with different natures and fine-tune two LLMs to address them with superior performance.
- We develop novel algorithms that generate precisely labeled training datasets for model fine-tuning. It features the ability to handle complex variables, e.g., arrays and structures, which are usually decomposed to singleton variables or complex expressions in the decompiled code, and it is difficult to associate them with their original symbols.
- We develop a rigorous Prolog-based reasoning algorithm to aggregate and cross-check local results, together with a similarity-biased voting system to resolve prediction conflicts, enabling comprehensive data structure recovery.
- We build a large-scale public dataset of C/C++ code with 7,416 projects, 113,696 binaries, and their decompiled versions. All the variables, structures, and field accesses in the decompiled code have the corresponding symbols annotated. The dataset can facilitate future research in binary name and type recovery.

```
1  void ixp_pstrings
2      (IxpMsg *msg, ushort num){
3    ushort len;
4    uint size;
5    uchar *s;
6    if(msg->mode == 1){
7     s = msg->pos;
8     size = len * num;
9     emalloc(size);
10   }
11 }
```
Source Code

```
1  unsigned int64 sub_404056
2        (int64 a1, int16 a2){
3    unsigned int16 v3;
4    unsigned int v4;
5    void *dest;
6    if (*(int *)(a1 + 28) == 1){
7     dest = *(void **)(a1 + 8);
8     v4 = v3 * a2;
9     sub_406BB9(v4);
10   }
11 }
```
Decompiled Code

Figure 1: A function's source code and decompiled code

- We develop a prototype ReSym (*harnessing LLMs to REcover variable and data structure SYMbols from stripped binaries*). Our evaluation on a real-world dataset demonstrates ReSym's proficiency in recovering variable names and types, achieving overall accuracies of 56.4% and 64.6%, respectively. It can successfully reconstruct the complete layout of user-defined data structures with a precision of 72.9%. Notably, ReSym is adept at identifying inlined structures or arrays that the decompiler has fragmented into multiple variables with an F1 of 85.9%. Experiments show that ReSym outperforms the state-of-the-art in name and type recovery (by 1.5–16.5%) and structure recovery. Additionally, ReSym's application to a real-world malware case shows its ability to address security challenges.

Availability. Our dataset and artifacts are available to facilitate future research and reproducibility [1].

2 MOTIVATION

Fig. 1 depicts a function from a real-world project, with its source and decompiled code. The code is simplified for illustrative purposes. The function takes two parameters: a pointer `msg` to a user-defined structure `IxpMsg` and an unsigned short `num`. Lines 6 to 8 access two fields, `mode` and `pos`, of `msg`, and update two local variables, `size` and `s`. While the decompiled code offers more clarity than the binary representations, it still lacks critical information, such as variable types and names, rendering it difficult to understand. For example, most variables have non-descriptive names compared to the source code, such as `a1` and `v4`, and some have incorrect types, such as `uchar *s` being erroneously typed as `void *`. Additionally, due to the absence of information about user-defined structures, the decompiled code reduces field accesses to low-level memory operations. Consider the expression `(int *)(a1+28)` (highlighted in blue), which accesses bytes 28 to 31 of the structure referenced by `a1`, corresponding to `msg->mode` in the source code. This makes the decompiled code challenging and potentially misleading to interpret.

Therefore, our goal is to recover variable names and types from binary executables, improving the readability of decompiled code. Specifically, this involves not only identifying the types and names of the involved variables (e.g., `uchar *s`) but also recovering the structure declaration of the involved user-defined non-standard structures (e.g., `IxpMsg`) and their field accesses (e.g., `msg->mode`).

2.1 Limitations of Existing Techniques

Given the significance of recovering variable names and types from binary executables, over the past decade, numerous techniques have been developed for this purpose [11, 38, 39, 49, 50, 88]. Despite these

```
struct IxpMsg {
 char*    data;
 char*    pos;
 char*    end;
 _ixpuint size;
 _ixpuint mode;
};
```
Ground Truth

```
struct sha256_ctx {
 uint32_t H[8];
 uint32_t total[2];
 uint32_t buflen;
 char buffer[128];
};
```
DIRTY

```
struct struct0{
 int8*  s_0,
 int8*  s_1,
 int8*  s_2,
 int64  s_3,
 int64  s_4
};
```
OSPREY

```
struct Buffer {
 uint8_t* buffer;
 uint8_t* pos;
 uint8_t* streamPos;
 uint32_t bufferSize;
 uint32_t type;
};
```
RESYM

Figure 2: Comparison of user-defined data structure recovery

efforts, there is still plenty of room to improve due to the intrinsic difficulty of the task. Fig.2 compares the performance of two state-of-the-art methods, DIRTY [11] and OSPREY [88], specifically in reconstructing the structure declaration of `IxpMsg` from the earlier example. DIRTY [11] utilizes a transformer-based multi-classification model to predict variable names and types. However, its effectiveness is limited to the types included in its training dataset. Consequently, DIRTY struggles with user-defined structures not seen during training, e.g., `struct IxpMsg`, leading to inaccurate results and incorrect layout predictions. It is worth noting that it is impractical for DIRTY to encompass all possible user-defined structures in its training set, particularly for specialized security tasks like malware analysis. On the other hand, OSPREY [88] successfully reconstructs the structure layout and accurately determines field sizes and pointers. However, it cannot recover high-level, human-readable symbol information (e.g., names and type names), making its results difficult to interpret. Additionally, OSPREY relies on a heavy-weight data-flow analysis [89], posing scalability challenges with complex binaries.

2.2 Opportunities and Challenges

With the emergence of LLMs and their demonstrated powerful ability to comprehend and generate code, applying them to decompiled code recovery appears promising. However, our initial efforts at using LLMs for this sophisticated task have shown limited success. Merely prompting LLMs to enhance the readability of decompiled code results in poor performance. Fig. 3 shows the output of GPT-4 [53] when asked to make the decompiled code more readable, i.e., with zero-shot learning. While it updates some types (e.g., changing `(int *)(a1 + 28)` to `(unsigned int *)(a1 + 28)`), it leaves most variables unchanged, resulting in minimal improvement. GPT-4 also fails to provide useful information about the fields of user-defined structures, such as `IxpMsg`. This highlights the first challenge in using LLMs for recovering variable names and types from decompiled code: *understanding decompiled code is an inherently complex task that typically requires a human analyst years of skill training and adherence to well-designed methodologies [46, 73]. Therefore, expecting a general-purpose LLM to directly produce readable decompiled code is impractical.*

Additionally, We fine-tuned an LLM with 3 billion parameters [40] in an End-to-End (E2E) style, where the input is the decompiled code, and the output is the corresponding source code (Section 5.3.1). The goal was to enhance the model's proficiency in interpreting decompiled code. As shown in Fig. 3, it shows some progress, like accurately renaming `len`. However, it still falls short in providing comprehensive information, and sometimes even presents erroneous information, e.g., interpreting the field access `msg->mode` as `s->last`. A further concern is the semantic divergence observed between the source code and the outputs of the fine-tuned model and GPT-4, e.g., the data-flow alterations (marked in red in Fig.3).

```
1  unsigned long long sub_404056
2    (long long a1, unsigned short a2){
3   unsigned int16 v3;
4   unsigned int v4;
5   void *dest;
6   if(*(unsigned int *)(a1 + 28) == 1){
7    dest = *(void **)(a1 + 28);
8                       // should be `8`
9    v4 = v3 * a2;
10   sub_406BB9(v4);
11   }
12 }
```
GPT-4 Output

```
1   void process
2     (state *s, unsigned short n){
3    unsigned short len;
4    int size;
5    char *tmp;
6    if (s->last == 1) {
7     tmp = s->tmp;
8     size = len * n;
9     min(len); // should be `size`
10    }
11  }
12
```
Fine-tuned E2E Model Output

Figure 3: GPT-4 and direct fine-tuning are limited.

This shows the difficulty of preserving the semantics, highlighting the challenges in the task . Consulting a human expert revealed that such an accurate field access recovery is challenging even for professionals without access to the entire code context. This leads to the second challenge: *accurate variable name and type recovery requires cross-referencing multiple functions, yet LLMs face input token limitations, making it difficult to process entire programs.* Furthermore, the token limit of LLMs (e.g., 4,096) is often exceeded by even just two decompiled functions, exacerbating the challenge.

2.3 Our Technique

While automatic reverse engineering remains a challenging task, manual approaches have yielded significant successes [7, 26]. Consequently, we are motivated to **employ LLMs to replicate the reverse engineering process used by human experts**, especially considering the recent advancements in LLMs' ability to comprehend code snippets as well as humans. Drawing from recent observational studies on reverse engineers' methodologies [46, 73], and an interview with an experienced reverse engineer with eight years of experience in the field, we have gleaned several key insights.

One key insight highlighted by the study is that *the decompiled code, while somewhat similar to source code, exhibits distinct distributions.* These differences often challenge even experienced developers, who may struggle to understand the decompiled code despite years of development experience. For example, Fig. 1 demonstrates that the decompiled code includes numerous low-level operations, complicating comprehension. Similarly, this complexity poses a significant challenge to a general-purpose LLM or even a specialized code model for understanding decompiled code. To address this, we choose to fine-tune LLMs to familiarize them with the unique patterns of decompiled code, effectively transferring their comprehensive knowledge of source code to decompiled code. This process mimics teaching reverse engineering to a senior developer in practice.

However, as demonstrated earlier, fine-tuning an LLM for direct application in reverse engineering has limited success. The second insight that enhances our fine-tuned LLMs' understanding of decompiled code is: *while reverse engineering is challenging, human experts often simplify the task by breaking it down into smaller, manageable sub-tasks.* In recovering variable names and types, for instance, experts typically divide the task into two sub-tasks: 1) identifying local variable types and names, and 2) reconstructing user-defined structures. Although similar to some extent, these tasks require distinct knowledge bases and skill sets. The former focuses on interactions among different local variables, while the latter concentrates on the similarities and differences among the fields within the same structures. We adopt a similar approach by

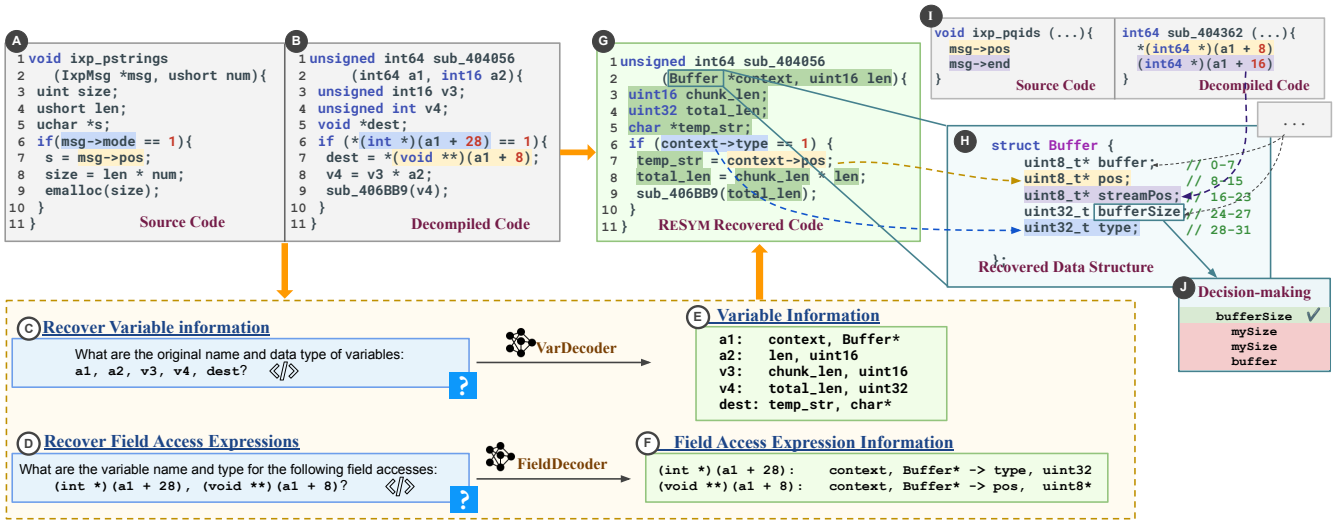

Figure 4: Motivating Example illustrating ReSym's hybrid approach synergizing insights from LLMs and program analysis to transform decompiled code (B) into (G). ReSym effectively recovers variable names and types (highlighted in green), field access expressions (blue and yellow), and the complete layout of user-defined data structure `struct Buffer` (H).

designing tasks with finer granularity, specifically prompting the LLMs for these two tasks instead of regenerating an entire function. By dividing the problem into these tasks and fine-tuning two distinct LLMs, we enable each LLM to focus on its specific, simpler task, thereby avoiding the confusion and difficulties that arise from task amalgamation.

While our fine-tuned LLMs show enhanced comprehension of decompiled code, they still face the constraints of token limitations. Our third insight regarding the manual reverse engineering process is: *as human experts can only focus on one piece of code at a time, they often gather insights from various code snippets and later cross-check them.* Typically, analysts connect code snippets by reasoning and analyzing data-flow. This suggests a workflow where LLMs first gain insights from individual code snippets and then cross-check these results with a reasoning component, where the predictions of both LLMs are aggregated. Therefore, LLMs are not obligated to process the entire program in one shot but can be prompted multiple times with different code segments, effectively circumventing the issue of input token limitations.

Driven by the aforementioned insights, we design and implement ReSym, an automatic reverse engineering technique that mimics the manual process by integrating fine-tuned LLMs (for raw results) with a lightweight Prolog-based reasoning system (to validate and aggregate results from diverse sources). Figure 4 demonstrates how ReSym works, using the function `ixp_pstrings` as an example with its source code (A), decompiled code (B), and the code reconstructed by ReSym (G). To gain raw results of good quality, ReSym performs two primary tasks: 1) recovering the names and types of local variables, and 2) extracting field access information, illustrated in (C) and (D), respectively. Leveraging the raw results from the LLMs ((E) and (F)), ReSym effectively reconstructs the original code in (G), where we mark all the modifications in green. This process includes recovering variable names and types (e.g., from `int64 a1` to `Buffer *context`) and field access expressions (e.g., from `(int *)(a1 + 28)` to `context->type`, highlighted in blue). Additionally, by cross-referencing the results from the LLMs with the reasoning component, ReSym accurately reconstructs the complete structure of `struct Buffer` (H). It achieves this detailed recovery by aggregating field access information from multiple functions (e.g., functions (B), (I), and others) via posterior reasoning (Section 3.4). During posterior reasoning, ReSym might encounter conflicting predictions from different functions, such as field name predictions (e.g., `bufferSize` vs. `mySize` in (J)). To resolve these, ReSym employs a similarity-biased voting system (Section 3.4.3), choosing `bufferSize` in this case.

3 APPROACH

3.1 Overview

Recognizing the unique nature of decompiled code and the insights from the reverse engineering process, ReSym carefully reduces the symbol recovery problem into two sub-tasks for the LLMs: 1) identifying names and types for local variables, and 2) extracting field access information. ReSym fine-tunes two models: VarDecoder and FieldDecoder, for these two respective tasks.

Fig. 5 presents the overview of ReSym, which consists of three phases. The first phase, *Training Data Preprocessing* (Section 3.2), prepares the data for fine-tuning VarDecoder and FieldDecoder. It is followed by the *Fine-Tuning* phase (Section 3.3), where the two models are fine-tuned separately. Finally, in the *Recovery* phase (Section 3.4), ReSym applies the fine-tuned models to the code decompiled from fully stripped binary files. The results from both models are then aggregated via *Posterior Reasoning* to get the final recovered code. Next, we discuss each step in detail.

3.2 Phase 1: Training Data Preprocessing

To fine-tune a model for recovering variable information and field access expressions, accurate alignment for each variable in the

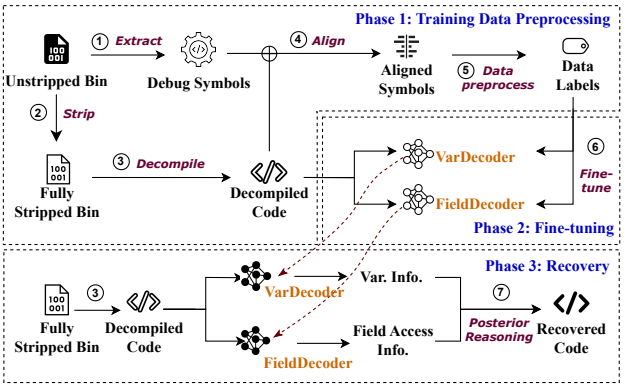

Figure 5: Overview of ReSym

decompiled code with its counterpart in the source code is essential for obtaining the ground truth. To enable the model to provide information on both *singleton variables* in decompiled code (e.g., `int v1` and `char *s`) and *field access expressions* (e.g., `(int *)(a1+28)`), we need to assign labels for these two types of data. In this phase (Fig. 5), we first extract the debug symbols from unstripped binary files (step 1). We then remove all debug symbols from them (step 2) and decompile them to generate the decompiled program (step 3). Finally, we align these debug symbols with the variables in the decompiled program (step 4), providing accurately labeled singleton variables for training VarDecoder and field access expressions for FieldDecoder.

3.2.1 Deriving Ground-truth Symbols for Decompiled Singleton Variables. In the decompiled code, local variables commonly appear as singleton variables. To correctly label a singleton variable in the decompiled code, it is essential to identify its corresponding variable in the source code. While a singleton variable in decompiled code usually matches directly with a local variable in the source code, this is not always the case. Sometimes, a singleton variable in decompiled code may represent an element of an array or a field within a structure. Fig.6 provides an example where `v2` in the decompiled code corresponds to `align` in the source code. However, the decompiler misinterprets a single 16-byte variable, `dt`, as a *cluster* of variables containing two 8-byte variables, `v3` and `v4`. A **cluster** is *a group of singleton variables in the decompiled code corresponding to an array or a structure in the source code.* Each of these variables represents a field of `struct CType`. Such misinterpretations significantly diminish code understanding, especially for user-defined structures. Recovering these clusters can be highly beneficial in enhancing code readability, yet this has been overlooked and remains unaddressed in previous research [11]. To handle it, the challenge lies in educating the model to identify these clusters through the careful design of variable labels for fine-tuning. In addition, we need to automatically detect these variables to generate their respective labels.

To accurately label singleton variables and identify clusters, we utilize the DWARF debug symbols, which contain rich information about variable names and types. As in Fig.5, debug symbols from binary files are used solely for training data construction and are excluded (stripped) during inference (Phase 3). Fig. 6 shows an example of the debug symbols containing detailed information about

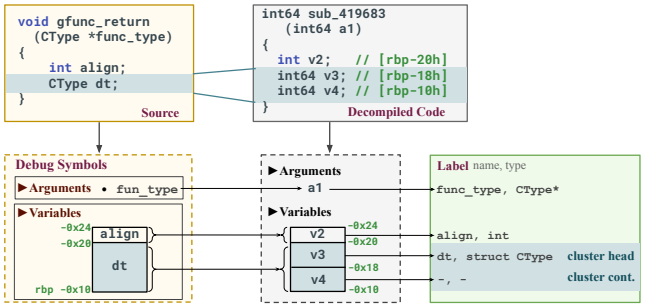

Figure 6: Data alignment and labeling for VarDecoder

arguments, variables, and their respective locations in memory. We then introduce our proposed Algorithm 1 for aligning and labeling the singleton variables and explain the process step by step.

Algorithm. In Algorithm 1, the function GetVarLabel (lines 3–17) takes the decompiled program (*Prog*) and the DWARF debugging symbols (*DBG*) and produces *VLabels*, containing labels (name and type) for singleton variables (in the decompiled code). Here, *Prog* represents the decompiled program derived from the fully-stripped binary, which encompasses arguments *Args*, variables *Vars*, and statements *Stmts*. Meanwhile, the debug symbols *DBG* extracted from the corresponding unstripped binary file contain *ASym* and *VSym*. *ASym* maps arguments to their symbols, while *VSym* maps variables to theirs. This symbol information contains the name, type, and memory location of each argument or variable as it appears *in the source code* (not the decompiled version).

Algorithm 1 Variable Labeling Algorithm

```
1:  Prog: Decompiled Program (Args, Vars, Stmts)
2:  DBG: DWARF Debug Symbols (ASym, VSym)
3:  function GetVarLabel(Prog, DBG) → VLabel
4:      VLabel ← {}
5:      sortedArgs ← Sort(DBG.ASym)
6:      for i ← 0 to Length(Prog.Args) − 1 do
7:          a ← Prog.Args[i]
8:          a′ ← sortedArgs[i]
9:          VLabel[a] ← (a′.name, a′.type)
10:     for v ∈ Prog.Vars do
11:         v′ ← AlignVar(v, DBG)
12:         if v′ ≠ Null then
13:             if v.start == v′.start then
14:                 VLabel[v] ← (v′.name, v′.type)
15:             else
16:                 VLabel[v] ← (" − ", " − ")
17:     return VLabel
18: function AlignVar(v, DBG)
19:     for v′ ∈ DBG.VSym do
20:         if v.start ≥ v′.start and v.end ≤ v′.end then
21:             return v′
22:     return Null
```

In Lines 5–9, the algorithm aligns the arguments in *Prog* (the decompiled code) sequentially with those in the debug symbols. Note that the mapping is always one-to-one, following the compiler conventions. For variables (Lines 10–16), each variable v in *Prog* is

aligned with its counterpart v' from the debug symbols via AlignVar (Lines 18–22). AlignVar iterates over v' in the debug symbols and returns v' if the address range (from `start` to `end`) of v is within that of v'. If no match is found, it returns `Null`. The rationale is that the decompiler may break a complex variable into multiple singleton variables. As such, the address range of a decompiled singleton variable must be a sub-range of the original complex variable. Consider Fig. 6 as an example. There is a *direct match* from `v2` to `align`, as their address ranges match. Both `v3` and `v4` lie within the address range of `dt` (from `rbp -0x10` to `-0x20`), forming a *cluster* where the two variables, `v3` and `v4`, correspond to a single source code variable, `dt`. Thus, `v3` and `v4` are both aligned with `dt`, with `v3` identified as the *cluster head*, and `v4` as a *subsequent variable* in the cluster.

Returning to the GetVarLabel function at line 11, if the aligned variable for v is `Null`, the variable is omitted. This exclusion, as also applied by existing works [11, 88], typically applies to variables introduced by compilers that do not have any meaningful correspondence in the original source anyway. Otherwise, if the starting address of v matches that of its aligned variable, suggesting a direct match or that it is a cluster head, we label v with the name and type of its aligned counterpart. For subsequent variables in a cluster, we assign a dash ("-") as their label.

We choose to use a dash instead of specific field names for variables in a cluster for several reasons. First, the number of variables in a cluster is not always the same as the number of fields in a structure. For example, the decompiler may interpret two fields as a single stack variable in the decompiled code, complicating the assignment of meaningful labels to each subsequent variable in the cluster. Second, clusters can represent structures or arrays. With arrays, there are no specific ground truth names for each element. Therefore, we decide to use dashes as labels for these subsequent variables in a cluster and leave the recovery of field information to later stages (e.g., with FieldDecoder and phase 3). During the inference stage, to distinguish whether a cluster is an array or a structure, we use the type predicted by VarDecoder for the cluster head: a primitive type indicates the cluster as an array, while a structure type suggests a structure.

3.2.2 Deriving Ground-truth Symbols for Decompiled Field Access Expressions. FieldDecoder focuses on recovering field access expressions in the decompiled code. Fig. 7 demonstrates the data alignment and labeling procedure for the expression `(int *)(a1 + 28)`, which accesses the 28th – 31st bytes of the data structure `a1` (base pointer) points to. Our procedure for aligning and labeling field access expressions is formally defined in Algorithm 2.

Algorithm. Algorithm 2 takes the decompiled program (*Prog*) and the debugging symbols (*DBG*) and outputs *FLabels*, containing labels for the field access expressions. For a given decompiled program *Prog*, we first use GetDerefExpression to identify all dereference expressions (line 3), each characterized by a base pointer v, an offset off, and a size sz. In Fig. 7, they correspond to `a1`, 28, and 4, which is the size of `int` type. For each expression, we align v with its source code counterpart using AlignVar (defined in Algorithm 1 lines 18–22). In Fig. 7, the aligned variable for $a1$ is `msg`. Expressions are excluded if the base pointer alignment fails (v' == Null), or if v does not directly correspond to a singleton

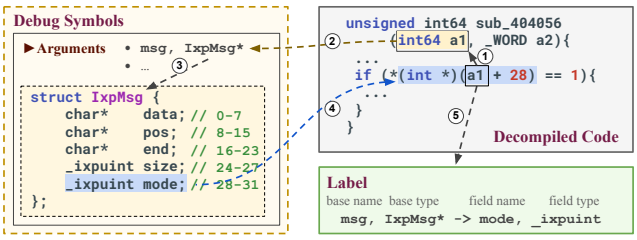

Figure 7: Data alignment and labeling for FieldDecoder

variable (v.start ≠ v'.start ∨ v.end ≠ v'.end). This exclusion process is primarily to filter out variables introduced by the compiler.

Algorithm 2 Feild Access Expression Labeling Algorithm

```
1:  function GetDerefLabel(Prog, DBG) → FLabels
2:      FLabelss ← {}
3:      DerefExp ← GetDerefExpression(Prog)
4:      for all (v, off, sz) ∈ DerefExp do
5:          v' ← AlignVar(v, DBG)
6:          if v' == Null ∨ v.start ≠ v'.start ∨ v.end ≠ v'.end then
7:              continue
8:          pointedType ← GetPointedType(v', DBG)
9:          if pointedType.isStruct() then
10:             f ← FindField(pointedType, off, sz)
11:             if f ≠ Null then
12:                 FLabels[v] ← (v'.name, v'.type, f.name, f.type)
13:         else
14:             FLabels[v] ← (v'.name, v'.type, " – ", " – ")
15:     return FLabels
```

Next, we determine the type pointed to by v' (line 8), such as `IxpMsg` for `msg` in Fig. 7. If this type is a structure, we use FindField to identify the accessed field, for example, `mode`, and assign labels including *base name*, *base type*, *field name*, and *field type* (line 12). The base name and type (`msg` and `IxpMsg*`) and the field name and type (`mode` and `_ixpuint`) are derived from the base pointer and accessed field, respectively. In (rare) cases where FindField fails to locate a corresponding field and returns `Null`, it suggests possible aggressive compiler optimization. We hence skip those cases. In line 13, when the *pointedType* is primitive, indicating a primitive type dereference or access to array elements, we assign a dash ("-") as the field name and type. As such, during inference, the model's predictions will help distinguish these from structure field accesses.

3.3 Phase 2: Fine-tuning

Upon completing data preparation, we fine-tune two LLMs for specific tasks: VarDecoder for recovering variable names and types, and FieldDecoder for extracting field access expression information.

3.3.1 VarDecoder. The recovery of stack variables is formulated as follows: given a decompiled function d and a list of its variables $V^d = \{v_1, \ldots, v_m\}$, we construct the input instruction I as "`What are the original name and data type of variables:` v_1, `. . . ,` v_m`?` $< d >$", which is a natural language question followed by the decompiled code ($< d >$). For each variable v_i asked in the instruction, the model is expected to generate the original name n_i and type t_i, with each in a line for easier postprocessing. The

output is formulated as

$$Y_1 = \begin{pmatrix} v_1 & n_1 & t_1 \\ v_2 & n_2 & t_2 \\ \vdots & \vdots & \vdots \\ v_m & n_m & t_m \end{pmatrix}$$

The objective of the fine-tuning process is to adjust the pre-trained LLMs' weights θ to minimize the loss L, which is the negative log-likelihood associated with generating the expected output from the given input instruction:

$$\begin{aligned} L &= -\log \mathrm{P}(Y_1 | I, \theta) \\ &= \underbrace{-\log \Big[\mathrm{P}(v_1 | I, \theta) + \mathrm{P}(n_1 | I, v_1, \theta) + \mathrm{P}(t_1 | I, v_1, n_1, \theta) \Big]}_{\text{Loss of recovering the first variable}} \\ &\quad \Bigg. - \log \sum_{i=2}^{m} \Big[\mathrm{P}\Big(v_i \Big| I, \underbrace{\{v_j,\ n_j,\ t_j\}_{j=1}^{i-1}}_{\substack{\text{Conditioned on} \\ \text{previous recovery}}},\ \theta\Big) \\ &\qquad + \mathrm{P}\Big(n_i \Big| I, \{v_j, n_j, t_j\}_{j=1}^{i-1}, v_i, \theta\Big) \\ &\qquad + \mathrm{P}\Big(t_i \Big| I, \{v_j, n_j, t_j\}_{j=1}^{i-1}, v_i, n_i, \theta\Big) \Big] \Bigg\} \text{Loss of recovering following variables} \end{aligned}$$

Note that for the same variable, the probability of the variable type recovery is conditioned on that of its name recovery, and recovery of the following variables is conditioned on the recovery of previous variables.

3.3.2 FieldDecoder. The formulation for recovering field access expressions is similar. Given a decompiled function d and a list of field access expressions $\mathbf{E} = \{e_1, \ldots, e_k\}$. The input instruction I is "`What are the variable name and type for the following field accesses:` $e_1, \ldots, e_k$`? <` d `>`". For each field access expression e_i, the model is designed to output the base name nb_i, base type tb_i, field name nf_i, and field type tf_i. The output is

$$Y_2 = \begin{pmatrix} e_1 & nb_1 & tb_1 & \rightarrow & nf_1 & tf_1 \\ e_2 & nb_2 & tb_2 & \rightarrow & nf_2 & tf_2 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\ e_k & nb_k & tb_k & \rightarrow & nf_k & tf_k \end{pmatrix}$$

The fine-tuning is to minimize the negative log-likelihood of the generation of nb_i, tb_i, nf_i, and tf_i, which is similar to the loss of fine-tuning VarDecoder.

3.4 Phase 3: Recovery

In this phase (Fig.5), we focus on recovery using fully stripped binaries, with the debug symbols removed. The initial step involves extracting decompiled code from these fully stripped binaries. Following this, we apply the fine-tuned VarDecoder and FieldDecoder to the decompiled code, following the prompt format detailed in Section 3.3. Note that FieldDecoder is employed only if there are field access expressions in the code. The final stage, **posterior reasoning**, processes the raw results obtained from individual functions and aggregates the outputs of the LLMs from multiple functions to recover information that requires a global view.

It is important to note that type information, particularly regarding user-defined structures, can often be incomplete or misleading when derived from a single code snippet. This problem commonly occurs if the function under examination does not access all fields of the subject structure. Inspired by existing efforts [60], we have devised a Prolog-based inference system [14] that mimics the reasoning process of human experts. This approach enables efficient cross-referencing of all variables related to a specific type. The process employs a similarity-biased voting system to determine the type, relying on initial predictions for all variables linked to the type in question, called *facts*. In the following, we will present the fact predicates (Section 3.4.1) and reasoning rules (Section 3.4.2) employed by ReSym, and discuss the similarity-biased voting system (Section 3.4.3).

Table 1: Fact predicates used by ReSym

	Name	Description
Initial Facts	Var(F, V, Ty, Nm)	In function F, variable V is of type Ty and named Nm according to VarDecoder.
	FieldAcc(F, V, Off, Sz, Ty, Nm)	In function F, variable V has a field access at offset Off and size Sz, with the field's type and name predicted as Ty and Nm, respectively, by FieldDecoder.
	Callsite(F_1, F_2, C, V, Id)	At callsite C, caller function F_1 invokes callee function F_2, passing variable V from F_1 as the Id-th argument.
	Arg(F, V, Id)	The Id-th argument of function F is variable V.
Derived Facts	DataFlow(F, V_1, V_2)	In function F, there is a data flow between variables V_1 and V_2.
	TypeAgnosticArg(F, Id)	The Id-th argument of function F is type-agonstic, i.e., variables are typically type-casted before being passed as the argument.
	TypeGrouping(F, V, Tg)	Variable V in function F is grouped into a type group Tg.
	TypeInfo(Tg, Ty, Nm)	A variable in type group Tg is predicted to be of type Ty with the name Nm.
	FieldInfo(Tg, Off, Sz, Ty, Nm)	A variable in type group Tg accesses a field at offset Off and size Sz, with the field's type and name predicted as Ty and Nm, respectively.

3.4.1 Fact Predicates. Table 1 presents the fact predicates utilized by ReSym, with the predicates and their descriptions in the two columns. The facts fall into two categories, *initial facts* and *derived facts*.

Initial facts denote facts directly derived from the predictions of VarDecoder and FieldDecoder, and the decompiled code's syntax. There are four types of initial facts in total. Specifically, **Var** and **FieldAcc** encode the results from VarDecoder and FieldDecoder, respectively. For instance, in Fig.4, (E) contains Var(sub_404056, a2, uint16, len), and (F) includes FieldAcc(sub_404056, a1, 28, 4, uint32, type), where 4 is the field access size, i.e., size of "`int`". The facts **Callsite** and **Arg** illustrate syntax features in a decompiled function, where Callsite details callsite information, and Arg describes the function signature. For example, in code (B) of Fig.4, we find Callsite(sub_404056, sub_406BB9, LINE-9, v4, 1) and Arg(sub_404056, a1, 1).

In contrast, there are five derived facts representing the reasoning results (both intermediate and final) during the analysis process. The fact **DataFlow** represents the results of the data-flow analysis. To minimize runtime overhead, we utilized a lightweight version of Andersen's algorithm [4] on decompiled code, avoiding the heavy data-flow analysis upon binary executables [89]. Although DataFlow cannot be directly acquired and requires analysis efforts, encoding data-flow analysis in Prolog is a well-established practice [77]. For simplicity, we omit the detailed inference rules and present it solely as a fact. The fact **TypeAgnosticArg** describes certain function arguments as type-agnostic, commonly seen in

$$\frac{\text{TRUE}}{\text{TypeGrouping}(F, V, \text{Tg}^{F.V})} \quad (1) \qquad \frac{\text{DataFlow}(F, V_1, V_2),\ \text{TypeGrouping}(F, V_1, Tg)}{\text{TypeGrouping}(F, V_2, Tg)} \quad (2)$$

$$\frac{\text{Callsite}(F_1, F_2, C, V_1, Id),\ \text{Arg}(F_2, V_2, Id),\ \neg\text{TypeAgnosticArg}(F_2, Id),\ \text{TypeGrouping}(F_1, V_1, Tg)}{\text{TypeGrouping}(F_2, V_2, Tg)} \quad (3)$$

$$\frac{\text{Callsite}(F_1, F_2, C, V_1, Id),\ \text{Arg}(F_2, V_2, Id),\ \neg\text{TypeAgnosticArg}(F_2, Id),\ \text{TypeGrouping}(F_2, V_2, Tg)}{\text{TypeGrouping}(F_1, V_1, Tg)} \quad (4)$$

$$\frac{\text{TypeGrouping}(F, V, Tg),\ \text{Var}(F, V, Ty, Nm)}{\text{TypeInfo}(Tg, Ty, Nm)} \quad (5) \qquad \frac{\text{TypeGrouping}(F, V, Tg),\ \text{FieldAcc}(F, V, Off, Sz, Ty, Nm)}{\text{FieldInfo}(Tg, Off, Sz, Ty, Nm)} \quad (6)$$

$$\frac{\begin{array}{c}\text{TypeGrouping}(F_1, V_1, \text{Tg}^1),\ \text{Callsite}(F_1, F_3, C_1, V_1, Id),\ \text{FieldInfo}(\text{Tg}^1, \text{Off}_1, Sz_1, \text{-}, \text{-}),\\ \text{TypeGrouping}(F_2, V_2, \text{Tg}^2),\ \text{Callsite}(F_2, F_3, C_2, V_2, Id),\ \text{FieldInfo}(\text{Tg}^2, \text{Off}_2, Sz_2, \text{-}, \text{-}),\\ \overline{\text{FieldOverlap}}^{*}(\text{Off}_1, Sz_1, \text{Off}_2, Sz_2)\end{array}}{\text{TypeAgnosticArg}(F_3, Id)} \quad (7)$$

$*$: $\text{Off}_1 + Sz_1 > \text{Off}_2$, assuming $\text{Off}_1 \le \text{Off}_2$ without losing generality

Figure 8: Reasoning rules used by ReSym

low-level memory functions (e.g., `memset`) or container APIs (e.g., `HashMap::insert`). For example, the first parameter of `memset` is type-agnostic, as it can point to any data type, e.g., `memset((void *)&fd, 0, 32)` and `memset((void *)str, 0, 128)`. Identifying type-agnostic arguments is crucial for posterior reasoning, as type information should not propagate through them. For instance, in the earlier `memset` example, we cannot conclude that `&fd` and `str` are of the same type. Also note that detecting type-agnostic arguments is more challenging at the binary level than in source code, due to the absence of explicit type casting in the source code. The fact **TypeGrouping** indicates that a variable V in function F belongs to a type group Tg, a group of variables believed to be of the same type. For example, at the callsite in Fig. 4 (B) line 9, we have TypeGrouping(sub_404056, v4, Tg), where Tg is the type group that the first argument of the callee function `sub_406BB9` belongs to. TypeGrouping is particularly useful for representing the type assignment of variables. **TypeInfo** and **FieldInfo** provide aggregated information for each type group. Specifically, if any variable in the type group Tg is predicted to have a type Ty and a name Nm, we record it using TypeInfo, and similarly for FieldInfo. These two derived facts, TypeInfo and FieldInfo, are then input into the similarity-biased voting system to aid ReSym in determining variable names and types.

3.4.2 Reasoning Rules. Fig. 8 presents the reasoning rules used by ReSym. These rules are expressed in the following format:

$$\frac{P_1, \quad P_2, \quad P_3, \quad \ldots, \quad P_n}{C}$$

where P_i denotes the i-th premise of the rule and C represents the conclusion. When all the premises are present in the current fact base, ReSym adds the conclusion to the fact base as well. Rule (1) states that each variable is initially assigned to a type group containing only itself. Rule (2) covers intra-procedural type inference, specifying that if there is data flow between two variables, V1 and V2, then V2 should belong to V1's type group, and vice versa. Rules (3) and (4) describe inter-procedural type inference. Rule (3) states that if function F_1 calls function F_2, passing variable V_1 as F_2's Id-th argument, and if V_2, the Id-th argument of F_2, is not a type-casting argument, then V_2 should belong to V_1's type group. Similarly, Rule (4) states that under the same condition, V_1 should belong to V_2's type group. Note that, if Id-th argument of F_2 is a type-agnostic argument, e.g., the first argument of `memset`, these two rules do not apply. Rules (5) and (6) address the process of aggregation, stating that any prediction made by VarDecoder and FieldDecoder about a variable will apply to every type group that the variable belongs to. Rule (7) outlines the process of identifying a type-agnostic argument. Specifically, if two variables, V_1 in F_1 and V_2 in F_2, belong to groups Tg^1 and Tg^2 respectively and are passed as the Id-th argument of F_3, any overlapping field accesses of Tg^1 and Tg^2 will categorize the Id-th argument of F_3 as a type-agnostic argument. Overlapping field accesses refer to two distinct field accesses sharing the same memory, e.g., access at offset 0 with size 8 and access at offset 4 with size 4. Rule (7) is based on the premise that field access offsets and sizes (identified by IDA) are typically reliable and rarely erroneous, suggesting that overlapping field access likely indicates a type collision, thereby making the argument under consideration type-agnostic. It is important to note that $\overline{\text{FieldOverlap}}$ is an auxiliary Prolog method for detecting such overlaps, rather than a fact predicate.

Hypothetical Reasoning. It is observed that Rule (3) and Rule (7) form a logical loop, where Rule (3) requires the facts of TypeAgnosticArg to detect TypeGrouping, while Rule (7) depends on TypeGrouping to identify TypeAgnosticArg. This often leads to a deadlock when recursion occurs in the code. To resolve this, we employ a trial-and-error method, akin to the strategies used by human analysts during reverse engineering type inference. Typically, analysts initially presume an argument is not type-agnostic and proceed with their reasoning, revising this assumption if conflicts arise. The Prolog system inherently supports such backtracking [61]. Our approach mirrors this methodology: initially considering an unvisited TypeAgnosticArg as false if the engine fails to generate new facts, and then continuing with the reasoning process. In the event of a conflict, we use Prolog to address the root cause of the problem (i.e., an incorrect assumption). This process is repeated until all potential TypeAgnosticArg facts in the code base are verified.

3.4.3 Similarity-biased Voting System. The primary goal of posterior reasoning is to conclude the type using the type predictions of LLMs across various functions. This is especially crucial in reconstructing user-defined structures, as inferring the complete structure from a single function is often impractical due to fields in these structures usually being accessed across multiple functions. We define the problem as follows, given a type group Tg and a specific offset Off (if applicable), our goal is to determine the fields' size $\bar{\text{Sz}}$, type $\bar{\text{Ty}}$, and name $\bar{\text{Nm}}$ using aggregated results { (Sz, Ty, Nm) | FieldInfo(Tg, Off, Sz, Ty, Nm) }. Conflicts may arise among predictions from different functions. For instance, in Fig. 4 (H), the fourth field, used in four different functions, receives varied name predictions like `bufferSize`, `mySize`, and `buffer`, as shown in (J). A simple majority vote, while effective for field sizes, may not be optimal for field types and names. In the aforementioned example, `mySize` might be erroneously selected because this method overlooks insights from similar predictions, such as `buffer` for `bufferSize`. To address this, we implement a novel, similarity-biased voting system. We use a similarity metric, $TSim_n$, to assess the resemblance of one name n to other predictions, $N \setminus n$, and select the one with the highest score. We adhere to three primary naming conventions,

camelCase, *PascalCase*, and *snake_case*, breaking down each name n into tokens $W_n = w_1, w_2, ...$ For instance, `bufferSize` becomes `buffer`, `size`. We furturer define $TSim_n$ as:

$$TSim_n = \sum_{n' \in N \setminus \{n\}} \frac{2 \times |W_n \cap W_{n'}|}{|W_n| + |W_{n'}|}$$

This metric represents the ratio of overlapping tokens to the total number of tokens. For instance, $TSim_{\texttt{bufferSize}}$ is $\frac{5}{3}$, whereas for `mySize` and `buffer` are $\frac{3}{2}$ and $\frac{2}{3}$, respectively. Consequently, `bufferSize` is chosen as the candidate that best reflects consensus. Intuitively, our method ensures that names sharing common components or meanings with other predicted values receive higher scores. This approach considers not just the frequency, but also the commonality and information embedded in each predicted value, making it more robust than a simple majority vote.

It is worth noting that a more complex similarity metric could involve using a standalone deep learning model to generate high-dimensional embeddings for each prediction and calculate similarity accordingly. However, we find this unnecessary in practice. Also, note that we apply the same methodology for determining type names based on TypeInfo.

3.5 Practical Challenges

We additionally addressed several practical challenges as follows:

Filtering Inaccurate Type-Size Predictions. In rare instances, VarDecoder and FieldDecoder may incorrectly predict types that are inconsistent regarding variable sizes. For example, VarDecoder may predict a pointer type (of 8 bytes) for a 2-byte variable. We refine the encoding of Var and FieldAcc to exclude such inconsistent predictions when determining the final type or name with the similarity-biased voting system (Section 3.4.3).

Additional Reasoning Rules. Beyond the rules outlined in Fig.8, we incorporated standard type inference rules into ReSym. For instance, de-referencing a pointer assigns the pointed type to the outcome variable. These standard rules, while not depicted in Fig.8, have been implemented in ReSym.

Field Overlap Due to Compiler Optimization. Rule (7) in Fig.8 suggests that overlapping field accesses typically indicate a type-agnostic argument. However, in rare cases, these overlaps might result from compiler optimizations [88], rather than the presence of a type-agnostic argument. To reduce this interpretative noise, we established a threshold for Rule (7), requiring ReSym to detect a minimum of three instances of overlapping field accesses before considering the subject argument as a type-agnostic argument.

Synonym Replacement before Similarity-biased Voting. Developers often use synonyms for the same purpose. For example, `buf` and `buffer` are used interchangeably. We include a set of common synonyms, replacing synonym tokens before computing similarity metrics. The complete list can be found in our artifacts [1].

4 EXPERIMENTAL SETUP

4.1 Dataset

To train the models and evaluate ReSym on real-world projects, we follow the method of previous work [11] and build a new dataset by compiling 7,416 popular C and C++ projects into 113,696 binary files from GitHub using GHCC [33]. Particularly, all projects were created in 2012 – 2022 with more than 20 stars. C++ projects constitute 6.5% of our dataset, consistent with the distribution in DIRT(6% [11]). We only include executable binary programs in our dataset, precluding intermediate relocatable binary files since the semantics of a relocatable file rely on their symbol table [76], which may be stripped away.

Among some projects, we observed significant overlap in functions and data structures, which may affect the diversity of the dataset and yield misleading results. We sanitize the dataset by retaining binaries with at least 80% unique functions, resulting in 16,217 binary files. The binary files have an average size of 116.2 KB and a maximum of 8.9 MB. Function similarity was determined through exact string matching of the decompiled functions with necessary normalization, following existing practices [11, 38]. In the decompiled code, local variables are annotated according to their declaration order and hence are normalized by nature. Additionally, we normalized function names and global variables.

The decompiled code is obtained from fully stripped binary files using the decompiler IDA Pro [32]. Our analysis revealed that, on average, 71.1% of data structures within a project appear in more than one binary file. Therefore, we choose to split the dataset by project to minimize function and structure overlaps between training and testing sets. The dataset was divided into training and test sets, denoted as $D = \{D_{\text{Train}}, D_{\text{Test}}\}$, with a ratio of 0.95. This splitting method guarantees that functions from the same project are exclusively assigned to either D_{Train} or D_{Test}. Field access expressions were identified using Clang [42]. Note that we do not consider predicting symbols for compiler-generated variables. For VarDecoder, functions without variables needing renaming or retyping were removed. Due to resource constraints and the rarity of clusters (Section 3.2.1) presence (only 5.9% of functions have it), we included the functions with clusters twice and sampled only 50% of the functions without clusters in D_{Train} for fine-tuning VarDecoder. This resulted in 269,735 functions with 1,176,417 variables for training, among which 20,411 functions (before resampling) have clusters, and 26,945 functions with 108,132 variables for testing. For FieldDecoder, functions without field access expressions were excluded, yielding 159,577 functions with 528,956 expressions for training, and 9,121 functions with 33,793 expressions for testing.

Functions in Training vs. Functions not in Training. Following previous work [11], we consider two subsets of functions within the testing set: functions that are present in the training set ("Functions in Training") and those that are not ("Functions not in Training"). The former typically includes library functions commonly reused by different developers, which the models could have encountered during training. While the inclusion of such functions in the testing set is practical for a realistic assessment, we separately evaluate the performance on these two subsets to show ReSym's ability to memorize seen data and generalizability on unseen data. We use string matching to determine the same (decompiled) functions as the previous works [11, 38] with normalization steps as discussed earlier in this section.

Existing Dataset. We choose not to use an existing dataset DIRT [38]. The main reason is that it does not provide source binary files or

decompiled code from fully-stripped binaries, which limits data labeling (as outlined in Section 3.2), particularly for obtaining necessary debug symbols and fully stripped binaries.

4.2 Training

We select pre-trained StarCoder 3B [40], one of the state-of-the-art LLMs, as the base model for both VarDecoder and FieldDecoder. During the training step, for VarDecoder, inputs exceeding 2,048 tokens are truncated, and those over 4,096 tokens are discarded. For FieldDecoder, we discard any input exceeding 4,096 tokens and choose not to truncate inputs to avoid truncating some field access expressions that may lead to misinterpretation of the model. Note that no functions are discarded during the inference stage, and they are only truncated if the input exceeds the model's inherent 8,192-token limit. Function discarding and truncation occurs only during training for efficiency purposes, which is a common practice in LLM training [6, 44, 69] and has been shown to have a negligible impact on model performance [6, 44, 64, 69].

The fine-tuning was conducted on four A100 GPUs for three epochs. The model weights are updated using an AdamW [43] optimizer with a batch size of 16 and a learning rate of 5e-5, adjusted by 500 warm-up steps followed by a cosine decay.

4.3 Metrics

We report the performance of VarDecoder on recovering variable information and FieldDecoder on its recovery of field access expressions. In addition, we access the capability of ReSym's recovery on user-defined structures with posterior reasoning.

4.3.1 VarDecoder: Variable Name and Type Recovery. For VarDecoder, we assess the accuracy of the variable name and type predictions using a **perfect match** criterion [11, 38]. The accuracy for names, defined as $Acc_{\mathsf{n}} = \frac{Nm_c}{|V|}$, is calculated by dividing the total number of correctly predicted names Nm_c by the total number of variables across all decompiled functions in the test set $|V| = \sum_{\mathbf{d} \in D_{\text{test}}} |V^d|$. Similarly, the accuracy for type is $Acc_{\mathsf{n}} = \frac{Ty_c}{|V|}$ with Ty_c being the count of correctly predicted types for variables across all functions in the test set. We further evaluate VarDecoder's ability to identify variable clusters, i.e., *stack-inlined structures and arrays*, as introduced in Section 3.2.1 with precision, recall, and F1 score. We consider a cluster correctly predicted if and only if VarDecoder identifies the exact same set of variables with the ground truth. Precision is the rate of correctly predicted clusters over the total number of predicted clusters. Recall is the rate of correctly predicted clusters over the total number of clusters in the ground truth. The F1 score is the harmonic mean of precision and recall.

4.3.2 FieldDecoder: Field Access Expression Recovery. The performance of FieldDecoder is evaluated based on its accuracy in predicting the four properties for each field access expression: base name, base type, field name, and field type. We report accuracy for each of these four properties. The accuracy metrics are calculated similarly to Acc_{n}, as defined previously.

4.3.3 User-Defined Data Structure Recovery. For each decompiled function d with variables V^d, we evaluate each variable $u \in U^d$ that is by ground truth or predicted as a structure or a pointer to a structure, where $U^d \subset V^d$. For example, in Fig. 4, $V^d = \{\texttt{a1}, \texttt{a2}, \texttt{v3}, \texttt{v4}, \texttt{dest}\}$, and $U^d = \{\texttt{a1}\}$ since `a1` by ground truth and predicted as a pointer to a structure. We evaluate the *struct layout* and *struct annotation* for these variables. It is important to note that we do not consider arrays and pointers to arrays in this context, as arrays have already been evaluated as variable clusters in VarDecoder. Similarly, pointers to arrays are essentially pointers to their corresponding primitive types and are thus evaluated with VarDecoder as well.

Table 2: ReSym-VarDecoder's accuracy (%) on variable name and type prediction.

Overall		In Train		Not in Train	
Name	Type	Name	Type	Name	Type
56.4	64.6	88.2	89.5	37.5	49.9

Table 3: ReSym-VarDecoder's Precision (P), Recall (R), and F1 score (F1) (%) on identifying clusters.

Overall			In Train			Not in Train		
P	R	F1	P	R	F1	P	R	F1
83.9	88.0	85.9	93.0	97.4	95.2	80.2	84.2	82.1

Struct Layout. We evaluate structure layout based on the predicted offsets and sizes for each field. Following existing work [88], we use a rigorous standard that considers a field as correct if and only if both the offset and size are correct. We define the predicted layout for variable u as $Pred_L^u = \{\langle o_1, s_1\rangle, \langle o_2, s_2\rangle, ...\}$ which contains a set of *fields* represented as *offset-size pairs*. For example, in Fig. 4, ReSym's predicted layout for `a1` (H) is $Pred_L^{\texttt{a1}} = \{\langle 0, 8\rangle, \langle 8, 8\rangle, \langle 16, 8\rangle, \langle 24, 4\rangle, \langle 28, 4\rangle\}$. Similarly, we define Gt_L^u as the ground truth layout, which in this case is the same with $Pred_L^{\mathsf{u}}$ as the predicted layout is correct. We thereby define the precision and recall of the struct layout as:

$$Pre_L = \frac{\sum_{u \in U} |Gt_L^u \cap Pred_L^u|}{\sum_{u \in U} |Pred_L^u|}, \; Rec_L = \frac{\sum_{u \in U} |Gt_L^u \cap Pred_L^u|}{\sum_{u \in U} |Gt_L^u|}$$

where $U = \bigcup_{d \in D_{\text{Test}}} U^d$. The F1 score is the harmonic mean of precision and recall $F1_L = \frac{Pre_L \cdot Rec_L}{Pre_L + Rec_L}$.

Struct Annotation. The struct annotation includes the struct types (e.g., `Buffer` in Fig. 4 (H)), field names, and field types (e.g., `pos` and `uint8_t *` of the second field). We compare them against all the fields of the ground truth and report the accuracy. If ReSym fails to detect the field, the field name and type are both considered wrong.

5 EVALUATION

In this section, we report ReSym's performance (Section 5.1), comparative experiments with existing approaches (Section 5.2), three ablation studies (Section 5.3), and a case study on a real-world malware (Section 5.4).

5.1 Our Results

5.1.1 Name and Type Recovery with VarDecoder. Table 2 presents the performance of ReSym VarDecoder in predicting variable names and types. The model demonstrates an overall accuracy of 56.4% for

Table 4: ReSym-FieldDecoder's accuracy (%) on predicting base name ($Name_b$), base type ($Type_b$), field name ($Name_f$), and field type ($Type_f$) for field access expressions.

Overall				In Train				Not In Train			
$Name_b$	$Type_b$	$Name_f$	$Type_f$	$Name_b$	$Type_b$	$Name_f$	$Type_f$	$Name_b$	$Type_b$	$Name_f$	$Type_f$
54.4	53.8	55.2	55.7	89.0	87.5	89.0	89.0	34.0	33.9	35.3	36.0

Table 5: ReSym's results of user-defined data structures (%).

Struct Layout			Struct Annotation (Accuracy)		
Precision	Recall	F1	Struct Type	Field Name	Field Type
72.9	28.9	41.4	46.8	16.3	16.9

name and 64.6% for type recovery. Notably, while ReSym exhibits high accuracy (over 85%) for variables within functions "In Train", it also shows great generalizability for variables from functions "Not in Train", achieving 37.5% accuracy in name and 49.9% in type recovery.

Table 3 shows the effectiveness of ReSym-VarDecoder on identifying variable clusters (Section 3.2.1). ReSym effectively identifies clusters in both seen and unseen data, achieving over 80% in precision, recall, and F1 score across both subsets. This high performance can be attributed to the earlier-mentioned fact that variable clusters predominantly consist of stack-inlined structures and arrays, which are adequately accessed within their enclosing functions. The advanced capabilities of LLMs enable VarDecoder to deliver accurate predictions when provided with adequate information.

5.1.2 Field Access Expression Recovery with FieldDecoder. Table 4 displays the effectiveness of ReSym's FieldDecoder in recovering field access expressions. The model achieves over 50% accuracy in predicting the base names, base types, field names, and field types. For variables from "Not in Train" functions, ReSym's accuracies remain above 30%, while it exceeds 85% for "In Train" functions.

5.1.3 User-Defined Data Structure Recovery. Table 5 outlines ReSym's performance in recovering user-defined data structures. Predominantly, ReSym successfully recovers struct layouts, with 72.9% of its predicted offsets and sizes accurate. The struct layout's relatively low recall rate of 28.9% arises from several factors. A primary issue is the discarding of some functions due to the token limit (Section 4.2). This often leads to incomplete calling contexts and potentially overlooked field access expressions, an inherent challenge with ML-based methods. Moreover, data-flow analysis is inherently undecidable [62], which impedes the collection of both sound and complete analysis results. Consequently, ReSym might miss some data-flow relations, leading to incomplete type grouping (Tg in Section 3.4.1). This means variables that are of the same type according to the source code might be divided into two distinct type groups. We believe that implementing a more sophisticated data-flow analysis could improve ReSym's performance. We leave it for future work.

For "Struct Annotation", ReSym achieves a 46.8% accuracy in correctly naming the data structures. For example, ReSym accurately predicts `struct Operand` for a data structure in the test set. The accuracies for "Field Name" and "Field Type" are relatively lower, primarily due to our strict evaluation criteria, where both the field name and type are considered wrong if ReSym fails to recover the field. Given the relatively low recall of 28.9% for "Struct Layout", it is expected that these accuracies would be correspondingly lower. Moreover, we use a strict evaluation metric, perfect match, as a correct criterion to evaluate variable names and types. For instance, in Fig. 2, field names with semantic similarities, such as `bufferSize` versus `size`, are marked as incorrect. Types like `uint8_t*` and `char*`, despite being functionally equivalent, are also considered incorrect under our strict evaluation criteria.

Overall, ReSym exhibits great proficiency in recovering variable information, field access expressions, and user-defined data structures. It demonstrates both its capacity to memorize seen data ("In Train" functions) and its ability to generalize to unseen data ("Not in Train" functions).

Table 6: Comparison on variable name and type accuracy (%).

Method	Overall				In Train				Not in Train			
	All		Struct		All		Struct		All		Struct	
	name	type	name	type	name	type	name	type	name	type	name	type
$ReSym_{Var}$	56.7	60.7	50.0	55.6	74.6	75.0	74.4	75.7	45.1	51.4	41.8	48.9
DIRTY [11]	48.7	55.8	38.6	39.6	70.4	73.5	62.3	59.2	34.8	44.5	31.0	33.3

5.2 Comparison with Prior Work

We compare ReSym on name and type recovery with DIRTY [11], as well as recovering *struct layout* of user-defined data structures with OSPREY [88] and DIRTY [11].

5.2.1 Name and Type Recovery. We reproduced DIRTY using our dataset of fully stripped decompiled functions, maintaining the same settings as the original study, i.e., split the training-testing set by binaries and having a token limit of 1,024. We followed DIRTY's preprocessing and training procedures to train the DIRTY model for 16 epochs. Note that the variable clusters are not included in the training/testing data in this experiment as they are out of the scope of DIRTY. Meanwhile, we fine-tuned ReSym-VarDecoder under identical settings for a direct comparison, with results in Table 6 as "$ReSym_{Var}$". Additionally, we report the performance of variables with structure types as DIRTY does in its paper.

There are discrepancies between our reproduced results in Table 6 and DIRTY's reported numbers. The "Overall" results are influenced by the proportion of "functions in training", which varies from the datasets and the splitting. The "In Train" and "Not in Train" performance are also 2.1%–21.3% less than their reported numbers. This divergence primarily stems from two factors: 1) we use a different and sanitized dataset that minimizes training-testing data leaking (Section 4.1), and 2) different from DIRTY, our dataset contains fully stripped decompiled code as input (Section 4.1), which may be a harder task for models.

In Table 6, $ReSym_{Var}$ outperforms DIRTY across all columns, achieving an overall increase in accuracy by 8.0% for names and 4.9% for types. In addition, unlike DIRTY, which is limited to predicting previously seen names and types through multi-classification methods (Section 2.1), ReSym can recover field access expressions and identify previously unseen user-defined data structures.

5.2.2 Struct Layout Recovery. Our study compared *struct layout* recovery of user-defined data structures on Coreutils [67] benchmark among OSPREY [88], DIRTY [11], and ReSym. OSPREY categorizes variables into two groups: *visited* and *non-visited*. Visited variables

Table 7: Comparison with existing approaches on struct layout recovery in Precision (P), Recall (R), and F1 score (F1) (%).

Method	Overall			Visited			Non-Visited		
	P	R	F1	P	R	F1	P	R	F1
ReSym	**81.9**	34.6	**48.6**	**88.9**	40.5	55.6	**54.4**	**18.0**	**27.1**
Osprey [88]	38.1	**60.2**	46.7	68.7	**78.8**	**73.4**	2.6	7.3	3.9
DIRTY [11]	54.6	3.3	6.2	48.1	2.4	4.6	64.8	5.7	10.5

are those accessible from the main function, and non-visited are the rest of them. We use the binaries and results for Coreutils provided by OSPREY's authors for this comparison.

For our analysis, we applied our reproduced DIRTY and ReSym to this benchmark. DIRTY, employing a multi-classification model, predicts types of variables, including complete definitions for structures, e.g., `struct rlimit {rlim_t rlim_cur; rlim_t rlim_max;}`, from which layout information can be inferred. We transformed these fields into offset-size pairs, as introduced in Section 4.3.3. The comparative results are in Table 7.

Overall, ReSym achieves a 27.3% higher precision and a 1.9% higher F1 score than the best of the two approaches. However, ReSym shows a lower recall than OSPREY. The reasons for this discrepancy are in Section 5.1.3. It is important to note that OSPREY utilizes a complex data-flow analysis [89], which has considerable overhead and limited scalability. On average, ReSym analyzes a binary program from the Coreutils benchmark in 3.4s (1.4s inference time for VarDecoder and FieldDecoder and 0.6s for posterior reasoning), whereas OSPREY takes 528.24s. While DIRTY exhibits a higher overall precision than OSPREY, it falls short of the recall, which is only 3.3%. This is because DIRTY focuses on name and type recovery and lacks of design tailored for data structure recovery.

5.3 Ablation Study

To further evaluate our design choices, we conducted three ablation studies: 1) We evaluate our reduced task design (Section 3.1), where we let the LLMs only output essential information with strict guidelines, against an end-to-end model approach for regenerating entire functions (Section 5.3.1). 2) We compare the effectiveness of fine-tuning versus using LLMs with Few-shot Learning (FSL) (Section 5.3.2). 3) We analyze the impact of posterior reasoning (Section 3.4) by comparing results before and after it (Section 5.3.3).

5.3.1 End-to-End Fine-tuning. For an end-to-end approach, we fine-tune a StarCoder 3B model [40] to transform decompiled code directly into source code, where the output from this model can be arbitrary, making it challenging to automatically align with the original source code. To assess the results, we randomly sample 100 outputs and manually evaluate them against their ground truth, focusing on semantic integrity. Our analysis revealed that 68% of these outputs altered the original semantics. The dominant issues include alterations in statements (47%) and modifications in control flow (40%). These findings highlight the necessity of our reduced sub-tasks (Section 3.1) when prompting models.

5.3.2 Few-Shot Learning (FSL). FSL [8] employs pre-trained LLMs to adapt to new tasks with several examples. We use GPT-4 [53], with a token limit of 8,192, and compare its effectiveness of FSL with ReSym-VarDecoder on variable name and type recovery. As FSL's effectiveness generally increases with more examples, as shown in previous study [8], we select as many examples as possible within the token limit. Due to the limited resources, we conducted FSL on 100 randomly sampled functions from the test set, each supplemented by an average of 19.45 randomly selected training set functions as examples. Table 8 compares FSL and ReSym-VarDecoder on these functions. ReSym outperforms GPT-4 by 32.5% and 33.3%, despite using a model with significantly fewer parameters. This underscores the challenges LLMs face in deciphering decompiled code without fine-tuning.

Table 8: Comparison of accuracy (%) with few-shot learning.

Method	Name	Type
ReSym-VarDecoder	51.2	63.9
GPT-4	18.7	30.6

Table 9: Before and after posterior reasoning (%).

Method	Struct Layout			Struct Annotation (Accuracy)		
	Precision	Recall	F1	Struct Type	Field Name	Field Type
ReSym	72.9	**28.9**	**41.4**	**46.8**	**16.3**	**16.9**
$\text{ReSym}_{\text{Light}}$	**74.9**	16.8	27.4	42.6	9.2	9.5

5.3.3 Without Posterior Reasoning. Table 9 shows the user-defined structure recovery results before (row "$\text{ReSym}_{\text{Light}}$") and after (row "ReSym") posterior reasoning. Notably, the overall performance after aggregation is with a 14.0% improvement over the F1 score for "Struct Layout". "Struct annotation" accuracy also increases by 4.2% – 7.4%, reinforcing the importance of the aggregation process.

However, there is a slight decline (2.0%) in struct layout precision. This decrease primarily results from the analysis noise encountered during posterior reasoning, particularly in identifying type-agnostic arguments (Section 3.4). Recall that identifying type-agnostic arguments in binaries is a significant challenge due to the absence of type-casting statements. In some cases, ReSym may incorrectly group variables of different types into the same type group, leading to potential false positives in posterior reasoning and a consequent reduced precision. One potential solution is to incorporate an additional LLM to more accurately identify type-agnostic arguments. Despite this, the aggregation still notably identified 16,411 additional correct offsets and sizes, enhancing the recall by 12.1%.

5.4 Case Study with a Real-World Malware

To demonstrate ReSym's applicability in addressing security challenges and provide qualitative insights into its recovery capabilities, we apply it to Mirai [78], a real-world malware that conducts large-scale network attacks. In Fig. 9, we compare a function's source code, decompiled code, and the code recovered by ReSym.

Compared to the decompiled code, ReSym significantly enhances readability by recovering meaningful names and types for variables, e.g., from `unsigned int16 *v68` to `struct udp_hdr *udp` in line 5. In addition, ReSym adeptly recovers field access expressions, converting `(_WORD*)(v68+2)` to `udp->dst_port` in line 11, etc. On the right side of the figure is the comparison of the ground truth with ReSym's recovered data structure used on line 5. ReSym accurately reconstructs the complete layout with meaningful field names and

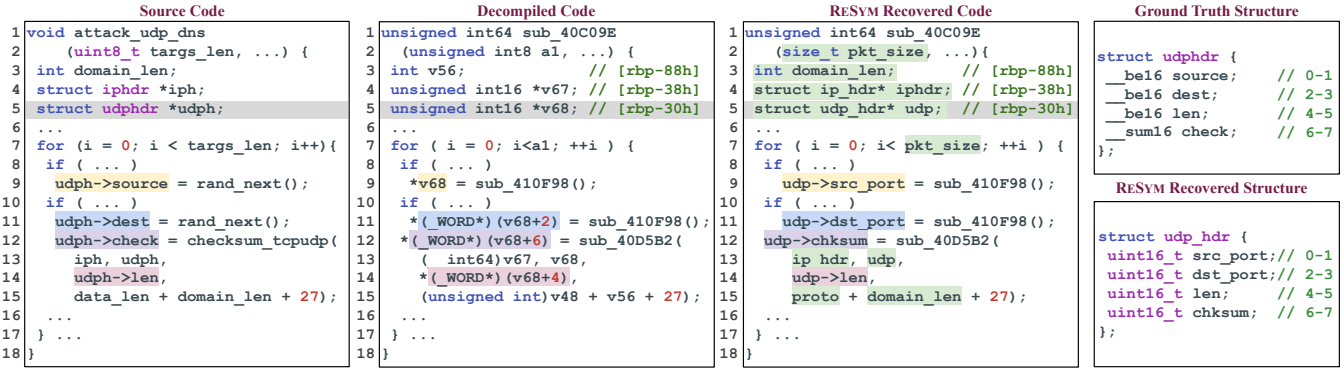

Figure 9: Case study: a recovered function and data structure in a real-world malware Mirai.

types. For example, it recovers the first field with the name `src_port` while the ground truth is `source`. As a result, the readability of the decompiled code is significantly enhanced thanks to ReSym.

6 RELATED WORK

6.1 Binary Analysis

Binary analysis is crucial in software security and engineering, impacting essential areas such as malware analysis [12, 24, 75, 85], vulnerability detection [5, 15, 30, 36, 55, 71], and software reuse [19]. Given the opaque nature of binary, many research papers focus on enhancing the maintainability and readability of binary code, such as translation [2, 27], similarity analysis [74, 82], memory analysis [54], identifying functions [37], recovering procedure names [16], etc. In this paper, we focus on recovering types and names of variables, field access expressions, and user-defined data structures. There have been previous efforts in relevant directions. DIRE [38], Direct [49], DIRTY [11], and LmPa [83] explored recovering variable names from the decompiled code. TIE [39], Retyped [50], and OSPREY [88] focus on recovering variable types. Similarly, Debin [31] and Cati [10] use machine learning approaches to predict debug information and types from stripped binaries. We discuss (Section 2.1) and compare (Section 5.2) ReSym with two state-of-the-art approaches DIRTY [11] and OSPREY [88].

6.2 Large Language Models

Modern large language models (LLMs), typically based on the Transformer [72] architecture, are categorized into three main structures: encoder-only, decoder-only, and encoder-decoder. Encoder-only LLMs, like BERT [18], RoBERTa [41], GraphBERT [87], CodeBERT [20], and GraphCodeBERT [28], focus on encoding natural language text or source code into vectorized embeddings for tasks such as similarity detection. Decoder-only LLMs, including GPTs [8, 52, 56], LLaMA [58, 70], and StarCoder [40], generate text or code auto-regressively, which are widely studied and adopted currently. Encoder-decoder LLMs like T5 [57] and CodeT5 [86] offer flexibility for both understanding and generation tasks.

LLMs have significantly advanced various code-related domains, thanks to their capabilities of understanding code and extensive knowledge of source code syntax and semantics acquired from massive pre-training datasets. These domains include code generation [3, 23, 29, 40, 48, 58, 86], code explanation and refinement [13, 45, 51, 59], software testing and fuzzing [17, 35], comments and specification generation [25, 81], automated program or vulnerability repair [34, 63, 79, 80], and proof synthesis [21, 84]. Recent work has explored LLMs' capabilities in binary-related tasks. CodeArt [66] pre-trains an attention-regularized BERT-like model on binary functions and explicit program dependencies to enhance binary code understanding. ProRec [65] augments binary code with source code contexts to improve binary summarization and function name recovery. LLM4Decompile [68] fine-tunes LLMs for decompilation, significantly increasing the re-executability rate. In this work, we fine-tune LLMs, addressing their limited understanding of binary or decompiled code, and combine them with program analysis methods to recover variable and data structure information from decompiled code.

7 CONCLUSION

In this paper, we present ReSym, an advanced system for automatic reverse engineering that effectively recovers symbol information from stripped binaries. ReSym reduces the complex task of symbol recovery into two specific sub-problems, utilizing two finely-tuned large language models, each tailored for a task. Integrating LLMs' insights with a Prolog-based reasoning system, ReSym substantially improves decompiled code readability. Our evaluation on real-world data shows ReSym's effectiveness, outperforming existing methods in variable and user-defined data structure recovery with 72.9% precision and identifying inlined structures or arrays with an 85.9% F1 score. ReSym's application to real-world malware further demonstrates its effectiveness in security challenges.

ACKNOWLEDGMENTS

We would like to thank all the anonymous reviewers for their constructive comments and feedback. We are grateful to the Center for AI Safety for providing computational resources. This research was supported in part by IARPA TrojAI W911NF-19-S0012; NSF 1901242, 1910300, and 2006688; ONR N000141712045, N000141410468 and N000141712947; and a CFI fund.

REFERENCES

[1] 2024. ReSym Artifact. https://github.com/lt-asset/resym/ Accessed: 2024-06-30.

[2] Iftakhar Ahmad and Lannan Luo. 2023. Unsupervised Binary Code Translation with Application to Code Clone Detection and Vulnerability Discovery. In *Findings of the Association for Computational Linguistics: EMNLP 2023*. 14581–14592.

[3] Wasi Ahmad, Saikat Chakraborty, Baishakhi Ray, and Kai-Wei Chang. 2021. Unified Pre-training for Program Understanding and Generation. In *Proceedings of the 2021 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*. Association for Computational Linguistics, Online, 2655–2668. https://doi.org/10.18653/v1/2021.naacl-main.211

[4] Lars Ole Andersen. 1994. Program analysis and specialization for the C programming language. (1994).

[5] Ioannis Angelakopoulos, Gianluca Stringhini, and Manuel Egele. 2023. FirmSolo: Enabling dynamic analysis of binary Linux-based IoT kernel modules. (2023).

[6] Xiao Bi, Deli Chen, Guanting Chen, Shanhuang Chen, Damai Dai, Chengqi Deng, Honghui Ding, Kai Dong, Qiushi Du, Zhe Fu, et al. 2024. Deepseek llm: Scaling open-source language models with longtermism. *arXiv preprint arXiv:2401.02954* (2024).

[7] Anthony Bouchard. 2022. New p0laris jailbreak for legacy iOS 9.x firmware released. https://www.idownloadblog.com/2022/04/20/p0laris-ios-9-jailbreak/ Accessed: 2024-01-01.

[8] Tom Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared D Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *Advances in neural information processing systems* 33 (2020), 1877–1901.

[9] Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2022. Codet: Code generation with generated tests. *arXiv preprint arXiv:2207.10397* (2022).

[10] Ligeng Chen, Zhongling He, and Bing Mao. 2020. Cati: Context-assisted type inference from stripped binaries. In *2020 50th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, 88–98.

[11] 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.

[12] Sanchuan Chen, Zhiqiang Lin, and Yinqian Zhang. 2021. SelectiveTaint: Efficient Data Flow Tracking With Static Binary Rewriting. In *30th USENIX Security Symposium (USENIX Security 21)*. 1665–1682.

[13] Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2023. Teaching Large Language Models to Self-Debug. arXiv:2304.05128 [cs.CL]

[14] William F Clocksin and Christopher S Mellish. 2003. *Programming in PROLOG*. Springer Science & Business Media.

[15] Victor Cochard, Damian Pfammatter, Chi Thang Duong, and Mathias Humbert. 2022. Investigating Graph Embedding Methods for Cross-Platform Binary Code Similarity Detection. In *2022 IEEE 7th European Symposium on Security and Privacy (EuroS&P)*. 60–73. https://doi.org/10.1109/EuroSP53844.2022.00012

[16] 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.

[17] Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. 2023. Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models. In *Proceedings of the 32nd ACM SIGSOFT international symposium on software testing and analysis*. 423–435.

[18] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2018. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. *CoRR* abs/1810.04805 (2018). arXiv:1810.04805 http://arxiv.org/abs/1810.04805

[19] Steven HH Ding, Benjamin CM Fung, and Philippe Charland. 2019. Asm2vec: Boosting static representation robustness for binary clone search against code obfuscation and compiler optimization. In *2019 IEEE Symposium on Security and Privacy (SP)*. IEEE, 472–489.

[20] Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A Pre-Trained Model for Programming and Natural Languages. *CoRR* abs/2002.08155 (2020). arXiv:2002.08155 https://arxiv.org/abs/2002.08155

[21] Emily First, Markus Rabe, Talia Ringer, and Yuriy Brun. 2023. Baldur: Whole-proof generation and repair with large language models. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1229–1241.

[22] Yanick Fratantonio, Antonio Bianchi, William Robertson, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. 2016. Triggerscope: Towards detecting logic bombs in android applications. In *2016 IEEE symposium on security and privacy (SP)*. IEEE, 377–396.

[23] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen tau Yih, Luke Zettlemoyer, and Mike Lewis. 2023. InCoder: A Generative Model for Code Infilling and Synthesis. arXiv:2204.05999 [cs.SE]

[24] Jian Gao, Xin Yang, Ying Fu, Yu Jiang, and Jiaguang Sun. 2018. VulSeeker: A semantic learning based vulnerability seeker for cross-platform binary. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 896–899.

[25] Mingyang Geng, Shangwen Wang, Dezun Dong, Haotian Wang, Ge Li, Zhi Jin, Xiaoguang Mao, and Xiangke Liao. 2023. An Empirical Study on Using Large Language Models for Multi-Intent Comment Generation. *arXiv preprint arXiv:2304.11384* (2023).

[26] GeoSn0w. 2022. New Blizzard Jailbreak released by GeoSn0w For iOS 9.0 – 9.3.6, 32-Bit Devices. https://idevicecentral.com/jailbreak-news/new-blizzard-jailbreak-released-by-geosn0w-for-ios-9-0-9-3-6-32-bit-devices/#google_vignette Accessed: 2024-01-01.

[27] Redha Gouicem, Dennis Sprokholt, Jasper Ruehl, Rodrigo CO Rocha, Tom Spink, Soham Chakraborty, and Pramod Bhatotia. 2022. Risotto: A Dynamic Binary Translator for Weak Memory Model Architectures. In *Proceedings of the 28th ACM International Conference on Architectural Support for Programming Languages and Operating Systems, Volume 1*. 107–122.

[28] Daya Guo, Shuo Ren, Shuai Lu, Zhangyin Feng, Duyu Tang, Shujie Liu, Long Zhou, Nan Duan, Alexey Svyatkovskiy, Shengyu Fu, Michele Tufano, Shao Kun Deng, Colin B. Clement, Dawn Drain, Neel Sundaresan, Jian Yin, Daxin Jiang, and Ming Zhou. 2020. GraphCodeBERT: Pre-training Code Representations with Data Flow. *CoRR* abs/2009.08366 (2020). arXiv:2009.08366 https://arxiv.org/abs/2009.08366

[29] Daya Guo, Qihao Zhu, Dejian Yang, Zhenda Xie, Kai Dong, Wentao Zhang, Guanting Chen, Xiao Bi, Y. Wu, Y. K. Li, Fuli Luo, Yingfei Xiong, and Wenfeng Liang. 2024. DeepSeek-Coder: When the Large Language Model Meets Programming – The Rise of Code Intelligence. arXiv:2401.14196 [cs.SE]

[30] Haojie He, Xingwei Lin, Ziang Weng, Ruijie Zhao, Shuitao Gan, Libo Chen, Yuede Ji, Jiashui Wang, and Zhi Xue. 2024. Code is not Natural Language: Unlock the Power of Semantics-Oriented Graph Representation for Binary Code Similarity Detection. In *33rd USENIX Security Symposium (USENIX Security 24), PHILADELPHIA, PA*.

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

[32] hex rays. 2024. IDA Pro. https://hex-rays.com/ida-pro/ Accessed: 2024-01-01.

[33] huzecong. 2024. GitHub Cloner & Compiler. https://github.com/huzecong/ghcc Accessed: 2024-01-01.

[34] Nan Jiang, Kevin Liu, Thibaud Lutellier, and Lin Tan. 2023. Impact of code language models on automated program repair. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 1430–1442.

[35] Sungmin Kang, Juyeon Yoon, and Shin Yoo. 2023. Large language models are few-shot testers: Exploring llm-based general bug reproduction. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. IEEE, 2312–2323.

[36] Dongkwan Kim, Eunsoo Kim, Sang Kil Cha, Sooel Son, and Yongdae Kim. 2022. Revisiting binary code similarity analysis using interpretable feature engineering and lessons learned. *IEEE Transactions on Software Engineering* 49, 4 (2022), 1661–1682.

[37] Soomin Kim, Hyungseok Kim, and Sang Kil Cha. 2023. FunProbe: Probing Functions from Binary Code through Probabilistic Analysis. In *Proceedings of the 31st ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 1419–1430.

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

[39] JongHyup Lee, Thanassis Avgerinos, and David Brumley. 2011. TIE: Principled Reverse Engineering of Types in Binary Programs. (2 2011). https://doi.org/10.1184/R1/6469466.v1

[40] Raymond Li, Loubna Ben Allal, Yangtian Zi, Niklas Muennighoff, Denis Kocetkov, Chenghao Mou, Marc Marone, Christopher Akiki, Jia Li, Jenny Chim, et al. 2023. StarCoder: may the source be with you! *arXiv preprint arXiv:2305.06161* (2023).

[41] Yinhan Liu, Myle Ott, Naman Goyal, Jingfei Du, Mandar Joshi, Danqi Chen, Omer Levy, Mike Lewis, Luke Zettlemoyer, and Veselin Stoyanov. 2019. RoBERTa: A Robustly Optimized BERT Pretraining Approach. arXiv:1907.11692 [cs.CL]

[42] LLVM. 2024. Clang: a C language family frontend for LLVM. https://clang.llvm.org/ Accessed: 2024-01-01.

[43] Ilya Loshchilov and Frank Hutter. 2017. Decoupled weight decay regularization. *arXiv preprint arXiv:1711.05101* (2017).

[44] Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2023. Wizardcoder: Empowering code large language models with evol-instruct. *arXiv preprint arXiv:2306.08568* (2023).

[45] Aman Madaan, Niket Tandon, Prakhar Gupta, Skyler Hallinan, Luyu Gao, Sarah Wiegreffe, Uri Alon, Nouha Dziri, Shrimai Prabhumoye, Yiming Yang, Sean Welleck, Bodhisattwa Prasad Majumder, Shashank Gupta, Amir Yazdanbakhsh, and Peter Clark. 2023. Self-Refine: Iterative Refinement with Self-Feedback. arXiv:2303.17651 [cs.CL]

[46] 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.

[47] NationalSecurityAgency. 2024. GHIDRA. https://ghidra-sre.org/ Accessed: 2024-01-01.

[48] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. Codegen: An open large language model for code with multi-turn program synthesis. *arXiv preprint arXiv:2203.13474* (2022).

[49] Vikram Nitin, Anthony Saieva, Baishakhi Ray, and Gail Kaiser. 2021. DIRECT: A Transformer-based Model for Decompiled Identifier Renaming. In *Proceedings of the 1st Workshop on Natural Language Processing for Programming (NLP4Prog 2021)*. 48–57.

[50] Matt Noonan, Alexey Loginov, and David Cok. 2016. Polymorphic type inference for machine code. In *Proceedings of the 37th ACM SIGPLAN Conference on Programming Language Design and Implementation*. 27–41.

[51] Theo X. Olausson, Jeevana Priya Inala, Chenglong Wang, Jianfeng Gao, and Armando Solar-Lezama. 2023. Demystifying GPT Self-Repair for Code Generation. arXiv:2306.09896 [cs.CL]

[52] OpenAI. 2024. ChatGPT. https://openai.com/blog/chatgpt Accessed: 2024-01-01.

[53] OpenAI. 2024. Models. https://platform.openai.com/docs/models/ Accessed: 2024-01-01.

[54] Kexin Pei, Dongdong She, Michael Wang, Scott Geng, Zhou Xuan, Yaniv David, Junfeng Yang, Suman Jana, and Baishakhi Ray. 2022. NeuDep: neural binary memory dependence analysis. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (<conf-loc>, <city>Singapore</city>, <country>Singapore</country>, </conf-loc>) *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 747–759. https://doi.org/10.1145/3540250.3549147

[55] Kexin Pei, Zhou Xuan, Junfeng Yang, Suman Jana, and Baishakhi Ray. 2023. Learning Approximate Execution Semantics From Traces for Binary Function Similarity. *IEEE Transactions on Software Engineering* 49, 4 (2023), 2776–2790. https://doi.org/10.1109/TSE.2022.3231621

[56] Alec Radford and Karthik Narasimhan. 2018. Improving Language Understanding by Generative Pre-Training. https://api.semanticscholar.org/CorpusID:49313245

[57] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2023. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. arXiv:1910.10683 [cs.LG]

[58] Baptiste Rozière, Jonas Gehring, Fabian Gloeckle, Sten Sootla, Itai Gat, Xiaoqing Ellen Tan, Yossi Adi, Jingyu Liu, Tal Remez, Jérémy Rapin, Artyom Kozhevnikov, Ivan Evtimov, Joanna Bitton, Manish Bhatt, Cristian Canton Ferrer, Aaron Grattafiori, Wenhan Xiong, Alexandre Défossez, Jade Copet, Faisal Azhar, Hugo Touvron, Louis Martin, Nicolas Usunier, Thomas Scialom, and Gabriel Synnaeve. 2023. Code Llama: Open Foundation Models for Code. arXiv:2308.12950 [cs.CL]

[59] Jérémy Scheurer, Jon Ander Campos, Tomasz Korbak, Jun Shern Chan, Angelica Chen, Kyunghyun Cho, and Ethan Perez. 2023. Training Language Models with Language Feedback at Scale. arXiv:2303.16755 [cs.CL]

[60] Edward J Schwartz, Cory F Cohen, Michael Duggan, Jeffrey Gennari, Jeffrey S Havrilla, and Charles Hines. 2018. 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*. 426–441.

[61] Ehud Shapiro and Akikazu Takeuchi. 1983. Object oriented programming in Concurrent Prolog. *New Generation Computing* 1, 1 (1983), 25–48.

[62] Johannes Späth, Karim Ali, and Eric Bodden. 2019. Context-, flow-, and field-sensitive data-flow analysis using synchronized pushdown systems. *Proceedings of the ACM on Programming Languages* 3, POPL (2019), 1–29.

[63] Benjamin Steenhoek, Md Mahbubur Rahman, Richard Jiles, and Wei Le. 2023. An Empirical Study of Deep Learning Models for Vulnerability Detection. In *2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*. 2237–2248. https://doi.org/10.1109/ICSE48619.2023.00188

[64] Jianlin Su, Murtadha Ahmed, Yu Lu, Shengfeng Pan, Wen Bo, and Yunfeng Liu. 2024. Roformer: Enhanced transformer with rotary position embedding. *Neurocomputing* 568 (2024), 127063.

[65] Zian Su, Xiangzhe Xu, Ziyang Huang, Kaiyuan Zhang, and Xiangyu Zhang. 2024. Source Code Foundation Models are Transferable Binary Analysis Knowledge Bases. *arXiv preprint arXiv:2405.19581* (2024).

[66] Zian Su, Xiangzhe Xu, Ziyang Huang, Zhuo Zhang, Yapeng Ye, Jianjun Huang, and Xiangyu Zhang. 2024. CodeArt: Better Code Models by Attention Regularization When Symbols Are Lacking. *arXiv preprint arXiv:2402.11842* (2024).

[67] GNU Operating System. 2024. Coreutils. https://www.gnu.org/software/coreutils/ Accessed: 2024-01-01.

[68] Hanzhuo Tan, Qi Luo, Jing Li, and Yuqun Zhang. 2024. LLM4Decompile: Decompiling Binary Code with Large Language Models. *arXiv preprint arXiv:2403.05286* (2024).

[69] Hugo Touvron, Thibaut Lavril, Gautier Izacard, Xavier Martinet, Marie-Anne Lachaux, Timothée Lacroix, Baptiste Rozière, Naman Goyal, Eric Hambro, Faisal Azhar, et al. 2023. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971* (2023).

[70] Hugo Touvron, Louis Martin, Kevin Stone, Peter Albert, Amjad Almahairi, Yasmine Babaei, Nikolay Bashlykov, Soumya Batra, Prajjwal Bhargava, Shruti Bhosale, et al. 2023. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288* (2023).

[71] Jayakrishna Vadayath, Moritz Eckert, Kyle Zeng, Nicolaas Weideman, Gokulkrishna Praveen Menon, Yanick Fratantonio, Davide Balzarotti, Adam Doupé, Tiffany Bao, Ruoyu Wang, et al. 2022. Arbiter: Bridging the static and dynamic divide in vulnerability discovery on binary programs. In *31st USENIX Security Symposium (USENIX Security 22)*. 413–430.

[72] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N. Gomez, Lukasz Kaiser, and Illia Polosukhin. 2023. Attention Is All You Need. arXiv:1706.03762 [cs.CL]

[73] Daniel Votipka, Seth Rabin, Kristopher Micinski, Jeffrey S Foster, and Michelle L Mazurek. 2020. An observational investigation of reverse {Engineers'} processes. In *29th USENIX Security Symposium (USENIX Security 20)*. 1875–1892.

[74] 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 detection. In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual, South Korea) *(ISSTA 2022)*. Association for Computing Machinery, New York, NY, USA, 1–13. https://doi.org/10.1145/3533767.3534367

[75] Junzhe Wang, Matthew Sharp, Chuxiong Wu, Qiang Zeng, and Lannan Luo. 2023. 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)*. 7339–7356.

[76] Yuting Wang, Xiangzhe Xu, Pierre Wilke, and Zhong Shao. 2020. CompCertELF: verified separate compilation of C programs into ELF object files. *Proceedings of the ACM on Programming Languages* 4, OOPSLA (2020), 1–28.

[77] John Whaley, Dzintars Avots, Michael Carbin, and Monica S Lam. 2005. Using Datalog with binary decision diagrams for program analysis. In *Programming Languages and Systems: Third Asian Symposium, APLAS 2005, Tsukuba, Japan, November 2-5, 2005. Proceedings 3*. Springer, 97–118.

[78] Wikipedia. 2024. Mirai (malware). https://en.wikipedia.org/wiki/Mirai_(malware) Accessed: 2024-01-01.

[79] Yi Wu, Nan Jiang, Hung Viet Pham, Thibaud Lutellier, Jordan Davis, Lin Tan, Petr Babkin, and Sameena Shah. 2023. How Effective Are Neural Networks for Fixing Security Vulnerabilities. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis* (, Seattle, WA, USA,) *(ISSTA 2023)*. Association for Computing Machinery, New York, NY, USA, 1282–1294. https://doi.org/10.1145/3597926.3598135

[80] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. 2023. Automated program repair in the era of large pre-trained language models. In *Proceedings of the 45th International Conference on Software Engineering (ICSE 2023). Association for Computing Machinery*.

[81] Danning Xie, Byungwoo Yoo, Nan Jiang, Mijung Kim, Lin Tan, Xiangyu Zhang, and Judy S Lee. 2023. Impact of Large Language Models on Generating Software Specifications. *arXiv preprint arXiv:2306.03324* (2023).

[82] Xiangzhe Xu, Shiwei Feng, Yapeng Ye, Guangyu Shen, Zian Su, Siyuan Cheng, Guanhong Tao, Qingkai Shi, Zhuo Zhang, and Xiangyu Zhang. 2023. Improving Binary Code Similarity Transformer Models by Semantics-Driven Instruction Deemphasis. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*. 1106–1118.

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

[84] Jianan Yao, Ziqiao Zhou, Weiteng Chen, and Weidong Cui. 2023. Leveraging Large Language Models for Automated Proof Synthesis in Rust. *arXiv preprint arXiv:2311.03739* (2023).

[85] Wei You, Zhuo Zhang, Yonghwi Kwon, Yousra Aafer, Fei Peng, Yu Shi, Carson Harmon, and Xiangyu Zhang. 2020. Pmp: Cost-effective forced execution with probabilistic memory pre-planning. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1121–1138.

[86] Wang Yue, Wang Weishi, Joty Shafiq, and C.H. Hoi Steven. 2021. CodeT5: Identifier-aware Unified Pre-trained Encoder-Decoder Models for Code Understanding and Generation. In *Proceedings of the 2021 Conference on Empirical Methods in Natural Language Processing, EMNLP 2021*.

[87] Jiawei Zhang, Haopeng Zhang, Congying Xia, and Li Sun. 2020. Graph-Bert: Only Attention is Needed for Learning Graph Representations. *arXiv preprint arXiv:2001.05140* (2020).

[88] 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.

[89] 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.