LmPa: Improving Decompilation by Synergy of Large Language Model and Program Analysis

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

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

Zian Su
Purdue University
West Lafayette, USA
su284@purdue.edu

Ziyang Huang
Purdue University
West Lafayette, USA
huan1562@purdue.edu

Shiwei Feng
Purdue University
West Lafayette, USA
feng292@purdue.edu

Yapeng Ye
Purdue University
West Lafayette, USA
ye203@purdue.edu

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

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

Siyuan Cheng
Purdue University
West Lafayette, USA
cheng535@purdue.edu

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

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

ABSTRACT

Decompilation aims to recover the source code form of a binary executable. It has many applications in security and software engineering such as malware analysis, vulnerability detection and code reuse. A prominent challenge in decompilation is to recover variable names. We propose a novel method that leverages the synergy of large language model (LLM) and program analysis. Language models encode rich multi-modal knowledge, but its limited input size prevents providing sufficient global context for name recovery. We propose to divide the task to many LLM queries and use program analysis to correlate and propagate the query results, which in turn improves the performance of LLM by providing additional contextual information. Our results show that 75% of the recovered names are considered good by users and our technique outperforms the state-of-the-art technique by 16.5% and 20.23% in precision and recall, respectively.

1 INTRODUCTION

Decompilation aims to reverse engineer a binary executable, which often has no debugging or symbol information, to a source code form that is close to its original source and human-understandable. During compilation, variables at the source level are transformed to registers and memory locations at the binary level; type information is discarded; statements are broken down to instructions, relocated, and even removed; code structure may be reformed; functions may be inlined; and function boundaries, data and code boundaries are no longer explicit [55, 75]. Decompilation attempts to reverse these transformations, which has various challenges including disassembly [24, 37, 55, 80], variable and type recovery [10, 46, 66, 87], code structure recovery [73], function boundary recovery [67, 83], and name recovery [10, 44].

Decompilation is critical in many security and software engineering tasks. For example, it is often the first step for malware analysis [53, 54, 74], in which human analysts inspect malware code to understand their behaviors. It is important for binary vulnerability analysis where analysts want to identify critical bugs in executables [19, 56], for software supply chain analysis [32, 59], and for code reuse in which legacy executables may need to be ported or hardened [21, 51, 68]. Its importance is evidenced by the popularity of decompilation tools such as IDA [35] and Ghidra [29], e.g., in security threat analysis [8, 13, 14, 53, 54]

There is a large body of existing work on binary reverse engineering and decompilation [11, 11, 12, 31, 31, 49, 66, 73]. The state-of-the-art disassembling methods achieve over 95% precision and recall [55, 65, 67]; type recovery techniques can achieve over 90% precision and recall for primitive types [10] and over 70% for user defined types [10, 87]; and function boundary recognition can achieve 97.1% precision and recall [83].

However, the state-of-the-art *name recovery* method DIRTY [10] only achieves 17.3% precision and 10.9% recall according to our experiment (Section 4.3). Name recovery is arguably one of the most valuable steps in decompilation, because natural language artifacts such as identifier names are crucial for human developers to effectively apprehend a piece of code. Yet, name recovery tends to be more challenging compared to a few other tasks. In addition to the challenges induced by the aforementioned compilation transformations, a large number of intermediate variables are introduced at the binary level and may not have correspondence to any source variable; function names and variable names are application

, ,
.

and context dependent such that machine instructions with few syntactical differences may have substantially different names.

In DIRTY [10], researchers proposed to use language models to infer variable types and names. They trained a transformer model using a large repository of executables and their ground truth symbol information, and then used the model to generate type and name information for a program decompiled by IDA. Note that IDA decompilation focuses on recovering basic control structure and hence largely lacks type or name information. Before DIRTY [10], there were a number of proposals using various deep learning [36, 45, 58] and probabilistic graph model based name generation [31]. More discussions are in the related work section.

DIRTY's performance degrades when the complexity of subject binary increases. This is due to a number of limitations in existing language models. For instance, they only support inputs with a limited size. Hence, DIRTY can only infer names for one function at a time and hardly consider calling contexts. In addition, although the training repository used in DIRTY has 75,656 binaries, it may not be large enough to leverage the true benefits of language models. In comparison, ChatGPT was trained on massive natural language and programming language corpora including Wikipedia, digital books, GitHub, StackOverflow, and so on [60].

In this paper, we develop a novel name recovery technique leveraging the synergy between pre-trained *large language model* (LLM) and program analysis. LLMs are usually trained on enormous datasets with multiple modalities, including source code, binary code, and natural languages. The scale of their training is the key to their impressive performance [5, 9, 61]. We hence propose to build on the success of SOTA pre-trained LLMs to achieve generalizability. In the meantime, existing LLMs still have the aforementioned input size limitation. For example, ChatGPT allows at most 4,096 tokens at a time. We propose to break the procedure of name recovery for a program down to multiple queries to an LLM and use program analysis to chain them together. The procedure is iterative, allowing LLMs to gradually improve over time. Specifically, we develop a name propagation algorithm that has a similar nature to type inference. Assume in one round of query, the LLM is able to derive a meaningful name for some decompiled variable within the queried code snippet (called the *query window*), the name can be propagated to other places in the program outside the query window, following strict program semantics. This allows LLM queries in future rounds to have more contextual information. For example, a newly generated callee function name is propagated to the invocation sites in its callers. To tolerate the non-determinism of LLM-generated names, our analysis abstracts the name of a variable to a set, instead of a singular identifier. After convergence, the distribution in the set naturally informs us the most likely name for the variable.

Our contributions are summarized as follows.

- We propose a novel approach to name recovery for binary executables. It features an iterative algorithm involving using both an LLM and a program analysis.
- We develop a systematic method to construct LLM queries. The method has the capability of including up-to-date information collected from previous rounds of queries.
- We develop a name propagation analysis that can propagate predicted names by the LLM to other places and even construct new meaningful names.
- We devise a post-processing step that filters out meaningless names and selects appropriate names from the analysis results after convergence.
- We have implemented a prototype LmPa. We evaluate it on 1258 functions from 6 popular binary analysis benchmarks. Our user study shows that 75% of the names recovered by LmPa are considered good while the number for DIRTY is 6%. Using an automatic metric based on name similarity, LmPa achieves 33.85% precision and 31.12% recall, substantially outperforming DIRTY, which has 17.31% precision and 10.89% recall. It takes on average 8 LLM queries to name variables in a function. The total fee for our experiments is only 30 USD. Our ablation study shows that if we directly query the LLM without the program analysis, the precision and recall degrade to 31.04% and 18.21%, respectively.

2 MOTIVATION AND OVERVIEW

We use a motivating example to illustrate challenges in decompilation, as well as the limitations of state of the arts. We then present our method.

2.1 Motivating Example

The example is adapted from two functions in Coreutils [16]. The source code of the functions is shown in Fig. 1. Function `c_tolower` (defined at line 1) converts an input character to its lower case. Specifically, it is implemented with a switch-statement (line 2). If the input is an upper case letter, the function converts it to lower case (line 6); otherwise the input character is returned unchanged (line 8). Function `c_strcasecmp()` (defined at line 11) takes as input two strings and compares them in a case-insensitive fashion. It declares four variables (lines 12-14): `p1` and `p2` are pointers to the next characters to be compared in the two input strings, respectively; `c1` and `c2` are two temporary variables holding the compared characters in lower cases. The function iteratively compares each pair of characters (at the same position) in both strings (line 21), and stops at the first difference. It finally returns the difference (line 22). Note that before character comparison, the function calls `c_tolower()` (lines 17-18) to convert both characters to lower cases.

Challenges in Decompilation We compile the example with GCC and the option `-O0` (i.e., no optimization), resulting in a binary program. Then we remove the debugging information and symbol information from the binary, following a typical real-world reverse-engineering scenario [6, 50]. We further use IDA [35] to decompile the binary program, and part of the results are shown in Fig. 2.

During compilation and deployment, the symbol information and high-level code structures in the source code are lost. For example, Fig. 2a shows the decompiled form of `c_tolower()` by IDA [35], and the corresponding assembly instructions are shown in the grey box. We can observe that (1) the function name `c_tolower` and the variable name `c` are not preserved in the binary; (2) the switch-statement in lines 2-9 of Fig. 1 is translated to comparison instructions like line 37 in the grey box of Fig. 2a; (3) the expression `- 'A' + 'a'` (at line 6 in Fig. 1) is simplified to `+ 0x20` (at line

```
1   int c_tolower(int c) {
2     switch (c) {
3     case 'A':
4     // ...
5     case 'Z':
6       return c - 'A' + 'a';
7     default:
8       return c;
9     }
10  }
11  int c_strcasecmp(char *s1, char
        *s2) {
12    char *p1 = s1;
13    char *p2 = s2;
14    char c1, c2;
15    if (p1 == p2) return 0;
16    do {
17      c1 = c_tolower(*p1);
18      c2 = c_tolower(*p2);
19      if (c1 == '\0') break;
20      ++p1; ++p2;
21    } while (c1 == c2);
22    return c1 - c2;
23  }
```

Figure 1: Motivating example

```
1  int sub_4BECE3
2          (uint32_t a1){
3    uint8_t result;
4    if (a1 - 65 > 25)
5      result = a1;
6    else
7      result = a1 + 32;
8    return result;
9  }
```

```
33 0x4BECE3:
34 ;...
35 mov eax, [rbp-4]
36 sub eax, 0x41
37 cmp eax, 0x19
38 ;...
39 add eax, 0x20
3a ret
```

(a) `c_tolower` decompiled by IDA

```
1  int sub_4BED21
2        (uint8_t* a1, uint8_t* a2){
3    uint8_t *v2, *v3;
4    uint8_t v5, v6;
5    // ...
6    do{
7      v6 = sub_4BECE3(*v2);
8      v5 = sub_4BECE3(*v3);
9      if(v6 == 0) break;
10     ++v2;
11     ++v3;
12   }while (v6 == v5);
13   return v6 - v5;
14 }
```

```
21 0x4BED21:
22 ;...
23 mov eax, [r12]
24 mov edi, eax
25 call sub_4BECE3
26 ;...
27 add r12, 1
28 ;...
```

(b) `c_strcasecmp` decompiled by IDA

Figure 2: Decompiled code. (a) and (b) show the decompiled functions generated by IDA. The grey boxes next to them show the corresponding assembly instructions.

Can you help me guess some information for the following decompiled C function from a binary program?
The following is the decompiled C function:

```
int sub_4BECE3
          (uint32_t a1){
  uint8_t result;
  if (a1 - 65 > 25)
    result = a1;
  else
    result = a1 + 32;
  return result;
}
```

In the above function, what are good names for `a1`, `result`, and `sub_4BECE3`, respectively?
You MUST follow the format <original_name> → <new_name>.

Sure, I can help you with that! Here are the names I would suggest:
<a1> → <input_parameter>
<result> → <converted_value>
<sub_4BECE3> → <convert_to_ascii>
Let me know if you have any questions about the names I suggested!

Figure 3: How LmPa interacts with ChatGPT. Blue and green boxes are LmPa's query and ChatGPT's response, respectively.

39 in the grey box of Fig. 2a). Without any symbol or structural information from the original source code, the decompiled code is not similar to the original source, but rather a direct translation of the assembly code, which is difficult to understand. Similarly, Fig. 2b shows the decompiled form of `c_strcasecmp()` and its corresponding assembly code. Note that the variables and callee functions do not have meaningful names. For example, the variable `p1` at line 17 of Fig. 1 is stored in `r12` at line 23 in the grey box of Fig. 2b. IDA thus fills in a dummy name `v2`. Also, the callee function `c_tolower()` is now invoked by its address 0*x*4*BECE*3. The decompiler gives it a dummy name `sub_4BECE3`. Without a meaningful name reflection mechanism, it is hard to understand the decompiled function.

Limitations of State-of-the-Art Methods DIRTY [10] leverages a transformer model to predict types and names of variables in decompiled programs. Although it demonstrates impressive results in type recovery, its name recovery results are limited. In our motivating example, it does not produce any names different from those that are already in the decompiled code. There are two possible reasons. First, limited by the input size of transformer models, DIRTY handles one function at a time and it does not support information sharing across functions. Second, DIRTY assumes binaries still have function names and uses such names in training. Function names provide strong contextual information as the model can learn the typical variable names used in a function with a particular name. For example, in its training data, line 7 in Fig. 2b would be something like `v6 = c_tolower(*v2)`. However in practice, stripped binaries do not have function names. Therefore, the transformer model cannot pick up enough context, and thus can hardly generate meaningful variable names.

Another line of work leverages probabilistic graph models [42] (PGM) to rename decompiled variables with names seen in the training data. We test our motivating example on DEBIN [31], a representative technique in this line. It could not generate desirable names either. For example, in `c_tolower()`, it gives variable `c` a name `index`. PGMs can be considered a more powerful form of Bayesian networks. They model type and name predicates of program artifacts as nodes, e.g., a predicate $isInt(x)$ asserting x is of int type, and edges denote statistical dependences between nodes, which are acquired by program semantics and training. Typing and naming patterns are hence learned and encoded as weight values in the PGMs. However, their results heavily depend on the quality of training data, and PGM inferences are largely local, lacking an ability similar to the attention mechanism in transformers. In our case, the decompiled code body of `c_tolower()` is too simple and does not provide much hint for DEBIN. However, in the caller `c_strcasecmp()`, individual characters in a string are passed to variable `c` in order. Such behavior pattern has been seen and encoded by the PGM, but it was connected with an id of `index`. We show the full function with predicted names from DEBIN in Fig. 23 of our supplementary material.

2.2 Our Technique

Existing techniques suffer from the relatively limited scale of their training. We thus propose a technique that builds on the recent advances in large language models. LLMs [5, 9, 30, 60, 61, 70, 71] are typically trained on multi-modal data of an enormous scale. They demonstrate superior capabilities in many natural language tasks and coding tasks [9, 30, 60, 61]. However, they only allow input of a limited size. Our idea is hence to query an LLM many times, requesting names for separate code snippets of a program, and use program analysis to propagate and filter query results. The process is iterative, meaning that information acquired from past queries are used to provide additional contextual information for future queries, improving the LLM's performance. We use ChatGPT as our underlying LLM, but our technique can be easily generalized to other LLMs (e.g., GPT-4 [62]).

```
int convert_to_ascii(uint32_t input_parameter){
    uint8_t converted_value;
    if (input_parameter - 65 > 25)
        converted_value = input_parameter;
    else
        converted_value = input_parameter + 32;
    return converted_value;
}
```

1

(a) Round 1: Response for c_tolower. Round 1's query is partially shown in Figure 3.

```
1  int compare_strings(uint8_t* string1, uint8_t* string2){
2      uint8_t *string1_pointer, *string2_pointer;
3      uint8_t string2_char, string1_char;
4      // ...
5      do{
6          string1_char = sub_4BECE3(*string1_pointer);
7          string2_char = sub_4BECE3(*string1_pointer);
8          if(string1_char == 0) break;
9          ++string1_pointer;
10         ++string2_pointer;
11     }while (string1_char == string2_char);
12     return string1_char - string2_char;
13 }
```

1

(b) Round 1: Response for c_strcasecmp

```
/* Called in compare_strings
   with input_parameter = *string1_pointer */

int convert_to_lowercase(uint32_t input_char){
  uint8_t converted_char;
  if (input_char - 65 > 25)
      converted_char = input_char;
  else
      converted_char = input_char + 32;
  return converted_char;
}
```

2

(c) Round 2: Response for c_tolower

```
int compare_strings_ignore_case
        (uint8_t* string1, uint8_t* string2){
// ...
string1_char
   = convert_to_lowercase(*string1_pointer);
string2_char
   = convert_to_lowercase(*string1_pointer);
// ...
}
```

3

(d) Round 3: Response for c_strcasecmp

Figure 4: Three rounds of interactions with ChatGPT. Round 1's query is partially shown in Fig. 3. Each box shows a function with symbols renamed according to ChatGPT's response. Red circles show the numbers of rounds. Red arrows and boxes indicate how LmPa uses information from previous response to craft new queries. Names different to previous rounds are highlighted in yellow.

Query to LLM. ChatGPT is an online chat-bot that mimics the dialogue behavior of a human. Its input and output are natural language sentences. To leverage ChatGPT in generating variable names, we have to: (1) formulate the problem into natural language questions; and (2) automatically parse ChatGPT's response and associate the suggested name with the corresponding variable in code. We show in Fig. 3 an example about how LmPa queries ChatGPT to rename function `c_tolower()`. The blue and green boxes are LmPa's query and ChatGPT's response, respectively. At the beginning, LmPa briefly describes the task of predicting names in the decompiled code. Then the decompiled function is attached. After that, LmPa enumerates each variable and specifies the response format requirements. As shown in Fig. 3, ChatGPT follows the format requirements in its response, and thus LmPa can post-process ChatGPT's answer by recognizing the format.

Fig. 4a and Fig. 4b show the two functions in our motivating example, with the variables and functions renamed according to ChatGPT's initial response (using the ChatGPT website between March 6–10). For function `c_tolower()`, we can see that ChatGPT mistakenly considers it as a function converting digits to ASCII code, which shares some common behavior patterns with the target function. The suggested name `input_parameter` for variable `c` is not that informative either. On the other hand, for function `c_strcasecmp()`, ChatGPT produces a close name of `compare_strings`, while missing the case-insensitive part. The predicted variable names in this function are of good quality too (e.g., `string1` for `s1`, `string1_pointer` for `p1`, and `string1_char` for `c1`). We speculate that the good results are due to the sufficient context, namely, the pairwise comparison of array elements (lines 5–11), the comparison with literal number 0 (line 8) to break the loop, and the return value that reports the first difference.

Iterative Name Propagation. To leverage ChatGPT's success in one place to improve its performance in other places such as `c_tolower()`, we further propose a name propagation technique that iteratively propagates names between functions. The key insight is that some functions might be easier for ChatGPT to understand. Information (e.g., variable/function names) derived from these functions can provide better context for other functions. The insight aligns with how a human reverse engineer understands a binary program [6, 50]. She typically starts from functions with special literals or well-known program idioms. The information from these functions will help her understand the other connected parts.

Take Fig. 4c as an example of name propagation. LmPa adds a code comment at the beginning of the queried function. The comment describes how the function is used in its caller. As depicted by the red dashed arrows and the red boxes, LmPa leverages the name of the caller function (i.e., `compare_strings`) and the name of the argument variable (i.e., `string1_pointer`) to compose a comment, propagating the newly acquired contextual information. Readers may be curious why we use comments to propagate information instead of directly setting function and variable names. The reason is that ChatGPT often refuses to generate new names if variables already have non-trivial names in the code. Using comments does not have such restraint. Note that using comments in natural language to convey program analysis results to the chat bot is a unique capability enabled by the underlying LLM.

In Fig. 4c, the changes of ChatGPT's response are highlighted in light yellow. With the additional context, ChatGPT realizes that function `c_tolower()` takes as input a character, and further correctly recognizes the functionality of this function is converting a character to its lower case. Based on the correct functionality, ChatGPT generates a better name (i.e., `input_char`) for variable `c`. Similarly, in the third round shown in Fig. 4d, LmPa conversely propagates the name `convert_to_lowercase()` back to its caller. ChatGPT then generates a more precise name for `c_strcasecmp()` (see part of the function name in yellow). This time, the case insensitive part of the function name is recovered. The example illustrates the power of LLMs, the importance of name propagation, and the gradual improvement through multiple iterations.

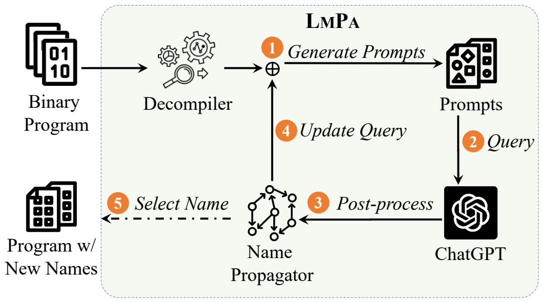


Figure 5: Workflow. In the green boxes are main components of LmPa. The major steps are numbered with orange circles.

⟨Program⟩ P ::= $list\ Decl$ ⟨Identifier⟩ Id ::= $id_0, id_1, id_2 \ldots$

⟨ArgList⟩ $Args$ ::= $list\ Id$ ⟨Operation⟩ $\diamond$ ::= $\{+, -, *, >, \ldots\}$

⟨Declaration⟩ $Decl$ ::= **def** $Id(Args)\{S\}$

⟨Literal⟩ Lit ::= $\{0, 1, 2, \ldots\}$

⟨Expression⟩ E ::= $Id \mid E_1 \diamond E_2 \mid Id(E_1, E_2, \ldots) \mid Lit$

⟨Statement⟩ S ::= $S_1; S_2 \mid E_1 := E_2 \mid$ **return** E

$Conf \in$ Confidence ::= {High, Low}
$Name \in$ Name ::= String
$Pred \in$ Prediction ::= $Name \times Conf$
$NS \in$ NameScheme ::= $(Id_0 \times Id_1) \rightarrow list\ Pred$

Figure 6: Syntax of our language (top) and abstract domains of LmPa (bottom)

3 METHOD

The overall workflow of LmPa is in Fig. 5. It takes as input a binary program, and outputs the decompiled program with recovered names. LmPa first leverages IDA to decompile the input binary program to C code, and then iteratively queries ChatGPT to generate names for functions and variables in the C code. Specifically, after the decompilation, LmPa first generates prompts for each function in the input C program (step 1 in Fig. 5), and then queries ChatGPT with the generated prompts via the ChatGPT API [60], one function at a time (step 2). After LmPa obtains responses from ChatGPT, it parses the natural language outputs and maps the names proposed by ChatGPT back to the C code (step 3). Then a program analysis (*Name Propagator* in Fig. 5) is applied to propagate good names among functions. How to determine if a name is good by its confidence will be discussed later in Section 3.2. The results of propagation are further leveraged to construct the next round queries to ChatGPT (step 4), enabling improvement over time. After convergence, the final results are further processed by selecting the most appropriate names from those that were ever predicted over the multiple rounds (step 5). In the following, we discuss more details.

3.1 Formalization of Problem

This section illustrates how we formulate the problem of name generation for decompiled programs. We first introduce a simple language and the abstract domains (for the program analysis) to facilitate the discussion. Then we show the iterative algorithm LmPa uses to refine variable names.

Language. To simplify the discussion, we use a simple language to model the decompiled C code. Our implementation is based on the Clang-AST parser, and supports most commonly-used C syntax in decompiled functions. The definition of our language is shown in the top part of Fig. 6. A *program* in our language consists of a list of function *declarations*. Each declaration consists of an identifier for the function (Id), a list of arguments ($Args$), and the function body (S). We use *identifier* to refer to the dummy names (e.g., v6) in the decompiled program. Our language has three types of *statements*: $S_1; S_2$ is used to concatenate two statements; $E_1 := E_2$ is the assignment statement; **return** E is used to return values to caller functions. The definitions for *expressions* are standard: Id and Lit are expressions referring to an identifier and a literal, respectively; $E_1 \diamond E_2$ denotes a binary operation over two operand expressions; and $Id(E_1, E_2, \ldots)$ is a function call expression.

Abstract Domains. We show LmPa's abstract domains in the bottom part of Fig. 6. Our program analysis aims to derive information in these domains. LmPa maintains a key-value mapping from an identifier to a list of its candidate names (NS in Fig. 6). Note that the key consists of a pair of identifiers. The first one denotes the function in which the name of the second identifier is predicted. LmPa has a mechanism to force ChatGPT to report its confidence when predicting a new name. Thus each predicted name ($Pred$) has both a confidence($Conf$) and a name ($Name$).

Algorithm. Algorithm 1 shows how LmPa iteratively queries ChatGPT. The enter function of LmPa is defined at line 1. It begins from an empty name scheme (line 2), and adds new names to the name scheme in each iteration (line 4). For each iteration, LmPa first goes over each function and asks ChatGPT to generate name predictions (line 5). Then the propagation rules are applied to each function (line 10) to get better contexts from high-quality names in other functions. Then LmPa updates the query program according to the results of propagation (line 12). Finally, before returning the name scheme to the user, LmPa picks one name for each variable (line 14).

The name propagation sub-procedure (line 10) has termination guarantees, which are determined by the lattice over the set-based abstract domain with set inclusion and the finite universal set. In theory, Algorithm 1 terminates as well if ChatGPT can only generate a finite set of names. In practice, we employ an early termination policy when a round of new queries yields fewer than 10% changes. Another policy is to limit the number of rounds by a query budget. As LLMs' responses are by their nature nondeterministic, ideally we would repeat each query for a few times. However, our study for robustness in Section 4.6 shows that name predictions are stable when ChatGPT is given the same queries, and hence the repetitions are elided in our implementation for query budget savings.

3.2 Interaction with ChatGPT

Both the input and output of ChatGPT are natural language sentences. Thus the key challenge is to formulate the problem of name generation into natural language questions, and to automatically parse ChatGPT's responses. Our solution is to use a prompt template to enumerate each variable we want ChatGPT to predict, and ask ChatGPT to follow a specific output format.

Prompt Generation. As shown in Fig. 3, to query for a function, LmPa first describes the task with a few natural language sentences,

Algorithm 1: Iterative Query and Propagation

```
Function LmPa (program)
    nameScheme = ∅
    budget = N
    while budget ≥ 0 do
        // Ask ChatGPT for each function
        for decl ∈ program do
            // Ask ChatGPT for one function
            predNames = askOneFunc(decl)
            // Update the name scheme with new predictions
            for id, pred ∈ predNames do
                nameScheme[(decl.id, id)].append(pred)
        // Propagate names for each function
        propagationRet = ∅
        for decl ∈ program do
            propagationRet ∪= propagate(decl, program, nameScheme)
        program = updateQuery(program, propagationRet)
        budget = budget - 1
    selected = selectName(nameScheme)
    return selected
```

followed by the decompiled C code. Then LmPa enumerates individual variables in the function and sends the query. We observed that ChatGPT may *miss some variables* when the question is too general, e.g., "What are the good names for all variables in the above function?" If a function has many variables, LmPa groups them in two separate queries to prevent the length from going beyond ChatGPT's token limit. Note that LmPa also asks names for functions.

In addition to names, LmPa guides ChatGPT to report the confidence for each prediction. This is because ChatGPT may generate dummy names (e.g., "`function_input_argument`") or randomly pick irrelevant names when it cannot predict a good name from the context. LmPa prunes out these low-quality names by confidence. Specifically, in prompts, LmPa instructs ChatGPT as follows:

> *You MUST mark your confidence as 'Confident' or 'Not Sure' for each name. If you are confident about a name, you should mark it as 'Confident'. Otherwise, if you are not sure about a name, you should mark it as 'Not Sure'.*

Then LmPa simply filters out all the predictions that are marked as *Not Sure* in post-processing. We observe that ChatGPT may overestimate its confidence for sub-optimal names but rarely underestimate. For example, in our motivating example, ChatGPT marks the (wrongly) predicted name `convert_to_ascii` as *Confident*. LmPa alleviates this problem by considering the name candidate distributions returned by ChatGPT over multiple iterations. Details are in Section B of the supplementary material.

Finally, LmPa requires ChatGPT to output names in a machine-readable format. Without the output format requirements, ChatGPT tends to generate its answers in natural language, or even give a rewritten version of the program.

Post-processing. Although LmPa specifies the output format, ChatGPT still has some variance in its answer. We manually craft a set of regular expressions for LmPa to parse the output, and LmPa will retry the query for one more time if the output format cannot be correctly read. Typically, we observe less than 3% format errors.

3.3 Name Propagation

LmPa's name propagation shares a similar nature as type inference in which known types of some variables are used to derive types for other variables, following program semantics. For example, assume a statement `x=y` and `x` has a known type of int, type inference algorithms can determine `y` also has an int type. In LmPa, good names for a variable are leveraged to derive good names for other variables. Initially, all high confidence names from ChatGPT are considered good names and literals are assigned good names of their own textual forms. A set of rules are used to propagate good names. For instance, a good callee function name will be propagated to its invocation sites in callers. New good names may be constructed for an expression only involving operands with good names. Different from type inference, name propagation is inclusive, meaning that a variable may have multiple good names. Therefore, the propagation is by monotonically deriving more and more relations.

$CallerOf(id_0, id_1)$: Function id_0 is a caller of function id_1.

$GoodNameOf(name_0, id_0, id_1)$: In function id_0, $name_0$ is considered as a good name of id_1.

$GoodNameOf(name_0, id_0, e_1)$: In function id_0, $name_0$ is considered as a good name of expression e_1.

$\mathtt{str}(\diamond|lit)$: Convert an operator or a literal number to string.

Figure 7: Relations and Functions for Name Propagation

Relations and Auxiliary Functions. To facilitate the discussion, we define a few relations and functions in Fig. 7. A good name is represented by a relation. Specifically, $GoodNameOf(name_0, id_0, id_1)$ indicates a string ($name_0$) is considered a good name for an identifier (id_1) in a function (id_0). Similarly, $GoodNameOf(name_0, id_0, e_1)$ indicates $name_0$ is a good name for an expression e_1 in function id_0. $CallerOf(id_0, id_1)$ indicates id_0 is a caller of id_1. LmPa iteratively derives such relations during analysis till it reaches a fixed point. The auxiliary function **`str()`** maps a literal number or an operator to its string representation. □

The analysis is formally defined by a set of inference rules shown in Fig. 8. Each rule is interpreted as follows: the predicates above the line are the premises of a rule; and the formula below the line depicts how new relations are inferred.

Rule *Caller* recognizes caller-callee relations. It means that if a call to function id_2 is found in a statement of id_1, then there is a relation $CallerOf(id_1, id_2)$. That is, id_1 is a caller of id_2. Rules *GN-Id* and *GN-Lit* denote starting points of our inference. *GN-Id* denotes that if in the function id_1, ChatGPT predicts a name n for id with high confidence, then n is considered a good name for id in function id_1. *GN-Lit* specifies the string representations for all literal values are good names. The rationale is that literals (e.g., magic numbers) are important for human reverse engineers [6, 50]. Rule *PropExpr* constructs a good name for an expression $e_1 \diamond e_2$ if both subexpressions have a good name. Note that LmPa similarly constructs good names for other expressions, such as call-expressions and unary operations. Details are elided.

Rules *PropCalleeName* and *PropCalleeArg* are inter-procedural and propagate name information from a callee function to its caller. Specifically, Rule *PropCalleeName* denotes that a good name for the callee is considered a good name for the function invocation in the caller. Rule *PropCalleeArg* represents how LmPa propagates the name for a formal argument in the callee to the corresponding actual argument expression in the caller. For example, if a formal argument is named as `file_descriptor` in the callee function, then

Notations:
$decl_0 = \textbf{def}\ id_0(args_0)\{s_0\}$: Function declaration
ns: Name scheme $\quad$ $prog$: Program $\quad$ lit: Literal

$$\frac{decl_1\ decl_2 \in prog \quad decl_1 = \textbf{def}\ id_1(args_1)\{s_1\} \quad decl_2 = \textbf{def}\ id_2(args_2)\{s_2\}}{id_2(e_1, e_2, \ldots) \in s \wedge s \in s_1 \rightarrow CallerOf(id_1, id_2)}\ \textit{Caller}$$

$$\frac{decl_1 \in prog \quad decl_1 = \textbf{def}\ id_1(args_1)\{s_1\} \quad (id_1, id) \in ns \quad (n, c) \in ns[(id_1, id)]}{c = \mathrm{High} \rightarrow GoodNameOf(n, id_1, id)}\ \textit{GN-Id}$$

$$\frac{decl_1 \in prog \quad decl_1 = \textbf{def}\ id_1(args_1)\{s_1\}}{lit \in s \wedge s \in s_1 \rightarrow GoodNameOf(\mathtt{str}(lit), id_1, lit)}\ \textit{GN-Lit}$$

$$\frac{GoodNameOf(n_1, id_1, e_1) \quad GoodNameOf(n_2, id_1, e_2)}{GoodNameOf(n_1 \mathtt{str}(\diamond) n_2, id_1, e_1 \diamond e_2)}\ \textit{PropExpr}$$

$$\frac{CallerOf(id_0, id_1) \quad GoodNameOf(n, id_1, id_1)}{GoodNameOf(n, id_0, id_1)}\ \textit{PropCalleeName}$$

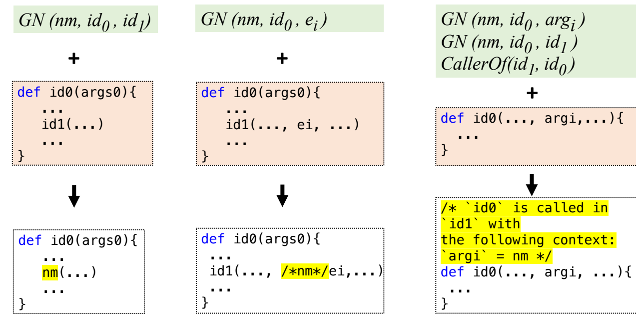
$$\frac{id_1(e_1, e_2, \ldots) \in s \quad s \in s_0 \quad CallerOf(id_0, id_1) \quad decl_0 = \textbf{def}\ id_0(args_0)\{s_0\} \quad decl_1 = \textbf{def}\ id_1(args_1)\{s_1\}}{GoodNameOf(n, id_1, args_1[i]) \rightarrow GoodNameOf(n, id_0, e_i)}\ \textit{PropCalleeArg}$$

Figure 8: Propagation Rules

the expression corresponding to that argument at the invocation site may also be a file descriptor. The set of rules for propagation from a caller function to all its callees are symmetric and hence elided.

3.4 Query Update

After name propagation, LmPa further leverages the propagated names to construct the next round queries. The query update algorithm takes as input a query text of a function and the *GoodNameOf* relations derived by the propagation rules, and outputs a new query for the function. A few query construction rules are presented in Fig. 9. The green boxes show the derived *GoodNameOf* relations, and the tan boxes show the function. In Fig. 9a, LmPa derives a good name for the callee function id_1 in the context of function id_0. It renames all the invocations to id_1 to the good name. Note that there may be multiple good names for a function/variable, LmPa selects the one with the latest timestamp. Fig. 9b shows how to leverage good name information regarding an expression, including a singleton variable expression. Recall that our name propagation allows generating names for composite expressions. We cannot simply rename any identifier to utilize such information. LmPa thus propagates the information by code comments. As shown in Fig. 9b, it puts the propagated *name* in the code comment before the expression e_i. Note that even if the related expression is a singleton variable, simply replacing its identifier with a good name may yield undesirable results. The reason is that ChatGPT tends not to rename a variable that already has a meaningful name in the code. Thus directly setting variable names in the code prevents ChatGPT from generating any new names. Fig. 9c shows that when a caller function and its actual argument expression have good names, they can be utilized in the query of a callee of the function. Specifically, a new comment is added before the callee function describing which caller function may call it and the good name for the argument expression.



(a) Update Callee Name **(b) Update Arg Name from Callee** **(c) Update Context from Caller**

Figure 9: How LmPa composes new queries. LmPa rewrites a query based on both the *GoodNameOf* relation(s) (green box, noted as *GN*) and function code body (tan box). The white boxes on the bottom show the new queries with the modification highlighted in yellow.

4 EVALUATION

We develop LmPa on top of IDA Pro 7.5 and Clang 12. LmPa consists of a total of 2,770 lines of Python code and 3,214 lines of C++ code.

We examine the effectiveness of LmPa by addressing the following research questions (RQs):

RQ1: Can LmPa effectively help developers comprehend decompiled code? How does it compare with the SOTA?

RQ2: How well do names generated by LmPa and SOTA match their original versions in the source code?

RQ3: What are the impacts of the name propagation analysis on the overall performance of LmPa?

RQ4: Does LmPa scale well on real-world data?

RQ5: Is LmPa resilient to the nondeterminism of LLM answers?

In addition to these RQs, we conduct four case studies to illustrate how LmPa helps in the real-world use scenarios.

4.1 Setup

Benchmark. We assess LmPa using six well-established real-world projects that have been extensively employed in previous studies [23, 34, 41, 51, 68]. It is worth noting that OpenAI enforces various resource restrictions when accessing ChatGPT [60], such as query fees and intentional delays (e.g., around 20 seconds per query). Our dataset consists of 16,212 functions in total. However, evaluating all the functions from our dataset would lead to high resource consumption. Therefore, we adhere to existing practices [10, 68, 78] and randomly sample a subset of 1,258 functions consisting of 4,277 variables. Detailed statistics of our dataset can be found in Section C of our supplementary material.

Evaluation Metrics. Assessing the degree of alignment between predicted names and ground-truth names (i.e., the variable names in the source code) presents a significant challenge because there may be many semantically equivalent names for a variable. For instance, `buffer_size` and `buffer_n` are often deemed semantically equivalent in the context of programming, yet they do not match each other. To address the issue, we propose the following two metrics for evaluation.

Developer Preferences. Taking into account the complexity in evaluating the semantic equivalence of symbol names, incorporating professional developers in the evaluation process is a judicious approach. To this end, we conduct a user study with a group of developers, including a number of participants with substantial reverse engineering experience. Each participant was presented with several functions, accompanied by their source code and ground-truth names. The participants were then asked to score each predicted name on a scale of 1 to 5, with higher scores reflecting better predictions. A more detailed description can be found in Section 4.2.

Name Similarity. While user studies can provide reliable results, they are inherently difficult to scale up. In order to automate the evaluation process, we introduce a similarity score function that quantifies the similarity between a predicted name and its corresponding ground-truth name.

$$Similarity(S_{TP},\ S_P) = \frac{|LCS(S_{TP},\ S_P)|}{|S_{TP}|}$$

In the formula above, S_{TP} and S_P represent the ground-truth and predicted names, respectively. LCS represents the longest common subsequence between the two input strings. Essentially, this function assesses the proportion of characters in the ground-truth name that are accurately predicted in order. For example, it yields similarity scores of 0.64 and 0.6 for the aforementioned `buffer_size` and `ret_buffer` examples, respectively. It is important to note that the similarity function generates a score rather than a binary outcome, providing a more refined evaluation of the predictions. More importantly, outcomes derived from our user study align well with results by this automated method, as detailed in Section 4.2. This provides additional support for its validity in practice.

4.2 RQ1: User Study

To evaluate the effectiveness of LmPa, we conduct a sizeable user study. Specifically, we randomly select 30 functions from our dataset, and all variables present in the sampled functions are examined as subjects within the study. To help participants understand the context, each function is accompanied by its respective source code and decompiled code.

We task the participants with evaluating the quality of predicted names by comparing them to their ground-truth counterparts. The study encompasses four variable name prediction methods: DEBIN, DIRTY, ChatGPT without the propagation mechanism (one-shot), and LmPa. Participants are instructed to rate each predicted name on a scale of 1 to 5, with the scores indicating (1) *misleading*, (2) *meaningless*, (3) *general but missing context*, (4) *acceptable*, and (5) *comparable to or better than ground truth*. We include concrete samples of the study in Section A in the supplementary material.

In addition to the randomly-sampled 30 functions, we mix in the study another 8 functions with 33 variables as validation samples. In each validation sample, one of the four methods demonstrates a clear advantage over the others. These samples are used to ascertain participants' attentiveness during the study. It should be noted that results from validation questions are excluded from our final analysis. In total, we construct 528 questions, consisting of 396 testing questions and 132 validation questions.

Table 1: Comparison between LmPa and DIRTY. Column "Total" is the number of variables in our samples in each dataset. Column "Overlap" shows the portion of variables that overlap with the training set of DIRTY.

Dataset	Total	LmPa		DIRTY		
		Precision	Recall	Precision	Recall	Overlap
Coreutils	820	30.86%	27.93%	11.61%	8.17%	16.95%
Findutils	743	32.33%	30.42%	15.60%	11.84%	14.00%
SQLite	563	27.77%	26.29%	15.08%	4.80%	3.20%
ImageMagick	488	48.73%	43.24%	11.96%	6.76%	0.00%
Diffutils	727	33.28%	31.09%	13.45%	9.49%	14.86%
Binutils	886	30.15%	27.77%	36.13%	24.27%	61.40%
Average	-	**33.85%**	**31.12%**	**17.31%**	**10.89%**	-

We recruit 31 participants, with 16 from our institution, and the rest from three world-class CTF (Capture The Flag) teams[1]. All participants have extensive programming experience, with 26 of them having utilized C/C++ in project development and 10 possessing over three years of hands-on expertise in reverse engineering. We ensure that at least four participants respond to each question.

Overall Results. Fig. 10 delineates the results of the user study, with the x-axis representing user scores and the y-axis indicating the count of predicted names corresponding to each score. It is clear that LmPa surpasses the other three methods, as the majority of its predicted names achieve scores of 4 and 5, i.e., "good names", indicating that LmPa is good at providing semantically meaningful names. ChatGPT without propagation also exhibits a relatively commendable performance compared to the baselines. However, due to the lack of a propagation mechanism and the inability to aggregate derived information, it yields fewer good names. Specifically, LmPa generates good names for 75% variables, and ChatGPT without propagation generates good names for 45%. The two baseline methods DIRTY and DEBIN generate good names for 6% and 5% variables, respectively. Note that the majority of DIRTY's predictions are scored 2 (i.e., meaningless names), and none of them obtain a score of 1 (i.e., misleading names). This can be attributed to DIRTY's conservative nature, which tends to generate dummy names such as `a1`.

Effectiveness of the Name Similarity Metric. Piggybacking on this experiment, we validate the effectiveness of the proposed automated metric. Specifically, for each predicted name, we calculate its similarity to the corresponding ground-truth name and compare the similarity score with the score by users. Fig. 11 presents the results. The x-axis represents a name similarity score threshold. The y-axis indicates the average user study scores of the predicted names whose similarity scores exceed the threshold. Observe that the average user study score has a close-to-linear positive relation with the threshold. It validates that the similarity score serves as a reasonable approximation of semantic equivalence of variable names from a user standpoint. Observe that when the threshold is 0.0, the user score is still slightly above 3. It essentially indicates that the average user study score of all predicted variables (generated by the four subject methods) is marginally above 3.

[1]CTFs are renowned competitions designed to challenge participants in solving computer security problems, including reverse engineering tasks. In order to determine the world-class standing of a CTF team, we assess whether they have achieved a top-10 ranking on CTFTime [17] at least once during the period spanning from 2013 to 2023.

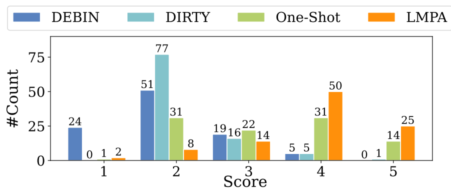


Figure 10: The study comprises 99 variables, each rated by four users and averaged.

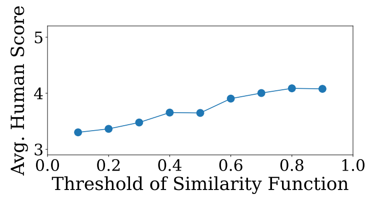


Figure 11: Relation between similarity function and human scores.

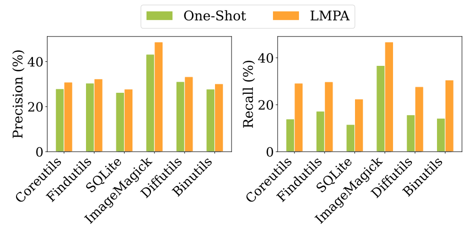


Figure 12: Comparison with one-shot.

4.3 RQ2: Quality of Predicted Names

To assess the degree to which the names generated by LmPa correspond with their ground-truth counterparts, we employ the similarity function to gauge the prediction quality, thereby enabling the evaluation to be scaled across the entire benchmark. Fig. 11 shows that when the threshold is set at 0.6, the average human score exceeds 4, indicating that the predicted names are acceptable alternatives as rated by users. Consequently, we select a threshold of 0.6 for the similarity metric, meaning that a predicted name is deemed a "good name" if its similarity score surpasses 0.6. A good prediction is treated as a *true positive*, based on which we can further calculate the *precision* and *recall* of a name prediction technique [10].

Overall Results. Table 1 shows the performance of LmPa in comparison to DIRTY, the current state-of-the-art technique for predicting variable names in decompiled code. Note that DIRTY assumes the decompiled program has the ground-truth function names. *We thus provide the names of functions (only) in DIRTY's test samples.* The results of LmPa are obtained on programs without ground-truth function names. Although the setup for LmPa is more challenging, LmPa outperforms DIRTY on most datasets in terms of both precision and recall. We attribute this to the advances of LLM, and the name propagation technique that provides more context for the queries to LLM. LmPa achieves the highest improvement on the ImageMagick dataset, with a precision that is over four times that of DIRTY's and a recall over six times. Further analysis attributes the relatively higher performance to ImageMagick's heavy reliance on external library function calls, which supply an abundance of hints to the LLM. On the Binutils dataset, DIRTY slightly outperforms LmPa in terms of precision. That is because more than 60% of the variables in that dataset overlap with DIRTY's training set (see the last column of Table 1), while such overlap is lower than 17% in other benchmarks. Note that DIRTY was trained on functions randomly sampled from Github, and thus their training data may overlap with some functions in our test sets. On the other hand, LmPa still outperforms DIRTY in terms of recall. That is because LmPa propagates program context across functions, while DIRTY makes prediction on the local context of a function. The authors of DIRTY reported better precision and recall in their paper. The reason is that they used many small projects from Github, in which variable names tend to have stronger connections with the provided ground-truth function names. In comparison, the benchmarks used in our evaluation are more complex than 80% of those used in DIRTY.

Discussion. Observe that the precision and recall of LmPa are not as remarkable as one would hope. However, it does not necessarily mean that LmPa cannot provide informative names. Based on our observations, the similarity metric used is relatively strict. Even if the predicted names are semantically equivalent to the ground-truth names, they may not receive a high similarity score. For example, `params` and `options` are semantically equivalent, but they have a very low similarity score. Our user study indeed indicates that over 75% of the predicted names are considered good by users. An ideal solution would be to precisely measure semantic distance of two names. However, the substantial variations in naming conventions make the development of such a method very challenging. We will leave it to our future work. We further conduct a case study to show a typical failure of LmPa in Section 4.7.

Assessment with Various Thresholds. We further compare the performance of LmPa and DIRTY with different thresholds for a "good name". The result shows that LmPa outperforms DIRTY across the entire spectrum of threshold levels. Details can be found in Section D of our supplementary material.

4.4 RQ3: Ablation Study

To better understand the effects of the name propagation analysis, we conduct three ablation studies. The first study compares the performance of LmPa with that by asking ChatGPT for one-shot. The second study compares LmPa with a naive approach that simply appends callee functions of the query function in the query text. The last ablation study shows how LmPa gradually achieves better performance as the iteration of propagation grows.

Comparison with One-shot ChatGPT Queries. Fig. 12 presents a comparison between LmPa and one-shot ChatGPT queries, with the left figure illustrating precision and the right figure depicting recall. Notably, LmPa achieves a slightly superior, yet generally comparable precision in relation to one-shot ChatGPT queries. Upon closer examination, we find that, for a given variable, when ChatGPT lacks sufficient information to predict an appropriate name, it tends to generate a "dummy name". These names are subsequently eliminated through the name selection process. Consequently, only variables with adequate contextual information receive predicted names. As such, the precision primarily assesses ChatGPT's capability of predicting names for variables already rich in contextual information and is not directly related to the presence of the propagation mechanism. Nevertheless, LmPa significantly outperforms one-shot ChatGPT queries in terms of recall, achieving approximately twice the performance in most cases. This can be attributed to the effective propagation mechanism.

Comparison with a Naïve Algorithm. We conduct a study to show that LmPa substantially outperforms a method that includes callee functions in ChatGPT queries. Details are in Section E in the supplementary material.

Impact of the Number of Propagation Iterations. We observe performance improvement is substantial in the first a few rounds of analysis and 10 rounds deliver optimal results. Details are in Section F in the supplementary material.

4.5 RQ4: Scalability

On average, querying ChatGPT for one time takes 22.8 seconds, leading to a relatively high time consumption for LmPa. However, the queries can be easily parallelized and the cost are justifiable in practice, given the one-time nature of reverse engineering efforts. Furthermore, LmPa scales well to large programs which in fact provides more context. Details can be found in Section G in the supplementary material.

4.6 RQ5: Robustness

Due to the nondeterministic nature of LLM, we repeat an experiment on the Coreutils dataset for 8 times to illustrate the robustness of LmPa. In each run, we let LmPa propagate names for 4 iterations. The results show that LmPa has a stable performance among different runs, with less than 0.04% variations and the improvement from round to round is significantly larger than the variance. Details can be found in Fig. 22 in the supplementary material.

4.7 Case Studies

Performance on Unseen Programs. ChatGPT is trained on enormous data. It is unclear whether our benchmarks have been used in ChatGPT's training. To study LmPa's performance on unseen programs, we conduct a case study on AudioFlux [48], an audio processing library project started in 2023. The results show that LmPa is equally effective whereas the baselines have lower than 5% precision and recall. Details are in Section H in the supplementary material.

Failure Case of LmPa. We examine a failure case of LmPa, which received a score of 1 in our user study. Figure 13 presents the source code for this case, which is simplified for illustrative purposes. The code represents a wrapper function for `memcmp`, in which `buf1` and `buf2` are input memory buffers, while `size1` and `size2` denote the respective buffer sizes. The variable `tmp_size2` is a copy of `size2`. The code utilizes `tmp_size2` to store the value of `size2` that will be modified later. Although LmPa accurately predicts the name of `size2`, it erroneously assigns the name `size_diff` to `tmp_size2`.

One might wonder why `tmp_size2 = size2` (line 3) does not help resolve the issue, given the name propagation analysis. Recall that, unlike inter-procedural hints, LmPa does not employ code comments to explicitly propagate intra-procedural hints. Instead, we rely on LLM itself to detect the potential relations among variables within the same function, avoiding the submission of lengthy queries to the LLM that might end up confusing the model. In this case, ChatGPT does not correctly determine the relation between `size2` and `tmp_size2` and our propagation does not help either. This issue could be tackled either by devising more sophisticated propagation rules for intra-procedural hints or by adopting a more advanced LLM. In fact, we assessed the failure case utilizing a variant of LmPa built upon GPT-4 [62]. The GPT-4-based LmPa successfully determines the desired relation and predicts `tmp_size2` as `size2_copy`, supporting our hypothesis that LmPa's performance exhibits a positive correlation with LLM quality.

```
1 int memcmp2(void *buf1, size_t size1,
2             void *buf2, size_t size2) {
3   size_t tmp_size2 = size_2;
4   cmp = memcmp(buf1, buf2, min(size1, size2));
5   ...
6 }
```

Figure 13: A failure case of LmPa

Query with Program Functionality Description. To simulate realistic application in which analysts roughly know a program's functionalities, we provide a textual description of a program at the beginning of LmPa's query prompts and find that LmPa's performance improves. Details are in Section I in the supplementary material.

Query to GPT-4. We substitute ChatGPT with GPT-4 to investigate the impact of a more advanced LLM on LmPa's performance. Due to GPT-4's slower processing speed compared with ChatGPT [63], we randomly sample a smaller dataset from Coreutils, comprising 140 variables, and evaluate both GPT-4-driven LmPa and ChatGPT-driven LmPa on this dataset. The results are presented in Table 2. Observe that LmPa demonstrates better performance when powered by GPT-4. Specifically, with a propagation iteration count of 4, the GPT-4-driven LmPa achieves over 13% higher precision compared with its ChatGPT-driven counterpart. We attribute this to the superior capability of GPT-4. Note that precision essentially measures the LLM's performance when making confident predictions (see Section 4.3). Thus a stronger LLM leads to better performance of LmPa. Additionally, the GPT-4-driven version achieves better recall than the ChatGPT-driven one. We attribute the improvement to GPT-4's better capability of inferring good names based on local information, rendering the overall contextual information propagation more effective. It is also noteworthy that, for both GPT-4-driven LmPa and ChatGPT-driven LmPa, the propagation algorithm leads to improved results. This highlights the necessity of our name propagation analysis, regardless of the underlying LLM employed. Such results indicate that the performance of LmPa can be further enhanced as more powerful LLMs become available, while the propagation analysis continues to play an essential role in achieving optimal results.

5 THREATS TO VALIDITY

We choose to use ChatGPT, a closed-source LLM. The reported results may hence be tied to a specific version of ChatGPT. We have logged the interactions with ChatGPT for reproducibility. In addition, our technique is independent of the LLM. Our case study shows that the performance of LmPa has a positive correlation with LLM quality, which is supposed to improve over time. LLMs including ChatGPT are trained on enormous multi-modal data. It is unclear if the benchmarks used in the paper had been used in ChatGPT's training. This is a general threat-to-validity to any research using LLMs. On one hand, we compile the benchmarks and generate fresh binaries, which likely differ from the binaries used in LLM training. On the other hand, we argue that LLMs are so general that they unlikely overfit-on/memorize specific training examples. In addition, our ablation study on a very recent project (unlikely seen by ChatGPT) shows that LmPa is equally effective, whereas the baseline has substantially degraded performance. LLMs' responses are nondeterministic in general. Our ablation study shows that name predication by ChatGPT yields largely stable results.

Table 2: Performance on GPT-4. LmPa $_{GPT4}$ **and** LmPa $_{CHAT}$ **denote** LmPa **with GPT-4 and ChatGPT, respectively.**

Tool	Iteration: 0		Iteration: 4	
	Precision	Recall	Precision	Recall
LmPa $_{GPT4}$	50.00%	28.57%	60.95%	45.71%
LmPa $_{CHAT}$	41.25%	23.57%	47.22%	36.43%

Our user study is susceptible to human errors. To mitigate the threat, we have carefully planned the study, using validation tests as part of the study and choosing programmers with extensive experience (e.g., in reverse engineering), and having multiple users covering a test.

6 RELATED WORK

Binary Analysis. Binary analysis is of fundamental importance in the field of software security and software engineering, encompassing a range of critical downstream applications such as malware analysis [4, 7, 25, 28, 39, 40, 81], vulnerability detection [23, 79], software fingerprinting [15], APT attack forensics [2, 22, 43, 69, 77, 82], and software reuse [21, 68]. LmPa is intrinsically connected to decompilation [27, 33, 47], a foundational task in binary analysis. In addition to the related works discussed in Section 2.1, substantial research has been conducted in the area of decompilation, addressing topics such as type inference [11, 31, 49, 66], binary-level data-flow analysis [3, 88], function signature inference [11, 12, 31, 66], and binary similarity [20, 52, 68, 84, 86, 89]. Our work is orthogonal to these existing contributions.

Large Language Models. Large Language Models (LLMs) have made significant breakthroughs in language understanding and generative tasks, including language translation [18, 72], text summarization [5, 70–72], question answering [5, 18, 72], and so on. LLMs developed for programming languages [1, 9, 26, 57, 85] have also shown their capabilities in software engineering tasks, such as code translation [9, 76, 85], code completion [9, 26, 57], and program repair [1, 38, 85]. In this paper, we are the first to explore the potential of LLMs, especially ChatGPT, for name recovery, and demonstrate through extensive evaluation that they can significantly improve performance on this important task.

7 CONCLUSION

We develop a novel technique for symbol name recovery in decompilation. It leverages the synergy between large language models and program analysis. It features an iterative algorithm that propagates query results from ChatGPT following program semantics. The propagation in turn provides better context for ChatGPT. Our results show that 75% of the recovered names are considered good by users and our technique outperforms the state-of-the-art technique by 16.5% and 20.23% in precision and recall, respectively.

REFERENCES

[1] 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

[2] Abdulellah Alsaheel, Yuhong Nan, Shiqing Ma, Le Yu, Gregory Walkup, Z Berkay Celik, Xiangyu Zhang, and Dongyan Xu. 2021. ATLAS: A Sequence-based Learning Approach for Attack Investigation. In *USENIX Security*.

[3] Gogul Balakrishnan and Thomas Reps. 2004. Analyzing Memory Accesses in x86 Executables. In *Compiler Construction*, Evelyn Duesterwald (Ed.). Springer Berlin Heidelberg, Berlin, Heidelberg, 5–23.

[4] Leyla Bilge, Davide Balzarotti, William Robertson, Engin Kirda, and Christopher Kruegel. 2012. Disclosure: detecting botnet command and control servers through large-scale netflow analysis. In *Proceedings of the 28th Annual Computer Security Applications Conference*. ACM.

[5] Tom B. Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, Sandhini Agarwal, Ariel Herbert-Voss, Gretchen Krueger, Tom Henighan, Rewon Child, Aditya Ramesh, Daniel M. Ziegler, Jeffrey Wu, Clemens Winter, Christopher Hesse, Mark Chen, Eric Sigler, Mateusz Litwin, Scott Gray, Benjamin Chess, Jack Clark, Christopher Berner, Sam McCandlish, Alec Radford, Ilya Sutskever, and Dario Amodei. 2020. Language Models are Few-Shot Learners. *CoRR* abs/2005.14165 (2020). arXiv:2005.14165 https://arxiv.org/abs/2005.14165

[6] Kevin Burk, Fabio Pagani, Christopher Kruegel, and Giovanni Vigna. 2022. Decomperson: How Humans Decompile and What We Can Learn From It. In *31st USENIX Security Symposium (USENIX Security 22)*. 2765–2782.

[7] Mahinthan Chandramohan, Yinxing Xue, Zhengzi Xu, Yang Liu, Chia Yuan Cho, and Hee Beng Kuan Tan. 2016. BinGo: Cross-Architecture Cross-OS Binary Search *(FSE 2016)*. Association for Computing Machinery, New York, NY, USA, 678–689. https://doi.org/10.1145/2950290.2950350

[8] Check Point Research 2022. *TWISTED PANDA: CHINESE APT ESPIONAGE OPERATION AGAINST RUSSIAN STATE-OWNED DEFENSE INSTITUTES.* https://research.checkpoint.com/2022/twisted-panda-chinese-apt-espionage-operation-against-russians-state owned-defense-institutes/

[9] Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Ponde de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating Large Language Models Trained on Code. *CoRR* abs/2107.03374 (2021). arXiv:2107.03374 https://arxiv.org/abs/2107.03374

[10] 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)*. USENIX Association, Boston, MA, 4327–4343. https://www.usenix.org/conference/usenixsecurity22/presentation/chen-qibin

[11] Qibin Chen, Jeremy Lacomis, Edward J Schwartz, Claire Le Goues, Graham Neubig, and Bogdan Vasilescu. 2021. Augmenting Decompiler Output with Learned Variable Names and Types. *arXiv preprint arXiv:2108.06363* (2021).

[12] Zheng Leong Chua, Shiqi Shen, Prateek Saxena, and Zhenkai Liang. 2017. Neural nets can learn function type signatures from binaries. In *26th USENIX Security Symposium (USENIX Security 17)*.

[13] CISA 2020. *MAR-10292089-1.v2 – Chinese Remote Access Trojan: TAIDOOR.* https://www.cisa.gov/news-events/analysis-reports/ar20-216a

[14] CISA-TEARDROP 2021. *MAR-10320115-1.v1 - TEARDROP.* https://www.cisa.gov/news-events/analysis-reports/ar21-039b

[15] Christian Collberg and Clark Thomborson. 1999. Software watermarking: Models and dynamic embeddings. In *Proceedings of the 26th ACM SIGPLAN-SIGACT symposium on Principles of programming languages*. 311–324.

[16] Coreutils 2022. *Coreutils - GNU core utilities.* https://www.gnu.org/software/coreutils/

[17] CTFTime 2023. *All about CTF (Capture The Flag).* https://ctftime.org/

[18] Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of Deep Bidirectional Transformers for Language Understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*. Association for Computational Linguistics, Minneapolis, Minnesota, 4171–4186. https://doi.org/10.18653/v1/N19-1423

[19] Sushant Dinesh, Nathan Burow, Dongyan Xu, and Mathias Payer. 2020. Retrowrite: Statically instrumenting cots binaries for fuzzing and sanitization. In *2020 IEEE Symposium on Security and Privacy (SP)*. IEEE, 1497–1511.

[20] 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 *SP*. IEEE.

[21] Steven H. H. Ding, Benjamin C. M. 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)*. 472–489. https://doi.org/10.1109/SP.2019.00003

[22] Cho Do Xuan, Mai Hoang Dao, and Hoa Dinh Nguyen. 2020. APT attack detection based on flow network analysis techniques using deep learning. *Journal of Intelligent & Fuzzy Systems* 39, 3 (2020).

[23] Yue Duan, Xuezixiang Li, Jinghan Wang, and Heng Yin. 2020. DeepBinDiff: Learning Program-Wide Code Representations for Binary Diffing. https://doi.org/10.14722/ndss.2020.24311

[24] Antonio Flores-Montoya and Eric Schulte. 2020. Datalog Disassembly. In *Proceedings of the 29th USENIX Conference on Security Symposium (SEC'20)*. USENIX Association, USA, Article 61, 18 pages.

[25] Yanick Fratantonio, Antonio Bianchi, William Robertson, Engin Kirda, Christopher Kruegel, and Giovanni Vigna. 2016. Triggerscope: Towards detecting logic bombs in android applications. In *SP*. IEEE.

[26] Daniel Fried, Armen Aghajanyan, Jessy Lin, Sida Wang, Eric Wallace, Freda Shi, Ruiqi Zhong, Wen-tau Yih, Luke Zettlemoyer, and Mike Lewis. 2022. InCoder: A Generative Model for Code Infilling and Synthesis. https://doi.org/10.48550/ARXIV.2204.05999

[27] Cheng Fu, Huili Chen, Haolan Liu, Xinyun Chen, Yuandong Tian, Farinaz Koushanfar, and Jishen Zhao. 2019. Coda: An end-to-end neural program decompiler. *Advances in Neural Information Processing Systems* 32 (2019).

[28] Jian Gao, Xin Yang, Ying Fu, Yu Jiang, and Jiaguang Sun. 2018. *VulSeeker: A Semantic Learning Based Vulnerability Seeker for Cross-Platform Binary*. Association for Computing Machinery, New York, NY, USA, 896–899. https://doi.org/10.1145/3238147.3240480

[29] Ghidra 2022. *A software reverse engineering (SRE) suite of tools developed by NSA's Research Directorate in support of the Cybersecurity mission*. https://ghidra-sre.org

[30] GitHub. 2021. GitHub Copilot. https://copilot.github.com/. Accessed: March 29, 2023.

[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*.

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

[33] Iman Hosseini and Brendan Dolan-Gavitt. 2022. Beyond the C: Retargetable Decompilation using Neural Machine Translation. In *Proceedings 2022 Workshop on Binary Analysis Research*. Internet Society. https://doi.org/10.14722/bar.2022.23009

[34] Yikun Hu, Hui Wang, Yuanyuan Zhang, Bodong Li, and Dawu Gu. 2021. A Semantics-Based Hybrid Approach on Binary Code Similarity Comparison. *IEEE Transactions on Software Engineering (TSE)* 47, 6 (June 2021), 1241–1258. https://doi.org/10.1109/TSE.2019.2918326

[35] IDA Pro 2022. *A powerful disassembler and a versatile debugger*. https://hex-rays.com/ida-pro/

[36] Alan Jaffe, Jeremy Lacomis, Edward J Schwartz, Claire Le Goues, and Bogdan Vasilescu. 2018. Meaningful variable names for decompiled code: A machine translation approach. In *Proceedings of the 26th Conference on Program Comprehension*. 20–30.

[37] Muhui Jiang, Yajin Zhou, Xiapu Luo, Ruoyu Wang, Yang Liu, and Kui Ren. 2020. An Empirical Study on ARM Disassembly Tools. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual Event, USA) *(ISSTA 2020)*. Association for Computing Machinery, New York, NY, USA, 401–414. https://doi.org/10.1145/3395363.3397377

[38] Nan Jiang, 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)*.

[39] Alexandros Kapravelos, Chris Grier, Neha Chachra, Christopher Kruegel, Giovanni Vigna, and Vern Paxson. 2014. Hulk: Eliciting malicious behavior in browser extensions. In *23rd USENIX Security Symposium (USENIX Security 14)*.

[40] Amin Kharraz, William Robertson, Davide Balzarotti, Leyla Bilge, and Engin Kirda. 2015. Cutting the gordian knot: A look under the hood of ransomware attacks. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer.

[41] 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* (2022), 1–23. https://doi.org/10.1109/TSE.2022.3187689

[42] Daphne Koller and Nir Friedman. 2009. *Probabilistic graphical models: principles and techniques*. MIT press.

[43] Nickolaos Koroniotis, Nour Moustafa, and Elena Sitnikova. 2019. Forensics and deep learning mechanisms for botnets in internet of things: A survey of challenges and solutions. *IEEE Access* 7 (2019).

[44] 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.

[45] Jeremy Lacomis, Pengcheng Yin, Edward J. Schwartz, Miltiadis Allamanis, Claire Le Goues, Graham Neubig, and Bogdan Vasilescu. 2019. DIRE: A Neural Approach to Decompiled Identifier Naming. *CoRR* abs/1909.09029 (2019). arXiv:1909.09029 http://arxiv.org/abs/1909.09029

[46] JongHyup Lee, Thanassis Avgerinos, and David Brumley. 2011. TIE: Principled reverse engineering of types in binary programs. (2011).

[47] Ruigang Liang, Ying Cao, Peiwei Hu, Jinwen He, and Kai Chen. 2021. Semantics-recovering decompilation through neural machine translation. *arXiv preprint arXiv:2112.15491* (2021).

[48] libAudioFlu. 2023. . https://github.com/libAudioFlux/audioFlux

[49] Alwin Maier, Hugo Gascon, Christian Wressnegger, and Konrad Rieck. 2019. TypeMiner: Recovering types in binary programs using machine learning. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer.

[50] 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.

[51] Andrea Marcelli, Mariano Graziano, Xabier Ugarte-Pedrero, Yanick Fratantonio, Mohamad Mansouri, and Davide Balzarotti. 2022. How machine learning is solving the binary function similarity problem. In *31st USENIX Security Symposium (USENIX Security 22)*. 2099–2116.

[52] Luca Massarelli, Giuseppe Antonio Di Luna, Fabio Petroni, Roberto Baldoni, and Leonardo Querzoni. 2019. Safe: Self-attentive function embeddings for binary similarity. In *International Conference on Detection of Intrusions and Malware, and Vulnerability Assessment*. Springer.

[53] Microsoft 2021 2021. *GoldMax, GoldFinder, and Sibot: Analyzing NOBELIUM's layered persistence*. https://www.microsoft.com/en-us/security/blog/2021/03/04/goldmax-goldfinder-sibot-analyzing-nobelium-malware/

[54] Microsoft SolarWinds 2021. *A deep-dive into the SolarWinds Serv-U SSH vulnerability*. https://www.microsoft.com/en-us/security/blog/2021/09/02/a-deep-dive-into-the-solarwinds-serv-u-ssh-vulnerability/

[55] Kenneth Miller, Yonghwi Kwon, Yi Sun, Zhuo Zhang, Xiangyu Zhang, and Zhiqiang Lin. 2019. Probabilistic Disassembly. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 1187–1198. https://doi.org/10.1109/ICSE.2019.00121

[56] Stefan Nagy, Anh Nguyen-Tuong, Jason D Hiser, Jack W Davidson, and Matthew Hicks. 2021. Breaking through binaries: Compiler-quality instrumentation for better binary-only fuzzing. In *30th USENIX Security Symposium*.

[57] Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2022. A Conversational Paradigm for Program Synthesis. *arXiv preprint* (2022).

[58] Vikram Nitin, Anthony Saieva, Baishakhi Ray, and Gail Kaiser. 2021. DIRECT: A Transformer-based Model for Decompiled Variable Name Recovery. *NLP4Prog 2021* (2021), 48.

[59] Philippe Ombredanne. 2020. Free and open source software license compliance: tools for software composition analysis. *Computer* 53, 10 (2020), 105–109.

[60] OpenAI. 2022. ChatGPT. https://openai.com/chat-gpt/ Accessed: 2023-03-10.

[61] OpenAI. 2022. GPT-3.5. https://platform.openai.com/docs/models/gpt-3-5

[62] OpenAI. 2022. GPT-4. https://platform.openai.com/docs/models/gpt-4

[63] OpenAI. 2023. . https://platform.openai.com/docs/models/gpt-4

[64] OpenAI. 2023. . https://platform.openai.com/docs/guides/rate-limits/overview

[65] Chengbin Pang, Ruotong Yu, Yaohui Chen, Eric Koskinen, Georgios Portokalidis, Bing Mao, and Jun Xu. 2021. Sok: All you ever wanted to know about x86/x64 binary disassembly but were afraid to ask. In *SP*. IEEE, 833–851.

[66] Kexin Pei, Jonas Guan, Matthew Broughton, Zhongtian Chen, Songchen Yao, David Williams-King, Vikas Ummadisetty, Junfeng Yang, Baishakhi Ray, and Suman Jana. 2021. StateFormer: Fine-Grained Type Recovery from Binaries Using Generative State Modeling. In *FSE* (Athens, Greece). Association for Computing Machinery, New York, NY, USA.

[67] Kexin Pei, Jonas Guan, David Williams-King, Junfeng Yang, and Suman Jana. 2021. Xda: Accurate, robust disassembly with transfer learning. In *NDSS*. The Internet Society.

[68] Kexin Pei, Zhou Xuan, Junfeng Yang, Suman Jana, and Baishakhi Ray. 2020. TREX: Learning Execution Semantics from Micro-Traces for Binary Similarity. *arXiv preprint arXiv:2012.08680* (2020).

[69] Segun I Popoola, Ruth Ande, Bamidele Adebisi, Guan Gui, Mohammad Hammoudeh, and Olamide Jogunola. 2021. Federated deep learning for zero-day botnet attack detection in IoT edge devices. *IEEE Internet of Things Journal* (2021).

[70] Alec Radford, Karthik Narasimhan, Tim Salimans, and Ilya Sutskever. 2018. Improving language understanding by generative pre-training. (2018).

[71] Alec Radford, Jeff Wu, Rewon Child, David Luan, Dario Amodei, and Ilya Sutskever. 2019. Language Models are Unsupervised Multitask Learners. (2019).

[72] Colin Raffel, Noam Shazeer, Adam Roberts, Katherine Lee, Sharan Narang, Michael Matena, Yanqi Zhou, Wei Li, and Peter J. Liu. 2019. Exploring the Limits of Transfer Learning with a Unified Text-to-Text Transformer. *CoRR* abs/1910.10683 (2019). arXiv:1910.10683 http://arxiv.org/abs/1910.10683

[73] Edward J Schwartz, J Lee, Maverick Woo, and David Brumley. 2013. Native x86 decompilation using semantics-preserving structural analysis and iterative control-flow structuring. In *USENIX Security*, Vol. 16.

[74] Stantinko 2020. *Stantinko's new cryptominer features unique obfuscation techniques*. https://www.welivesecurity.com/2020/03/19/stantinko-new-cryptominer-unique-obfuscation-techniques/

[75] Linan Tian, Yangyang Shi, Liwei Chen, Yanqi Yang, and Gang Shi. 2022. Gadgets Splicing: Dynamic Binary Transformation for Precise Rewriting. In *2022 IEEE/ACM International Symposium on Code Generation and Optimization (CGO)*. 155–167. https://doi.org/10.1109/CGO53902.2022.9741259

[76] Chaozheng Wang, Yuanhang Yang, Cuiyun Gao, Yun Peng, Hongyu Zhang, and Michael R. Lyu. 2022. No More Fine-Tuning? An Experimental Evaluation of Prompt Tuning in Code Intelligence. In *Proceedings of the 30th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Singapore, Singapore) *(ESEC/FSE 2022)*. Association for Computing Machinery, New York, NY, USA, 382–394. https://doi.org/10.1145/3540250.3549113

[77] Guozhu Wang, Yiwen Cui, Jie Wang, Lihua Wu, and Guanyu Hu. 2021. A Novel Method for Detecting Advanced Persistent Threat Attack Based on Belief Rule Base. *Applied Sciences* 11, 21 (2021).

[78] 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

[79] Shuai Wang and Dinghao Wu. 2017. In-Memory Fuzzing for Binary Code Similarity Analysis. In *ASE* (Urbana-Champaign, IL, USA) *(ASE 2017)*. IEEE Press.

[80] Yapeng Ye, Zhuo Zhang, Qingkai Shi, Yousra Aafer, and Xiangyu Zhang. 2022. D-ARM: Disassembling ARM Binaries by Lightweight Superset Instruction Interpretation and Graph Modeling. In *2023 IEEE Symposium on Security and Privacy (SP)*. IEEE Computer Society, 728–745.

[81] 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)*. 1121–1138. https://doi.org/10.1109/SP40000.2020.00035

[82] Keping Yu, Liang Tan, Shahid Mumtaz, Saba Al-Rubaye, Anwer Al-Dulaimi, Ali Kashif Bashir, and Farrukh Aslam Khan. 2021. Securing critical infrastructures: Deep-Learning-Based threat detection in IIoT. *IEEE Communications Magazine* 59, 10 (2021).

[83] Sheng Yu, Yu Qu, Xunchao Hu, and Heng Yin. 2022. DeepDi: Learning a Relational Graph Convolutional Network Model on Instructions for Fast and Accurate Disassembly. In *USENIX Security 22*. 2709–2725.

[84] Zeping Yu, Rui Cao, Qiyi Tang, Sen Nie, Junzhou Huang, and Shi Wu. 2020. Order matters: Semantic-aware neural networks for binary code similarity detection. In *Proceedings of the AAAI Conference on Artificial Intelligence*, Vol. 34. 1145–1152.

[85] 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*.

[86] Xiaochuan Zhang, Wenjie Sun, Jianmin Pang, Fudong Liu, and Zhen Ma. 2020. Similarity Metric Method for Binary Basic Blocks of Cross-Instruction Set Architecture. In *Proceedings 2020 Workshop on Binary Analysis Research*. Internet Society.

[87] 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)*. 813–832. https://doi.org/10.1109/SP40001.2021.00051

[88] 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. *Proc. ACM Program. Lang.* 3, OOPSLA, Article 137 (oct 2019), 31 pages. https://doi.org/10.1145/3360563

[89] Fei Zuo, Xiaopeng Li, Patrick Young, Lannan Luo, Qiang Zeng, and Zhexin Zhang. 2018. Neural machine translation inspired binary code similarity comparison beyond function pairs. *arXiv preprint arXiv:1808.04706* (2018).

A DETAILS OF USER STUDY

A.1 Data Availability

In this section, we first detail the setup of our user study. Then we present an exemplary sample of our user study verbatim (with minor format changes for readability). Finally, we include five concrete examples that receive scores ranging from 1 to 5 from our users.

A.2 Setup

Our user study is conducted online, with experimental data collected anonymously. We record email addresses separately from each user for distributing compensation.

To better display the code, our questions are presented to participants in a Github repository in markdown formats. Participants read the samples on the repository and enter their responses in an anonymous Google Form. For each question sample, we first describe the task, the scoring criteria, and then present both the source code and the decompiled code to our user. Variables in the decompiled code are already renamed with their ground-truth names for better understanding. On the Google Form, for each studied variable, we present four candidate names generated by different techniques and ask users to assign a score from 1 to 5 for each name.

Each participant is assigned approximately 20 variables, typically related to 6-7 functions. We ensure that each variable is scored by at least 4 users.

The following section show our question sample verbatim. Note that the source code and decompiled code are shown to our participant in a separate Github repository.

A.3 A Sample of the Study

Task Description. Thank you very much for taking this user study! Since google form is unfriendly of showing code snippets, we provide a link for markdown files at https://... We aim to evaluate techniques that infer variable names from binary code. Specifically, we want you to help us evaluate the quality of recovered variable names in the decompiled code. For each variable under study, please help us to evaluate each candidate name with the following 5-score standard:

Score-5 Candidate name is similar or better than the name in the code. The following examples are expected to be considered as score-5:

- `file_descriptor` for ground truth name `fd`
- `dst` for ground truth name `destination`
- `size` for ground truth name `length`

Also, please consider synonyms (in the code context), e.g., in function `fwrite`, for a ground-truth variable `stream`, `file` is considered as a similar name.

Score-4 Candidate name is an acceptable name. But it is not as precise as the name in the code. The following examples are expected to be considered as score-4:

- `regex_info` for ground truth name `re_pattern`
- `buffer` for ground truth name `filename_buffer`

Score-3 Candidate name has general information, but is not precisely related to the code context. The following examples is expected to be considered as score-3:

- `ptr_to_structure` for ground truth name `hash_table`, if `hash_table` in the code is indeed a pointer-typed variable AND it indeed points to a compound structure.

Score-2 Candidate name is meaningless. The following examples are expected to be considered as score-2:

- `v1`, `v2`, `v3`
- `argument`
- `some_variable`

Score-1 Candidate name is misleading. The following examples is expected to be considered as score-1:

- `signal_handler` is misleading for variable `re_pattern`

More explanations. Since decompiled code has no symbol information (i.e., variable names), we have manually assigned names to variables if we can find the corresponding variables in the source code. So you do NOT have to read into source code. We provide source code just to make sure you roughly understand the context of the function. (e.g., what the function does, what the input and output are, etc.)

Note that in the decompiled code, the data type and code structure may be incorrect.

Format. For each problem, the title 'Q1-Var1-num' means: Please open the markdown file for Q1 and see the variable 'num' in the decompiled code. We provide multiple candidate names for it (recovered by different techniques).

Now we can start the user study!

Q1-Var1-num. Please see the corresponding markdown file and rate the following candidate names.

`a1`	☐ 1	☐ 2	☐ 3	☐ 4	☐ 5
`file_index`	☐ 1	☐ 2	☐ 3	☐ 4	☐ 5
`a1`	☐ 1	☐ 2	☐ 3	☐ 4	☐ 5
`sock`	☐ 1	☐ 2	☐ 3	☐ 4	☐ 5

> ***Note:*** *The remaining questions are not shown here for simplicity.*

A.4 Concrete Examples of Different Scores

In this section, we show five concrete examples receiving scores from 5 to 1 from our user. For each example, we also specify its question ID in the title. Note that for better readability, we only show source code in this section.

Score 5. A score of 5 implies that the predicted variable name is comparable to or better than the ground truth. Fig. 14 shows a binary search function that determines whether an element is in a set. The evaluated variable has a ground-truth name of `right`, representing the upper bound of the search range. Our user assigns a score of 5 to the predicted name `upper_bound_index`. Although these two names differ, they convey the same meaning within the context of binary search.

```
static Idx __attribute__((pure))
re_node_set_contains(const re_node_set *set, Idx elem) {
    __re_size_t idx, right, mid;
    if (set->nelem <= 0)
        return 0;

    /* Binary search the element. */
    idx = 0;
    right = set->nelem - 1;
    while (idx < right) {
        mid = (idx + right) / 2;
        if (set->elems[mid] < elem)
            idx = mid + 1;
        else
            right = mid;
    }
    return set->elems[idx] == elem ? idx + 1 : 0;
}
```

Figure 14: An example of score 5. Ground-truth name (highlighted) is `right` and predicted name is `upper_bound_index`. This is a simplified version of Q26 in our user study.

Score 4. A score of 4 denotes that the predicted name is deemed acceptable, but not as precise as the ground truth. Fig. 15 illustrates an instance of a variable with a score of 4. The ground-truth name of the studied variable is `bitset_i` and the candidate name with a score 4 is `loop_index`. This variable is specifically within a function designed to execute a bitwise *Not* operation on a composite structure, `bitset_t`. The variable `bitset_i` serves as a loop index traversing the `bitset_t` structure. While the predicted name `loop_index` provides adequate information to facilitate reverse engineering, it does not explicitly convey the fact that it represents an index iterating over a `bitset_t` structure. Consequently, this variable receives a score of 4 in the evaluation.

```
static inline void bitset_not(bitset_t set) {
    int bitset_i;
    for (bitset_i = 0;
         bitset_i < SBC_MAX / BITSET_WORD_BITS;
         ++bitset_i)
        set[bitset_i] = ~set[bitset_i];
    ...
}
```

Figure 15: An example of score 4. Ground-truth name (highlighted) is `bitset_i` and predicted name is `loop_index`. This is a simplified version of Q21 in our user study.

Score 3. A score of 3 means that the predicted name encompasses general information, but not accurately correlated with the code's specific context. Fig. 16 showcases a function responsible for compiling a regular expression. The studied variable, with the ground-truth name `dfa`, is a pointer to a composite structure, `re_dfa_t`. The predicted name receiving a score of 3 is `struct_pointer`. Although it does not precisely related to the code context, a predicted name of `struct_pointer` still implies that it is a pointer referencing a structure-typed memory region. Note that the information can be helpful for a reverse engineer because structural information is typically not available in decompiled code. Identifying that a variable is a pointer to a structure-typed region can facilitate further analysis.

```
int re_compile_fastmap(struct re_pattern_buffer *bufp) {
    re_dfa_t *dfa = bufp->buffer;
    char *fastmap = bufp->fastmap;

    memset(fastmap, '\0', sizeof(char) * SBC_MAX);
    re_compile_fastmap_iter(bufp, dfa->init_state, fastmap);

    if (dfa->init_state != dfa->init_state_word)
        re_compile_fastmap_iter(
                bufp, dfa->init_state_word, fastmap);
    ...
    return 0;
}
```

Figure 16: An example of score 3. Ground-truth name (highlighted) is `dfa` and predicted name is `struct_pointer`. This is a simplified version of Q27 in our user study.

Score 2. A score of 2 means the predicted variable name is meaningless. For example, in Fig. 17, `v4` is simply a dummy name for the variable `translation`.

```
static char const *gettext_quote(
            char const *msgid, enum quoting_style s) {
    char const *translation = _(msgid);

    if (translation != msgid)
        return translation;
    ...
}
```

Figure 17: An example of score 2. Ground-truth name (highlighted) is `translation` and predicted name is `v4`. This is a simplified version of Q11 in our user study.

Score 1. A score of 1 means the predicted name is misleading. Fig. 18 delineates a function `gettext_quote` which tries to get the corresponding quote mark in the given context. The function argument `s` denotes the quoting style, but the predicted name is `input_length`, which is misleading. Such prediction is even worse than a meaningless one because it may draw the attention of analysts to the wrong direction.

```
static char const *gettext_quote(
            char const *msgid, enum quoting_style s) {
    ...

    return (s == clocale_quoting_style ? "\"" : "'");
}
```

Figure 18: An example of score 1. Ground-truth name (highlighted) is `s` and predicted name is `input_length`. This is a simplified version of Q11 in our user study.

B NAME SELECTION

After multiple iterations, each variable is associated with a list of candidate names. To select the best name from the list, LmPa employs a majority voting scheme in where each name receives a weight based on the confidence of its corresponding predictions. Consequently, names with higher confidence scores are assigned greater weight, allowing LmPa to consider both the frequency of each name in the predictions and the level of confidence in those predictions. In cases where support for the majority name is less

Table 3: Datasets

Dataset	#Funcs	#Sampled Funcs	#Variables
Coreutils	2742	181	820
Findutils	876	186	743
SQLite	3405	226	563
ImageMagick	5013	201	488
Diffutils	571	186	727
Binutils	3605	278	886
Summary	**16212**	**1258**	**4227**

Table 4: Compare LmPa with naïve inlining

	LmPa	Inline-0	Inline-1	Inline-2	Inline-3	Inline-4
Precision	32.45%	31.15%	30.94%	28.57%	31.22%	29.82%
Recall	28.74%	16.72%	20.23%	17.01%	20.23%	19.06%

than half, LmPa merely selects the most recent name with high confidence. The rationale is that queries from later iterations are more likely to possess better contextual information, thus rendering the predicted names of higher quality.

C DATASET STATISTICS

Table 3 presents the detailed dataset information, including the total number of functions, number of sampled functions, and the number of variables in the sampled functions (in columns 1 to 4, respectively). In total, we randomly select a subset of 1,258 functions from the entire pool of 16,212 functions, encompassing 4,227 variables within the sampled functions.

D ASSESSMENT WITH VARIOUS THRESHOLDS

To provide a more comprehensive depiction of LmPa's superior performance in comparison to DIRTY, we conduct an auxiliary experiment that evaluates LmPa and DIRTY under a range of threshold settings. The findings of this experiment are depicted in Fig. 19, where the left figure presents the precision under different thresholds, and the right figure demonstrates the recall. It is important to note that as the threshold increases, both precision and recall decline. This is due to the fact that a higher threshold implies a more stringent standard for "good names", which reduces the number of true positives. The results clearly demonstrate that LmPa persistently and substantially outperforms DIRTY across the entire spectrum of threshold levels.

E COMPARISON WITH A NAÏVE ALGORITHM

Given a query function, we develop a naïve algorithm that simply appends its (direct and transitive) callee functions to the query, and submits the whole query to ChatGPT. Note that we also slightly revise our prompts to let ChatGPT know which function LmPa tries to query. To avoid exceeding the input limit, we construct from Coreutils a dataset with relatively short functions consisting of 341 variables and limit the inlining bound to four. The results are shown in Table 4. Inline-N means the related algorithm recursively appends callee functions with a max depth of N. Inline-0 is the same as querying ChatGPT for one-shot. Observe that both the precision and recall for the naïve inlining algorithm are better than

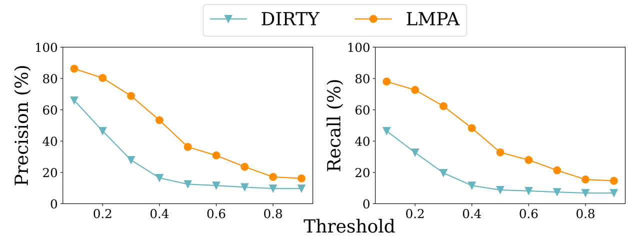


Figure 19: Ablation study on threshold for "good name". The x-axis represents for different thresholds. The y-axes represent for precision and recall, respectively. The blue and orange curves represent for the performance of LmPa and DIRTY, respectively.

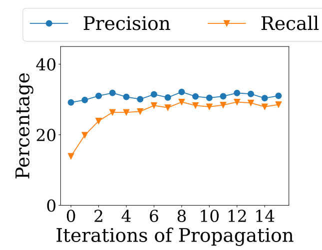


Figure 20: Performance w.r.t. number of iterations.

those for the one-shot algorithm. That indicates the context information of caller and callee functions are indeed important to LLM. However, the recall of these inlining algorithms are much lower than LmPa, because LmPa can propagate information globally and the propagation leverages precise program semantics. In addition, inlining cannot handle large functions.

F IMPACT OF THE NUMBER OF PROPAGATION ITERATIONS

Fig. 20 illustrates how the precision (blue line) and recall (orange line) of LmPa change with respect to the number of iterations. Specifically, the x-axis represents the iteration count, and the y-axis indicates the precision and recall. It is important to note that the precision remains relatively stable across varying iteration counts, as previously discussed. Conversely, the recall exhibits a significant increase after the first four rounds. This is because some variable names can hardly be derived from the local context and necessitate information from both direct or transitive callees and callers. The propagation mechanism enables ChatGPT to obtain such information. Furthermore, after the initial six iterations, both precision and recall stabilize, suggesting that all relevant information has approximately converged. This observation implies that selecting an iteration count of 10 is a reasonable choice, as it empirically ensures that LmPa reaches a fixpoint.

G SCALABILITY

Time and Query Cost. In our experiments, LmPa propagates names for 10 iterations. Note that in each iteration, we skip the queries for functions that are not changed during propagation. Also, due to the frequency limits on our OpenAI accounts [64], we insert 5 seconds waiting time after each query. Table 5 shows the number

Table 5: Number of query and time consumption. #Query denotes the number LmPa queries ChatGPT. Time/Query denotes the average time used per query.

Dataset	#Query	Time (hr.)	Time/Query (s.)
Coreutils	1566	8.7	20.0
Findutils	1909	15.8	29.8
SQLite	2046	17.1	30.1
ImageMagick	1083	5.2	17.3
Diffutils	1654	7.8	17.0
Binutils	2488	13.6	19.6
Average	1791	11.4	22.8

Table 6: Performance on larger samples. #Vars denotes the number of variables in the sample. Findutils$_L$ and Coreutils$_L$ are two larger samples from Findutils and Coreutils, respectively.

Dataset	#Vars	Precision	Recall
Findutils$_L$	2233	33.25%	29.74%
Findutils	743	30.63%	26.92%
Coreutils$_L$	2901	35.72%	31.95%
Coreutils	820	30.73%	26.34%

of ChatGPT queries for each dataset and the overall time consumption. We can see that for each dataset, LmPa queries ChatGPT 1791 times on average. It is typically related to a cost of around 5 USD. The time consumption of LmPa is relatively high, with 22.8 seconds per query and 11.4 hours per dataset on average. We inspect the logs and find that the time consumption varies depending on the server status of ChatGPT. For example, in dataset Findutils, for the same number of queries, the longest recorded time is 151 minutes while the shortest is 45 minutes. Note that in practice, as shown in Fig. 20, the user can get good performance with only 4 rounds of queries. Also, in each iteration, the queries can be conducted in parallel. We argue that, given the one-time nature of reverse engineering efforts, such resource costs are justifiable in practice.

Note that the time for propagation and all other data processing in one iteration is typically less than 50 seconds. Thus we omit the discussion for simplicity.

Performance on Larger Workload. To test whether LmPa scales well, we randomly sample two larger set of functions from Findutils and Coreutils. Each set consists of around 800 functions and more than 2000 variables. For each set, LmPa runs for four rounds. For comparison, we also collect the performance of LmPa at the fourth round on the corresponding but smaller datasets used in earlier experiments. The results are shown in Table 6. LmPa actually performs better on the larger sets in terms of both precision and recall. That is because on a larger set, LmPa has more context information to propagate, and thus can potentially provide more accurate information for the queries.

H PERFORMANCE ON UNSEEN PROGRAMS

ChatGPT is trained on enormous data. It is thus unclear whether our benchmarks have been used in ChatGPT's training. To study LmPa's performan on unseen programs, we conduct a case study on

Table 7: Comparison with DIRTY on unseen dataset. DIRTY$_N$ denotes DIRTY evaluated on samples with ground-truth function names.

Tool	Precision	Recall
LmPa	**32.31%**	**25.85%**
DIRTY	1.52%	0.54%
DIRTY$_N$	3.90%	1.63%

AudioFlux [48], an audio processing library project started in 2023. It has around 40k lines of code and receives 1k stars on Github. The chance that ChatGPT has seen the project in its training data is much lower than the others. We compile the project to a binary program, strip all the symbol and debugging information, and randomly sample 193 functions with 735 variables. We run LmPa on the dataset for 10 iterations. As shown in Table 7, the precision of LmPa is 32.31% and recall 25.85%. The precision is comparable to the others in Table 1. On the other hand, the recall is slightly lower than those in Table 1. That is because AudioFlux has many stand-alone functions (i.e., functions with no caller or callee functions). For those functions, LmPa cannot effectively propagate contextual information. Specifically, there are 7.6% stand-alone functions in AudioFlux, while the number for Coreutils is only 3%. We leave as future work to derive more sophisticated propagation rules for those functions. In comparison, we further run DIRTY on this dataset with two setups (i.e., with and without ground-truth function names, respectively). It achieves precision and recall of lower than 5% in both setups. It suggests that LmPa generalizes better than DIRTY.

I CASE STUDY: QUERY WITH PROGRAM FUNCTIONALITY DESCRIPTION

The results are shown in Fig. 21. We can see that LmPa with the whole-program information achieves better precision and recall at the first few rounds. Intuitively, that is because ChatGPT now knows the use scenario of the query function. It can generate more specific names. For example, ChatGPT predicts a variable with name `stack_frame_pointer` without the whole-program information. The ground-truth name for this variable is `db`. When told the program is from a database management system, ChatGPT realizes that the pointer may point to a database, and thus generates the name `database_handle` in the first round, which is closer to the ground-truth name. On the other hand, the advantage diminishes when LmPa propagates names for more rounds. That is because some functions in the dataset may disclose similar information, e.g., an error processing function with a literal string "`db connection error`".

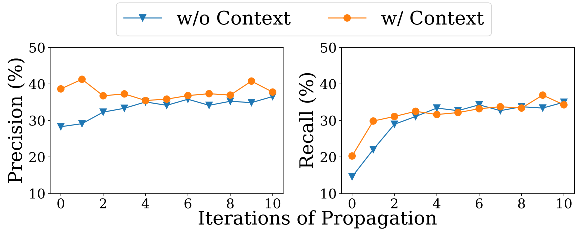


Figure 21: How whole-program functionality helps. The two figures show precision and recall for each iteration of propagation, respectively. The orange and blue curves depict the setups with and without whole-program functionality, respectively.

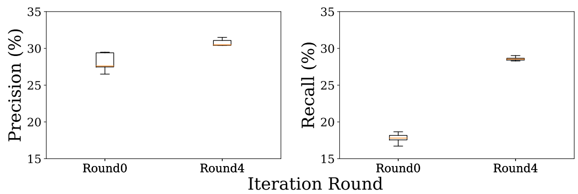


Figure 22: How performance of LmPa changes in repeated experiments. The two figures show precision and recall, respectively. The x-axis denotes different iteration rounds of propagation.

```c
int sub_4BECE3(uint32_t index){
  uint8_t result;
  if (index - 65 > 25)
        result = index;
  else
        result = index + 32;
  return result;
}

int sub_4BED21(uint8_t* listener, uint8_t* event_name){
  uint8_t *listener, *event_name;
  uint8_t v5, a;
  // ...
  do{
    a = sub_4BECE3(*listener);
    v5 = sub_4BECE3(*event_name);
    if(a == 0) break;
    ++listener;
    ++event_name;
  }while (a == v5);
  return a - vt;
}
```

Figure 23: Motivating example with names predicted by DEBIN