LMPPA: 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  
eye203@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 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 \( c_{\text{tolower}} \) (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 \( c_{\text{strcasematch}} \) (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 \( c_{\text{tolower}} \) (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 decomposed form of \( c_{\text{tolower}} \) by IDA [35], and the corresponding assembly instructions are shown in the grey box. We can observe that (1) the function name \( c_{\text{tolower}} \) 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 7).
int c_tolower(int c) {
    switch (c) {
        case 'a':
            // ...
        case 'A':
            // ...
        case 'A' + 'a':
            return c;
    }
    default:
        return c;
    }
}

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

int sub_4BED21(uint8_t* a1, uint8_t* a2) {
    uint8_t *v2, *v3;
    uint8_t v5, v6;
    // ...
    v6 = c_tolower(*v2);
    v5 = ... c1, c2; (\textsuperscript{moti:line:vardec1})
    if (p1 == p2) return 0;
    do {c1 = c_tolower(*p1); (\textsuperscript{moti:line:call-tolower0})
        // ...

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.

Figure 3: How LmP\textsubscript{a} interacts with ChatGPT. Blue and green boxes are LmP\textsubscript{a}'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 \texttt{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 \texttt{c_tolower()} is now invoked by its address 0x4BECE3. The decompiler gives it a dummy name sub_4BECE3. Without a meaningful name reflection mechanism, it is hard to understand the decompiled function.

\textbf{Limitations of State-of-the-Art Methods} DIRTY \cite{DIRTY} 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 \texttt{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 \cite{PGM} (PGM) to rename decompiled variables with names seen in the training data. We test our motivating example on DEBIN \cite{DEBIN}, a representative technique in this line. It could not generate desirable names either. For example, in \texttt{c_tolower()}, it gives variable \texttt{c} a name \texttt{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 \texttt{islit(x)} asserting \texttt{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 \texttt{c_tolower()} is too simple and does not provide much hint for DEBIN. However, in the caller \texttt{c_strcasecmp()}, individual characters in a string are passed to variable \texttt{c} in order. Such behavior pattern has been seen and encoded by the PGM, but it was connected with an id of \texttt{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 \cite{LLM1, LLM2, LLM3, LLM4, LLM5} are typically trained on multi-modal data of an enormous scale. They demonstrate superior capabilities in many natural language tasks and coding tasks \cite{LLM6, LLM7, LLM8}. 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 is 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 \cite{GPT4}).
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 variable c_tolower(). Conversely, ChatGPT generates a name that 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.

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: S1; S2 is used to concatenate two statements; E1 := E2 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; E1 ∘ E2 denotes a binary operation over two operand expressions; and Id(E1, E2, ...) 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 = propagationRet ∪ propagate(decl.with 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.

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(name0, id0, id1)` indicates a string (name0) is considered a good name for an identifier (id1) in a function (id0). Similarly, `GoodNameOf(name0, id0, e1)` indicates name0 is a good name for an expression e1 in function id0. `CallerOf(id0, id1)` indicates id0 is a caller of id1. 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.

```
Figure 7: Relations and Functions for Name Propagation
```

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 id2 is found in a statement of id1, then there is a relation `CallerOf(id1, id2)`. That is, id1 is a caller of id2. Rules GN-Id and GN-Lit denote starting points of our inference. `GN-Id` denotes that if in the function id1, ChatGPT predicts a name n for id with high confidence, then n is considered a good name for id in function id1. `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 e1 ◦ e2 if both sub-expressions 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
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 id0 in the context of function id0. It renames all the invocations to id0 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 ei. 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.

Figure 8: Propagation Rules

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 LLMs 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 lines of Python code and 3,214 lines of C++ code. 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.

\[
\text{Similarity}(S_{TP}, S_P) = \frac{|\text{LCS}(S_{TP}, S_P)|}{|S_{TP}|}
\]

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>
<thead>
<tr>
<th>Dataset</th>
<th>Total</th>
<th>LmPa</th>
<th>DIRTY</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Precision</td>
<td>Recall</td>
</tr>
<tr>
<td>Coreutils</td>
<td>820</td>
<td>30.86%</td>
<td>27.93%</td>
</tr>
<tr>
<td>Findutils</td>
<td>743</td>
<td>32.33%</td>
<td>30.42%</td>
</tr>
<tr>
<td>SQLite</td>
<td>563</td>
<td>27.77%</td>
<td>26.29%</td>
</tr>
<tr>
<td>ImageMagick</td>
<td>488</td>
<td>48.73%</td>
<td>43.24%</td>
</tr>
<tr>
<td>Diffutils</td>
<td>727</td>
<td>31.28%</td>
<td>31.09%</td>
</tr>
<tr>
<td>Binutils</td>
<td>886</td>
<td>30.15%</td>
<td>27.77%</td>
</tr>
<tr>
<td>Average</td>
<td>-</td>
<td>33.85%</td>
<td>31.12%</td>
</tr>
</tbody>
</table>

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.
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, parameters 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 Naive 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 is justifiable in practice, given the one-time nature of reverse engineering efforts. Furthermore, LmPa scales 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 Coreutil 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.

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.
We develop a novel technique for symbol name recovery in decom-
ware fingerprinting [15], APT attack forensics [2, 22, 43, 69, 77, 82],

Table 2: Performance on GPT-4. LMpA GPT4 and LMpA CHAT
denote LMpA with GPT-4 and ChatGPT, respectively.

<table>
<thead>
<tr>
<th>Tool</th>
<th>Iteration: 0</th>
<th>Iteration: 4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Precision</td>
<td>Recall</td>
</tr>
<tr>
<td>LMpA GPT4</td>
<td>50.80%</td>
<td>28.57%</td>
</tr>
<tr>
<td>LMpA CHAT</td>
<td>41.25%</td>
<td>23.57%</td>
</tr>
</tbody>
</table>

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], soft-
ware fingerprinting [15], APT attack forensics [2, 22, 43, 69, 77, 82], and software reuse [21, 68]. LMpA is intrinsically connected to de-
compilation [27, 33, 47], a foundational task in binary analysis. In addition to the related works discussed in Section 2.1, substantial re-
search 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 sum-
marization [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 decom-
pilation. It leverages the synergy between large language models and program analysis. It features an iterative algorithm that propa-
gates 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 20.3% in precision and recall, respectively.

REFERENCES

fied Pre-training for Program Understanding and Generation. In Proceedings of the 2021 Conference of the North American Chapter of the Association for Computa-


[8] Check Point Research 2022. TWISTED PANDA: CHINESE APT ESPIONAGE OPERATION AGAINST RUSSIAN STATE-OWNED DEFENSE INSTITU-


conference/usenixsecurity22/presentation/chin-qibin


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 users. 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 the following candidate names.

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 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 misleading. 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.

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

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>a1</td>
<td></td>
<td></td>
<td>□</td>
<td></td>
<td></td>
</tr>
<tr>
<td>file_index</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
</tr>
<tr>
<td>a1</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
</tr>
<tr>
<td>sock</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
</tr>
</tbody>
</table>

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 users. 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 the following candidate names.

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 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 misleading. 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.

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

<table>
<thead>
<tr>
<th></th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
</tr>
</thead>
<tbody>
<tr>
<td>a1</td>
<td></td>
<td></td>
<td>□</td>
<td></td>
<td></td>
</tr>
<tr>
<td>file_index</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
</tr>
<tr>
<td>a1</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
</tr>
<tr>
<td>sock</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
<td>□</td>
</tr>
</tbody>
</table>
static int __attribute__((pure)) re_node_set_contains(const re_node_set *set, Idxelem) {
    re_size_t idx, right, nelem;
    if (set->nelem == 0)
        return 0;
    idx = 0;
    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;
}

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];
    ...}

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)
        re_compile_fastmap_iter(
            bufp, dfa->init_state_word, fastmap);
    ...
    return 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-type memory region. Note that the information can be helpful for a reverse engineer because structural information is typically not available in decoded code. Identifying that a variable is a pointer to a structure-type 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)
        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 meaningful. For example, in Fig. 17, v4 is simply a dummy name for the variable translation.

static char const *gettext_quote(char const*msgid, enumquoting_styles) {...
    return (s== clocale_quoting_style? "\": ");
}

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, enumquoting_styles s) {
    char const *translation = s(msgid);
    if (translation != msgid)
        return translation;
    ...}

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
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 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.

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 from both direct or transitive callees and callers. The propagation leverages precise program semantics. In addition, inlining cannot handle large functions.

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 of true positives. The results clearly demonstrate that LmPa persistently and substantially outperforms DIRTY across the entire spectrum of threshold levels.
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 performance on unseen programs, we conduct a case study on 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".

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

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

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

<table>
<thead>
<tr>
<th>Dataset</th>
<th>#Vars</th>
<th>Precision</th>
<th>Recall</th>
</tr>
</thead>
<tbody>
<tr>
<td>FindutilsL</td>
<td>2233</td>
<td>33.25%</td>
<td>29.74%</td>
</tr>
<tr>
<td>Findutils</td>
<td>743</td>
<td>30.63%</td>
<td>26.92%</td>
</tr>
<tr>
<td>CoreutilsL</td>
<td>2901</td>
<td>35.72%</td>
<td>31.95%</td>
</tr>
<tr>
<td>Coreutils</td>
<td>820</td>
<td>30.73%</td>
<td>26.34%</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Tool</th>
<th>Precision</th>
<th>Recall</th>
</tr>
</thead>
<tbody>
<tr>
<td>LmPa</td>
<td>32.31%</td>
<td>25.85%</td>
</tr>
<tr>
<td>DIRTY</td>
<td>1.52%</td>
<td>0.54%</td>
</tr>
<tr>
<td>DIRTYN</td>
<td>3.90%</td>
<td>1.63%</td>
</tr>
</tbody>
</table>

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.
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.

Figure 23: Motivating example with names predicted by DEBIN

```c
int sub_4BECE3(uint32_t index) {
    uint8_t result;
    if (index - 65 > 25)
        result = index;
    else
        result = index + 52;
    return result;
}
```

```c
int sub_4BED21(uint8_t *listener, uint8_t *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 = v5;
}
```