

A Human Study of Automatically Generated Decompiler Annotations

Yuwei Yang

*Department of Computer Science
Vanderbilt University
Nashville, USA
yuwei.yang@vanderbilt.edu*

Skyler Grandel

*Department of Computer Science
Vanderbilt University
Nashville, USA
skyler.h.grandel@vanderbilt.edu*

Jeremy Lacomis

*Software and Societal Systems Department
Carnegie Mellon University
Pittsburgh, USA
jlacomis@cmu.edu*

Edward Schwartz

*Software Engineering Institute)
Carnegie Mellon University
Pittsburgh, USA
eschwartz@cert.org*

Bogdan Vasilescu

*Software and Societal Systems Department
Carnegie Mellon University
Pittsburgh, USA
vasilescu@cmu.edu*

Claire Le Goues

*Software and Societal Systems Department
Carnegie Mellon University
Pittsburgh, USA
clegoues@cs.cmu.edu*

Kevin Leach

*Department of Computer Science
Vanderbilt University
Nashville, USA
kevin.leach@vanderbilt.edu*

Abstract—Reverse engineering is a crucial technique in software security, enabling professionals to analyze malware, identify vulnerabilities, and patch legacy software without access to source code. Although decompilers attempt to reconstruct high-level code from binaries, essential information, such as variable names and types, is often dissimilar from the original version, hindering readability and comprehension.

Recent advancements have employed AI to enhance decompiler output by recovering original variable names and types. Traditional evaluation of recovery techniques relies on measuring similarity between original and recovered names, assuming that higher similarity enhances readability. However, studies suggest that these “intrinsic” metrics may not accurately predict “extrinsic” outcomes like user comprehension or task performance, revealing a gap in understanding readability and cognitive load in reverse engineering.

This paper presents an extrinsic evaluation of machine-generated variable and type names, focusing on their impact on reverse engineers’ comprehension of decompiled code. We conducted a user study with 40 participants—including students and professionals—to assess code comprehension both with and without AI-generated variable and type name assistance. Our findings indicate a lack of correlation between traditional machine learning metrics and actual comprehension gains, highlighting limitations in current evaluation techniques. Despite this, participants showed a preference for AI-augmented decompiler outputs. These insights contribute to understanding the effectiveness of automatic recovery techniques in enhancing reverse engineering tasks and underscore the need for comprehensive, user-centered evaluation frameworks.

Index Terms—Decompilation, Binary Reverse Engineering, Human Study

I. INTRODUCTION

Reverse engineering is an essential tool for software security professionals, enabling them to analyze the behavior or provenance of malware [3]–[5], discover vulnerabilities [5, 6], and patch bugs in legacy software [5, 6]. Without access to

source code, analysis often occurs at the binary level, where compiler optimizations for speed or size hinder readability [2].

To navigate these challenges, tools like disassemblers and decompilers are essential. Disassemblers translate binary instructions into assembly language, while decompilers, like Hex-Rays [1], Ghidra [7], and ANGR [8], go a step further by attempting to reconstruct the higher-level source code from binaries, offering a view closer to the original code structure [9]. Decompiled code is still often quite dissimilar from source code, however, as it is missing information like variable names and types [2, 10].

Recent works have focused on augmenting decompiler output by attempting to recover the original variables, types, structures, and comments to address this problem [2, 11]–[13]. These approaches use machine learning to infer and restore variable names, types, and structural details that are otherwise lost during the decompilation process. By training on extensive code repositories, these models learn patterns and context cues, enabling them to generate plausible names and types for the variables in the decompiler output. Evaluations of these methods typically rely on similarity metrics, comparing the AI-generated names and structures to those in the original source code. Based on these metrics and existing research showing the importance of variable names and types, researchers have claimed significant improvements in the readability of AI-augmented decompiled code over standard decompiled code [2, 11]–[13].

Still, even state-of-the-art approaches to variable and type name recovery have room for improvement. Consider the examples shown in Figure 1. This figure compares original source code to its decompiled counterpart with AI-generated variable and type names. The generated names, while potentially helpful, do not express the variable or type’s purpose

```

data_unset * array_extract_element_klen
(array * const a, const char *key, const uint32_t klen) {
    const int32_t ipos = array_get_index(a, key, klen);
    if (ipos < 0) return NULL;

    data_unset * const entry = a->sorted[ipos];
    ...
    return entry;
}

```

(a) Original Source Code

```

char *__fastcall array_extract_element_klen
(array_t_0 *array, void *key, int index){
    int indexa; // [rsp+28h] [rbp-18h]
    char *next; // [rsp+30h] [rbp-10h]

    indexa = array_get_index((__int64)array, (__int64)key, index);
    if ( indexa < 0 )
        return 0LL;
    next = *(char **)(8LL * indexa + *(_QWORD *)&array->size);
    ...
    return next;
}

```

(b) Decompiled Binary with Name Recovery

Fig. 1: Here we compare the `array_extract_element_klen` function from the `lighttpd` project before and after it has been compiled, decompiled, and augmented with AI-generated variable and type names. Here we use the Hex-Rays decompiler [1] and the DIRTY tool for recovering variable and type names [2]. This function locates an element within a custom array type by a given key and retains metadata within the array. Colors in these examples show corresponding variables and types. This comparison highlights the information that is lost in the process of compilation and the potential inaccuracies of current tools, increasing the burden of security professionals when they attempt to reverse engineer software.

in the same way the original (human-written) names do, nor are they particularly similar in appearance to the original names. Take, for example, `data_unset *` and `char *` or `klen` and `index`; these renamings are somewhat dissimilar in meaning, potentially misleading reverse engineers inspecting the code.

Previous works in this field leverage the insight that variable names and types are known to meaningfully contribute to program comprehension [14, 15] and that renaming variables and reconstructing types are some of the most common tasks performed by reverse engineers [16]. To evaluate these approaches, researchers typically measure the similarity between the original source code and the recovered variable and type names, under the assumption that higher similarity according to their metric will improve readability [11]–[13]. However, prior studies on similar tasks in code comprehension suggest that “intrinsic” similarity metrics alone may not reliably predict improvements in “extrinsic” measures like user comprehension or performance [17]. This disconnect indicates that even high similarity scores might not fully capture the nuances of readability and cognitive load in reverse engineering tasks.

We evaluate the extrinsic quality of machine-generated variable and type names by measuring their impact on programmers’ comprehension of decompiled code. Participants perform code comprehension tasks with and without AI-generated names. Comparing their performance reveals how automatic recovery techniques support security professionals during reverse engineering.

To our knowledge, no peer reviewed human studies on automatic variable and type-name recovery techniques have conducted a comprehensive extrinsic evaluation of their impact on program comprehension, nor have they analyzed correlations with intrinsic evaluation metrics. Several factors likely contribute to this gap: the significant cost of recruiting qualified programmers, the challenge of designing tasks that authentically capture real-world comprehension processes, and

the prevailing reliance on intrinsic evaluations as sufficient for publication.

We address this gap by conducting an extrinsic study evaluating how machine-generated variable names and types aid program comprehension. Forty participants—30 students, 9 professionals, and 1 unemployed individual, all with at least one semester or year of reverse engineering experience—completed decompiled code comprehension tasks, some with machine-generated renaming support.

This paper makes the following key contributions:

- **Empirical Evaluation of ML Performance Metrics:** We find that current ML performance metrics do not correlate with the effectiveness of variable and type name recovery for program comprehension, consistently challenging assumptions about metric reliability across multiple evaluations.
- **User Preference for ML-Augmented Decompiler Output:** Despite minimal performance gains, participants notably preferred ML-augmented code, highlighting a perceived value even without measurable comprehension improvements.
- **Developer Performance and Enriched Code Analysis:** Our findings indicate no significant correlation between developer task performance and the presence of enriched source code. This insight suggests that current enrichment techniques may have limited impact on practical comprehension outcomes.

II. RELATED WORK

Here we discuss two key research areas relevant to our study: empirical studies of program comprehension, with an emphasis on reverse engineering contexts, and machine learning approaches for variable and type name recovery in decompiled code.

A. Program Comprehension

Empirical studies of program comprehension in reverse engineering often focus on understanding the cognitive processes involved and evaluating the effects of factors like code obfuscation or recovery of missing information on comprehension. These two research streams are closely related: models of comprehension in reverse engineering help guide tool development to address specific needs of security professionals [18].

1) *Comprehension Models in Reverse Engineering*: In reverse engineering, comprehension models extend general program comprehension models, incorporating top-down, bottom-up, and integrated approaches [19]–[21]. Top-down comprehension typically involves forming hypotheses about a binary’s functionality and refining them with further details, while bottom-up comprehension begins by analyzing low-level instructions and gradually constructing higher-level abstractions. Integrated models combine both strategies, enabling practitioners to adapt according to the task at hand.

A key aspect of comprehension in reverse engineering is the search for structural cues—known as “beacons”—that provide clues about code functionality. While program comprehension research has identified several common beacons—such as API calls, strings, variable names, sequences of operations, and constants—that aid in understanding code structure and functionality [22]–[25], developers face significant challenges when these elements are obfuscated or absent [25]. In reverse engineering, professionals must adapt to this resource-starved environment by relying on alternative beacons, such as control flow structures, compiler artifacts, and program flow [16]. However, if beacons like variable and type names could be recovered, they would provide semantic information that could significantly enhance comprehension, helping practitioners to form more accurate hypotheses about code behavior and intent.

2) *Evaluation of Decompiled Code Comprehension*: Human studies in reverse engineering, particularly those evaluating decompiled code comprehension, remain limited. Hu et al. [13] conducted a human study to assess optimizations of decompiler output using large language models, evaluating user comprehension with and without optimizations by measuring differences in correctness and task completion time; however, this study does not provide an in-depth analysis of its human subjects study, leaving questions about the causes of improved comprehension and the correlation between intrinsic metrics and human comprehension underexplored. Similarly, Cao et al. [26] examined the effectiveness of modern decompilers by assessing user correctness in recompiling decompiled code, noting that decompiler outputs often fail to compile without modification. Votipka et al. [16] indirectly explored reverse engineering through structured interviews with practitioners, offering insights into the practical challenges and strategies reverse engineers employ.

Given the scarcity of studies directly addressing decompiled code comprehension, we draw on evaluation methods from related areas of software comprehension, such as code summarization [17, 27]–[29], code regularity [30], and patch management [31]. These studies commonly employ empirical

methods that measure comprehension through task completion time, correctness, or qualitative feedback. Although these approaches are not specific to reverse engineering, they establish a foundation for designing evaluation techniques adaptable to decompiled code contexts.

A prevalent methodology in these studies involves using questions identified by Sillito et al. [32] as relevant to developer comprehension. Based on empirical studies, Sillito’s work highlights common question types programmers encounter during software evolution tasks. Additionally, Pacione et al. [33] devised a set of nine principal comprehension activities by reviewing tasks used across comprehension evaluation literature. In our work, we adapt Sillito et al.’s questions to suit reverse engineering tasks, collaborating with a professional reverse engineer to ensure the questions’ relevance and realism. This approach of tailoring comprehension questions to specific domains is well-established in prior studies [17, 30, 31].

B. Variable and Type Name Recovery

Recent works apply machine learning to improve the readability of decompiled code by recovering variable names, types, and structural details. These techniques typically involve training deep learning models on large code repositories to learn patterns that can be applied to decompiled binaries, and leveraging both lexical and structural information to enhance the readability of decompiler output. Jaffe et al. utilized statistical machine translation techniques for variable renaming, proposing an alignment method between source code and decompiler output to construct an effective training set [34]. Building on this, Lacomis et al. developed DIRE, which incorporates structural information recovered by the decompiler to improve variable renaming [11]. DIRE employs an LSTM encoder to capture lexical features and a Gated Graph Neural Network (GGNN) to encode structural information, thus providing a more contextually aware model for renaming tasks. DIRECT, a tool by Nitkin et al., extends DIRE using transformer-based models [12]. DIRTY further advances these techniques by integrating data layout information from the decompiler [2]. Using a transformer-based model, DIRTY predicts both variable names and types, demonstrating the potential of transformer architectures in handling the complex relationships within decompiled code. Most recently, Hu et al. introduced deGPT, an end-to-end framework for refining decompiler output using a Large Language Model [13]. This work uses a three-role mechanism—referee, advisor, and operator—to optimize readability through structure simplification, variable renaming, and comment generation. Hu et al. also conduct a user study to measure preference and comprehension gains, but this study is based solely on subjective metrics like user preference and subjective grading. We conduct a study with objective measures of code comprehension and correlate them with intrinsic and subjective measures to establish the extent to which these measures signify actual improvements in comprehension.

III. EVALUATION METHODOLOGY

A. Study Design

Our study draws inspiration from two prior works: one that assessed the maintainability of computer-generated patches [31] and another that evaluated the effectiveness of a research decompiler [5]. In these studies, participants reviewed code snippets and responded to questions about them. Similarly, we provided participants with decompiled code snippets generated by Hex-Rays v8.2 [1], prompting them to analyze the code and answer various questions regarding its functionality. To gauge participants' perceptions, we also gathered feedback on the quality of type names and asked participants to rate their agreement with general statements about the code overall. For each code snippet, each participant received either a snippet directly from the decompiler or one enhanced with DIRTY's output, assigned randomly.

B. Code Selection

Our study design imposed specific constraints on the code snippets used. First, to fit within the one-hour study limit and avoid scrolling issues, snippets had to be short enough to fit on a single screen with the questions, limiting each snippet to a maximum of 50 lines. Second, the snippets needed to be sufficiently "interesting" to detect performance differences between groups, which we ensured by selecting snippets with at least two levels of nested structures, such as if branches or for loops. Third, each snippet needed to be self-contained so that questions could be asked without additional context on the functions involved. Fourth, as our study examines the effects of renaming and retyping tools on performance, each snippet required at least three renamed or retyped variables.

We sourced the snippets from the projects `lighttpd`, `coreutils`, and `openssl`, as these projects include common functionality—such as networking, encryption, and file handling—that is often repurposed by malware developers. To ensure the questions were relevant, we consulted a professional reverse engineer. The selected snippets were as follows:

- **`array_extract_element_klen` (AEEK):** This function from `lighttpd` locates an element within a custom array type by a given key and retains metadata within the array.
- **`buffer_append_path_len` (BAPL):** This function from `lighttpd` concatenates two file paths while ensuring only one path separator appears between them. For instance, given the inputs `"usr/"` and `"/bin"`, calling `buffer_append_path_len()` will yield `"usr/bin"`.
- **`postorder`:** This function from `coreutils` accepts a binary tree, a function pointer, and auxiliary information, calling the function pointer at each node in `postorder` traversal of the binary tree.
- **`twos_complement` (TC):** This function from `openssl` takes an input buffer, an output buffer, and a length. It copies the input buffer to the output buffer, and if the padding argument is set to `0xff`, it converts the input buffer to its two's complement form before copying.

If `a1 + 8` points to an array and the `array_get_index` call on line 8 returns an index, what is the purpose of the `if` and `memmove` on lines 13-17?

Please write your answer here:

Fig. 2: AEEK question 1, an example question from our survey. This question appears directly below the AEEK function in a syntax highlighted window.

C. Question Formulation

To assess the effectiveness of our method in supporting the reverse engineering process, we crafted questions akin to those typically asked by reverse engineers when analyzing binaries. The questions included the following types:

- If the function is called with arguments X, what will be the value of Y at line number Z?
- What is the purpose of the code from lines X to Y?
- What are the potential return values of this function?
- Which argument in this function is associated with functionality X?

These questions are modeled after those used in a prior study [31] and were developed in collaboration with a professional reverse engineer to ensure practical relevance. Our questions were formulated to have well-defined and unambiguous answers to facilitate objective manual grading. Based on this approach, we created two questions per code snippet, resulting in a total of eight questions. Figure 2 includes an example of a question that participants answered in this study. All code snippets and specific questions asked for each are included in our replication package (cf. Section VIII).

D. Experimental Design

We conducted a between-subjects experiment in which participants analyzed code snippets either decompiled solely with Hex-Rays or with Hex-Rays augmented with machine-generated variable and type names by DIRTY [2]. We choose DIRTY over alternative tools [11]–[13] because it is the best performing tool focused on both variable and type name generation that does not perform additional augmentations that would act as confounding variables for our study. The study was administered online through LimeSurvey. Each participant received a comprehensive overview of the study and the procedure before beginning. To ensure our focus remained solely on code quality, participants were not permitted to use the Internet during the study.

Each participant reviewed one code snippet at a time and answered questions about it. All four code snippets were shown to every participant, with treatment randomized by snippet (e.g., one participant might view the Hex-Rays version

of `buffer_append_path_len` and the DIRTY version of `postorder`). This randomization offered flexibility, as incomplete responses from a participant would not exclude an entire group from the treatment for all questions. We collected data on both timing and accuracy for each participant.

Variables and Conditions. Our experiment included two independent variables:

- 1) Treatment: Code snippets were decompiled using the Hex-Rays decompiler, widely utilized by malware analysts. We used Hex-Rays version 8.2.230124, the most current version at the time, comparing the standard decompiler output with the same output supplemented by DIRTY annotations.
- 2) Questions: Since each question varies in difficulty, they are analyzed individually.

User Perception. Upon completing the questions for each snippet, participants completed a brief survey. They rated the impact of argument types and names on their understanding on a 1-5 scale and could optionally suggest improved names or types. Additionally, they rated their agreement with statements reflecting their overall impressions of the code on a 5-point scale. The full list of statements can be found in our replication package (c.f., Section VIII).

E. Participants and Recruitment

To assess our tool’s impact on real-world reverse engineering, we recruited participants with verified reverse engineering expertise. Personalized invitations were sent to professional reverse engineers at various companies and institutions as well as student members of capture-the-flag (CTF) teams with experience in reverse engineering. Each participant received an email soliciting participation and giving information on the task. This email specified the focus on the impact of type and variable names on reverse engineering and included a link to allow recipients to participate remotely via LimeSurvey.

We received responses from 31 students, 10 professionals, and 1 individual currently unemployed. Participation was anonymous, and no compensation was provided. Although responses to all questions were optional, we implemented a quality check to identify any cases of rapid, non-meaningful responses. We required participants to spend at least as much time on each snippet as it took an author to fully read the question. Based on this criterion, we excluded data from one student and one professional who were removed from the study entirely.

IV. EXPERIMENTAL RESULTS

In this section, we present our findings on the impact of machine-generated variable renamings and type recoveries on reverse engineering tasks, specifically using the DIRTY model as our test case.

Each research question targets a distinct aspect of comprehension and performance:

- **RQ1:** Do renamings and retypings allow reverse engineers to correctly answer more questions about decompiled code?

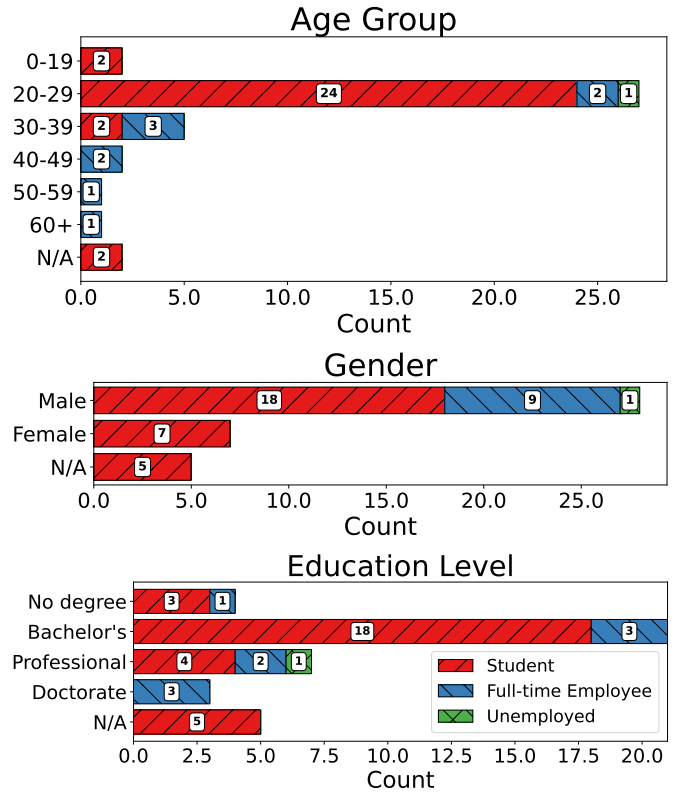


Fig. 3: The distribution of age, gender, and education level among our participants. N/A denotes participants who selected that the “prefer[ed] not to answer” the given question.

- **RQ2:** Do renamings and retypings allow reverse engineers to answer questions about decompiled code more quickly?
- **RQ3:** Do users of DIRTY perceive its renamings and retypings as improving their understanding?
- **RQ4:** Do users’ perceptions of DIRTY’s helpfulness align with their performance?
- **RQ5:** Do similarity metrics such as BLEU scores correlate with code comprehension?

A. RQ1: Correctness

The first research question asks if machine generated variable and type renamings allow users to answer more questions correctly. Recall that each participant is randomly assigned to a treatment group for each of 4 snippets (i.e., a snippet with DIRTY annotations, or a snippet without DIRTY annotations). For each snippet we ask the participant 2 questions, for a total of 8 questions per participant.

Our analysis compares variable distributions between tasks completed under treatment and control conditions, accounting for differences in question difficulty, participant experience in both general coding and reverse engineering, and repeated responses per participant. To address the variability in participant skill and task complexity, we used a mixed-effects regression model [35], which groups residuals by random effects—here, by user and question.

In R syntax, our model is:

$$\text{correctness} = \text{uses_DIRTY} + \text{Exp_Coding} + \text{Exp_RE} + (1|\text{user}) + (1|\text{question}) \quad (1)$$

Since correctness is binary, we apply logistic regression, using `glmer` in R and setting statistical significance at $p < 0.05$. To evaluate model fit, we report both marginal (R_m^2) and conditional (R_c^2) coefficients, computed with the `r.squaredGLMM` function [36].

TABLE I: GLMER Correctness Performance Model

Dependent variable: Correctness	
Uses DIRTY	-0.074 ± 0.227
General Coding Experience	0.056 ± 0.030
Reverse Engineering Experience	-0.024 ± 0.044
Constant	0.563 ± 0.513
Observations	273
Num Users	36
Num Questions	8
$\sigma(\text{Users})$	0.85
$\sigma(\text{Questions})$	1.14
R_m^2	0.041
R_c^2	0.405
Akaike Inf. Crit.	313.091
Bayesian Inf. Crit.	334.747

Note: For all correctness values, $p > 0.05$

Table I summarizes the results of our model. This model fits our data reasonably well, with an R_c^2 of 40.5%. **We found no statistically significant difference when participants use DIRTY and do not have sufficient evidence to conclude DIRTY users answer more questions correctly. In fact, the usage of variable renaming has a slight (though insignificant) negative effect on correctness on average. However, the large standard error and insignificance make this result inconclusive.**

Incorrect or imprecise renamings can be misleading. Occasionally, DIRTY can be misleading. An example is shown in Figure 4. In this question, `postorder` question 2, we told the participants that the three arguments represented a pointer to a tree structure, a function pointer to call on each node, and auxiliary information to maintain as the tree was being traversed and asked them to match the arguments to their description. The body of the code example ¹ makes it fairly clear that the first argument is the tree structure, meaning participants must reason more about the function pointer and auxiliary information. Figure 4a shows the original Hex-Rays version of the code, which strongly suggests that the second argument is the function pointer. DIRTY’s suggestions in Figure 4b are seemingly quite good; it correctly identifies that two of the arguments correspond to a tree and a comparison function. However, it reverses the order of the last two suggestions. Figure 5 depicts the correctness of answers to the questions asked during the survey. This shows that while

¹The body of this function is largely omitted for brevity, but it can be found in our replication package.

```

1 __int64 __fastcall postorder(
2     _QWORD *a1,
3     __int64 (__fastcall *a2)(__int64, _QWORD *),
4     __int64 a3) {
5     // ...
6     v5 = a2(a3, a1);
7     // ...
8 }

```

(a) Hex-Rays

```

1 __int64 __fastcall postorder(
2     tree234 *t,
3     void *e,
4     cmpfn234 cmp) {
5     // ...
6     ret = (e)(cmp, t);
7     // ...
8 }

```

(b) DIRTY

Fig. 4: `postorder` question 2: an example where DIRTY suggests reasonable types and names but applies them to the wrong arguments. Participants were asked to match each argument to its purpose. In (a), `a1` is a tree pointer, `a2` a function pointer, and `a3` auxiliary info. In (b), DIRTY correctly identifies a tree and function type but swaps `a2` and `a3`. This mistake could be the reason that participants who did not receive DIRTY renamings performed better on this question.

almost every person who received only the Hex-Rays output answered this question correctly, nearly half of the participants who received the DIRTY annotations answered this question incorrectly (a Fisher’s exact test on this data confirms this confusion, $p = 0.01059$).

Anticipating this, we also asked participants to justify their answers by answering the question “Informally, how did you reach your conclusion?” For qualitative analysis we used the standard grounded theory approach of open coding [37]. Each response was individually coded, then these codes were synthesized and used to identify themes. Two main themes were identified in answers from the users who received the DIRTY version of the code, which were directly correlated with correctness of their answer.

Among participants who received the DIRTY code and answered correctly, we identified the theme: The usage of the variables inside the code demonstrate their purpose (P5, P6, P7, P8, P9, P11, P14, P15, P16, P17, P18, P19). These participants indicated that they considered not just the names and types of the variables, but also their usage in the code. These participants summarize this thought process well:

“Line [6] shows the actual function call; that requires `e` to be the function and `cmp` to be an argument to it, at odds with the type information in the arguments” (P8)

“Similar to the previous answer – I ignored the types and looked at the use. The only actual call through a function pointer is on line [6], so `e` is the visit/comparison function. It is passed in `cmp` (which is never changed, despite being confusingly

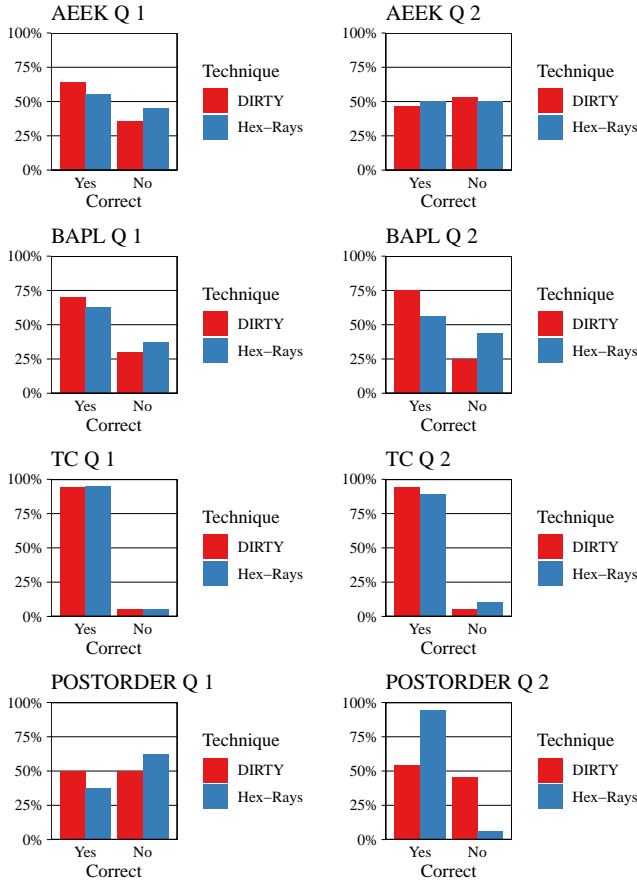


Fig. 5: Answers to questions grouped by treatment. “Yes” means the answer was correct, “No” means incorrect.

named the same as the `> cmp` field), so the `cmp` argument is the additional information [...] (P11)

Participants who answered the question incorrectly reached their conclusion for a different reason: the variable names and types themselves indicate their intended usage (P1, P2, P3, P4, P6, P10, P12, P13). These participants took the types and names at face-value and did not consider their usage in the code. Indicative responses are:

“The variable names were very intuitive. For the tree, the type and its usage in the code was really helpful.” (P1)

“The main giveaway is the naming. Also I see that `cmpfn234` is defined as a function pointer. The naming are very descriptive and helped in identifying what each component does.” (P13)

Notice how participants who referenced the code itself and were skeptical of the types suggested by DIRT got the answer correct, while participants who got the answer incorrect trusted the types it suggested. The pattern of accepting DIRT’s annotations at face value was not unique to the `postorder` example. For users who received DIRT annotations, we compared groups based on their correctness by the Likert opinions participants assigned to the types DIRT

suggested using a Wilcoxon rank sum test with continuity correction. We found that participants who answered incorrectly tended to trust DIRT’s suggestions more than participants who answered correctly across the board ($p = 0.02477$). This correlation could suggest that it is important to train users to remain skeptical while reverse engineering, even with the types suggested by tools like DIRT.

We did not find statistical evidence that using tools like DIRT allows for more users to reach a correct conclusion. However, there is some anecdotal evidence that these tools might help when used correctly while considering code structure, which may help inform future tool development and use.

B. RQ2: Timing

Research question 2 asks if DIRT has an impact on the timing of participants’ answers. Similarly to correctness, we fit a mixed-effects model of the equation using the user and question as random effects. Below is the R formula we used:

$$\text{timing} = \text{uses_DIRTY} + \text{Exp_Coding} + \text{Exp_RE} + (1|\text{user}) + (1|\text{question}) \quad (2)$$

TABLE II: LMER Timing Performance Model

Dependent variable: Completion Time	
Uses DIRT	26.296 ± 16.865
General Coding Experience	4.488 ± 2.620
Reverse Engineering Experience	−5.647 ± 3.948
Constant	192.658* ± 54.308
Observations	296
Num Users	37
Num Questions	8
$\sigma(\text{Users})$	94.77
$\sigma(\text{Questions})$	130.96
R^2_m	0.025
R^2_c	0.431
Akaike Inf. Crit.	4,026.521
Bayesian Inf. Crit.	4,052.354

Note: * $p < 0.05$

Unlike the binary “correctness” results, timings are continuous and we can use a standard linear mixed-effects model provided by the `lmer` function in R. Our results are shown in Table II. First, the model does fit our data reasonably well, with an R^2_c of 43.1%. As with the correctness results, none of our coefficients were statistically significant. We did not find statistically significant evidence to conclude that users of DIRT were able to answer questions more quickly. **We find no significant difference in completion time between the groups that use and do not use AI variable and type renaming.**

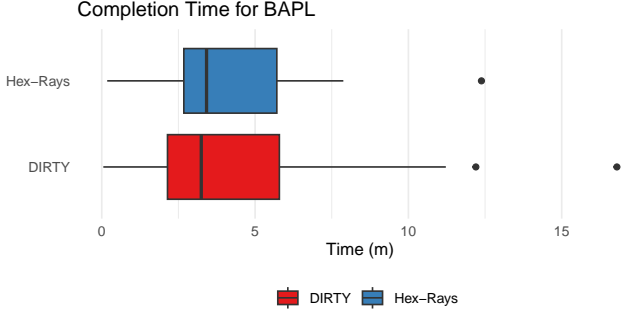
In some cases, AI variable and type renaming can help improve correctness without requiring more time. Figure 5 shows that DIRT improved correctness on BAPL tasks.

```

// Original
void buffer_append_path_len(buffer * restrict b, const
char * restrict a, size_t alen)
// Hex-Rays
void __fastcall buffer_append_path_len(__int64 a1, __BYTE
*a2, size_t a3)
// DIRTY
void __fastcall buffer_append_path_len(SSL *s, const
char *str, size_t n)

```

(a) Function signature



(b) Completion time

Fig. 6: Function signature and completion time for both `buffer_append_path_len` tasks. Here, DIRTY is able to reasonably recover some names and types (e.g., `char *str` and `size_t n`) but not all of them (e.g., `SSL` and `__fastcall` could be confusing to users).

Figure 6 shows the completion times for both groups. There is not a statistically significant difference in the completion time between these groups (the Hex-Rays group has a mean of 256.26 seconds and a standard deviation of 145.1, while the DIRTY group has a mean of 242.3 seconds and a standard deviation of 202.28. We performed a Welch Two Sample t-test and $p = 0.7204$), but DIRTY does have some impact on reasoning ability. For example, notice how the signatures shown in Figure 6a indicate that DIRTY’s choice of the type and name of the second argument suggest that it is used as an input string. This likely aided participant understanding of the BAPL tasks, which ask about the state of variables related to `str`.

Sometimes AI variable and type renaming correlates with people taking longer to reach the correct conclusion. Figure 7 shows the amount of time participants took to correctly answer AEEK question 2. DIRTY users took just over three and a half minutes longer to reach a correct answer than non-DIRTY users. From our experience, we suspect that this is for multiple reasons: first, DIRTY assigns the name `ret` to a variable that is never used for a return value, therefore users need to carefully scan and make sure they have spotted every `return` statement in the code. Second, the previously confusing statement on line 9 has become even more confusing. For a practiced reverse engineer, the pattern `8 * index + *(a1 + 8)` indicates an access of an element of an array inside a struct. DIRTY’s annotation does not make this clearer, in fact it muddies the water. Although the type

```

1 __int64 __fastcall array_extract_element_klen(__int64 a1,
__int64 a2, unsigned int a3) {
2     //...
3     //...
4     int index;
5     __int64 v7;
6     //...
7     if ( index < 0 )
8         return 0LL;
9     v7 = *(__QWORD *) (8LL * index + *(__QWORD *) (a1 + 8));
10    //...
11    return v7;
12 }

```

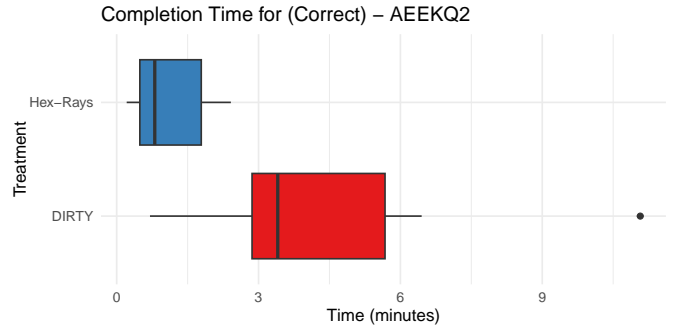
(a) Hex-Rays Output

```

1 char *__fastcall array_extract_element_klen(array_t_0 *
array, void *key, int index) {
2     //...
3     int indexa;
4     int ret;
5     char *next;
6     //...
7     if ( indexa < 0 )
8         return 0LL;
9     next=*(char**) (8LL*indexa + *(__QWORD*)&array->size);
10    //...
11    return next;
12 }

```

(b) DIRTY Output



(c) Completion time

Fig. 7: Functions and completion times for AEEK tasks. Here, users of DIRTY were slower despite the more meaningful variable and type names.

DIRTY predicts is generally “correct” (it predicts the type `array_t_0 *`, while the original code used the type `array *`), the layouts of these types are different. The `size` field of `array_t_0` should instead be a pointer to an array.

We were unable to find statistically significant evidence that machine-generated variable and type name recoveries reduce the time that users spend on reaching a correct conclusion. This is likely partially due to particularly confusing names that can lead users to take more time to reach the correct conclusion.

C. RQ3: Opinions

Our third research question asks about users’ perception of the usefulness of the renamings and retypings provided by DIRTY. To answer this, for each argument in a snippet, we asked participants to fill in the blank in the statement “The type and name of this argument _____ understanding.”

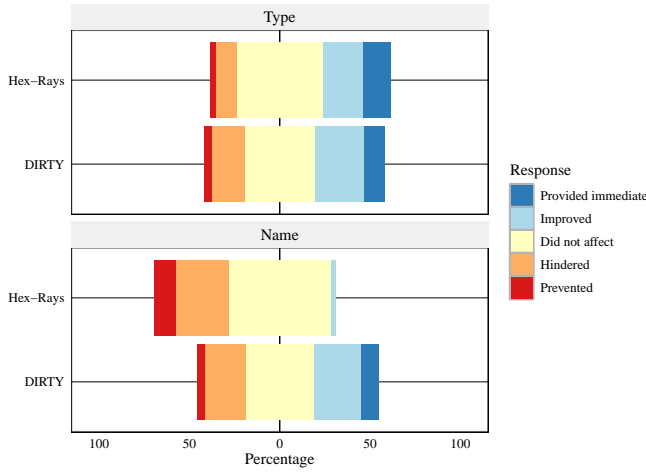


Fig. 8: Participants' overall opinion of types and names impacted their understanding of the code.

with “Prevented”, “Hindered”, “Did not affect”, “Improved”, or “Provided immediate”.

Figure 8 shows the participants' overall opinion of how the types and names from both DIRTY and Hex-Rays impacted their understanding of the code. We performed a Wilcoxon rank sum test with continuity correction on these results and found that in general, users prefer when variables are at least given some name, even if it does not agree with the actual use of the variable ($p = 5.072e - 14$, difference in location = 1). This result is not surprising: the names generated by Hex-Rays in this study are themselves rarely indicative of the purpose of the variable, except in cases like `ret` for a return variable or `src` and `dest` for source and destination variables. In general, tools like DIRTY are better at assigning names that carry more nuanced semantic meaning.

There is an inability to make an overall statistical claim about users' opinions about types, as there is no significant difference between the DIRTY and Hex-Rays groups ($p = 0.2734$). However, we did notice that the `twos_complement` snippet appears to be an outlier where DIRTY's suggestions are considered quite poor by users, where other snippets experienced an increase in preference over Hex-Rays, suggesting that reception is mixed overall. Thus, according to subjective participant ratings, DIRTY's type renaming did not significantly improve user understanding over the original Hex-Rays output overall.

We find that users universally prefer the **variable** names provided by DIRTY compared to the variable names provided by Hex-Rays. However, we find no statistically significant evidence that users perceive that the **types** provided by DIRTY are more helpful than the default types.

D. RQ4: Users' Perception vs. Performance

The fourth research question examines whether participants' perceptions of the helpfulness of DIRTY's generated variable names and types align with their actual performance on reverse engineering tasks. To address this, we analyzed the relationship between participants' ratings of DIRTY's output and their correctness on task-related questions. Participants rated the usefulness of DIRTY's variable and type renamings on a five-point Likert scale: (1) “Provided immediate,” (2) “Improved,” (3) “Did not affect,” (4) “Hindered,” and (5) “Prevented.”

To evaluate the alignment between perceived helpfulness and task performance, we conducted Spearman correlation tests on the Likert scores for both variable names and types against participants' correctness. The Spearman test was applied because it does not assume a normal distribution of data and it assesses monotonic relationships instead of linear relationships, making it suitable for our Likert-scale responses [38]. This approach allows us to more accurately evaluate the relationship between participants' subjective perceptions and their actual performance. For variable types, the Spearman correlation analysis indicated a significant positive correlation, with a p-value of 0.02459 and a ρ of 0.1035. Because lower Likert ratings indicate increased preference, this positive correlation indicates that as ratings get worse, correctness increases. The correlation for variable names was not statistically significant ($p = 0.6467$).

In alignment with our correlational analysis, we find that it is sometimes the case that users do not prefer the DIRTY-suggested types, despite them being helpful. For example, participants given the DIRTY version of the `twos_complement` function were more likely to answer questions correctly and answered correctly faster on average compared to participants given the Hex-Rays version. However, participants given the DIRTY version of the `twos_complement` function often rated the types in that function poorly, indicating that they “Hindered” or “Prevented” their understanding. Meanwhile, no participants who were given the Hex-Rays version of this function indicated that any types “Hindered” or “Prevented” their understanding despite their performance decreasing compared to the DIRTY version. These results suggest that participants' perceptions of the usefulness of DIRTY's annotations do not always consistently align with their actual performance on the tasks. Alternatively, users could have a higher expectation of a system that suggests type information, even when it actively improves their performance on a task. In either case, we find that user preference alone is not a reliable metric for measuring the usefulness of decompiler annotation tools.

We find that users' perceptions of the usefulness of DIRTY's annotations do not always align with their actual performance on reverse engineering tasks.

E. RQ5: Do Similarity Metrics Reflect Code Comprehension?

We next investigate the extent to which similarity metrics reflect how well machine-generated renamings improve a developer’s ability to comprehend code. Prevailing techniques in this area use accuracy (what percent of machine-generated names are exactly the same as the ground-truth names from the source code) [2, 11, 12], Levenshtein distance (the edit distance between machine-generated and ground truth names) [11]–[13, 39], and Jaccard Similarity (the ratio of the intersection of the sets of n-grams in machine generated and ground truth names to the union of those sets) [12, 40]. Note, however, that tokens like `size` and `length` are maximally distant according to these metrics, even though semantically they are quite similar. We therefore investigate additional similarity metrics from machine translation aimed at overcoming this shortcoming. Specifically, we include the BLEU score [41], codeBLEU [42], BERTScore F1 [43], and VarCLR [44] similarity metrics. BLEU score assesses n-gram overlap between generated and reference texts, providing an indicator of surface-level similarity between machine-generated and ground-truth names. CodeBLEU extends BLEU to include AST and dataflow information for comparing code, rather than natural language. Meanwhile, BERTScore F1 leverages embeddings from the BERT Transformer model to evaluate semantic similarity, providing a more nuanced measure that can capture meaning rather than exact n-gram overlap. Finally, VarCLR leverages embeddings from training on variable names specifically to encapsulate semantic meaning for this domain, similarly to BERTScore F1.

For each code snippet in our survey, we manually matched each variable and type name to its corresponding name in the associated original source code, and appended all the names into paired strings (one string for DIRTY-renamed variables and types, and one string for the original variables and types) so that entire renamed code snippets could be compared to the original source using BLEU score, Jaccard Similarity, Levenshtein distance, and BERTScore F1. For codeBLEU, we calculate similarity scores between lines of code containing analogous variable and type names so that we can capture the structure of the code around the names. Finally, since VarCLR is trained on variable names and not on sequences or sentences, it reports a separate similarity result for each variable name. However, we do not have correctness or timing measures associated with each individual variable name; we instead keep correctness and timing measures for entire functions with multiple variables and types. Thus, we compare matching variable names and types in isolation and average the resulting scores over each function to obtain an average VarCLR score for each function.

To explore how similarity metrics accurately reflect participants’ code comprehension, we examined the correlations between these metrics and two key performance indicators: time taken and correctness. Specifically, we used Spearman correlation tests to analyze the relationships between participants’ task performance and various similarity metrics, in-

cluding BLEU scores, Levenshtein distance, Jaccard similarity, and BERTScore F1. However, despite a significant positive correlation with timing and negative correlation with correctness, Levenshtein distance had high values² that indicate it may not be suitable for this context. To further understand the similarity between the original code and DIRTY code, 12 expert coders rated the variable names and types on their similarity with the original source code. These ratings were represented by Likert scores, and we average these scores to achieve our human evaluation score. These ratings had an ordinal Krippendorff’s α of 0.872, indicating substantial and reliable agreement [45]. We then conducted Spearman correlation tests on the similarity scores generated by each metric against task time and correctness. These results are depicted in Tables III and IV.

TABLE III: Correlation Between Similarity Metrics and Participant Time Taken on DIRTY Annotated Code Snippets















Similarity Metric	Correlation	ρ	p-value
BLEU		0.2568	0.0010
codeBLEU		0.2568	0.0010
Jaccard Similarity		0.5193	<0.0001
BERTScore F1		0.006	0.94
VarCLR		0.2568	0.0010
Human Evaluation (Variables)		0.2611	<0.0001
Human Evaluation (Types)		0.1065	0.0004542

TABLE IV: Correlation Between Similarity Metrics and Participant Correctness on DIRTY Annotated Code Snippets

Similarity Metric	Correlation	ρ	p-value
BLEU		0.0792	0.3437
codeBLEU		0.0792	0.3437
Jaccard Similarity		-0.2173	0.0086
BERTScore F1		0.2302	0.0053
VarCLR		0.0792	0.3437
Human Evaluation (Variables)		-0.1241	<0.0001
Human Evaluation (Types)		0.0517	0.1072

Our results show a significant positive correlation between task time and every similarity metric, save for BERTScore F1. This suggests that higher similarity to the original source code, as measured by these metrics, is associated with longer task times. Meanwhile, BERTScore F1 is only slightly and insignificantly positively correlated with time. Counterintuitively, All of the significant cases indicate that improved AI renaming performance correlates with developers to take more time to answer comprehension questions.

In terms of correctness, the correlations with similarity metrics were mixed. While BLEU score, codeBLEU, and VarCLR showed positive but insignificant correlations with correctness,

²On average, renamings in our code snippets exhibited Levenshtein distances greater than the total length of the renamed string, indicating that most characters had to be edited and some characters had to be added to produce the ground truth string.

BERTScore F1 had a significant positive correlation. Meanwhile, Jaccard Similarity and human evaluated Likert scores for both variable and type names exhibited significant negative correlations with correctness, indicating that improved AI renaming performance according to these metrics correlates with developers answering questions correctly less often.

These findings reinforce the conclusion from RQ4, indicating that participants’ perceptions of DIRTY’s usefulness do not consistently align with their performance. In these trials, BLEU score, codeBLEU, and VarCLR all have distinct scores, but very similar distributions, causing their correlational results to be nearly identical. Interestingly, Jaccard Similarity appears to correlate most closely with human similarity judgments in terms of both timing and correctness, suggesting it may be a more effective metric for capturing users’ comprehension in this context. However, our results also reveal that DIRTY’s annotations may actually hinder the participants’ performance, with better-performing similarity metrics often correlating with lower correctness and longer time. This suggests that while DIRTY enhances the perceived readability of code, it may inadvertently lead users to over-rely on its suggestions, resulting in more errors.

While previous evaluations of DIRTY primarily focused on system-level metrics like recovery rate and optimization, our human-centered study suggests that these commonly used metrics may not effectively reflect human code comprehension. This highlights the need for the research community to develop more robust metrics that accurately reflect comprehension and usability in real-world tasks.

V. DISCUSSION

In the results section, we discussed the implications of results found for each research question. Our results highlighted two key challenges in developing advanced decompiler techniques. First, as decompiler outputs become more readable, users tend to place greater trust in them, sometimes leading to errors they might not have made with more basic outputs. Second, there is a clear need for more refined metrics to evaluate the effectiveness of augmentation techniques accurately.

Our findings suggest that users of decompiler annotations frequently rely on the provided renamings, even when aware that the output may not be fully reliable. In some cases, this led users to misinterpret or over-rely on these annotations, resulting in incorrect conclusions. In other instances, participants were skeptical of the renamings, which may have increased cognitive load and extended the time required to reach the correct answer. These results indicate that, while annotations can be a useful aid, they may also introduce new challenges. Consequently, annotations should complement a reverse engineer’s direct analysis of the decompiled code, rather than serve as a primary tool for comprehension.

Our study also indicates a need for more appropriate metrics that account for the way variable and type names function

within the broader code context, as successful participants often reported relying on these contextual clues to make judgments about the code’s purpose and behavior. Existing similarity metrics primarily assess surface-level similarity by comparing generated names with those in ground-truth code. However, these metrics may overlook the functional relevance of annotations and the extent to which they assist in actual comprehension.

A refined metric would ideally consider not only the lexical match between the generated and original names but also their role in the code’s logic, helping evaluate whether annotations improve clarity in understanding data flow, control structures, or key functional relationships within code. For instance, a name recovery that identifies a pointer to a data structure may have limited impact if it fails to convey that structure’s role in the code. Similar approaches have successfully addressed this issue in the domain of code summarization. For example, Mastropaolo et al. leveraged contrastive learning techniques to evaluate how well a generated summary or annotation aligns with the underlying code semantics, providing a more reliable measure of comprehension impact than traditional similarity metrics [46]. Additionally, Zhang et al. applied eye-tracking technology to analyze developers’ visual attention during code comprehension tasks, identifying which tokens and structures are most cognitively salient [47]. Their findings enabled the development of evaluation metrics that weight tokens based on their actual importance to comprehension, offering a more human-centered assessment of annotation quality. By incorporating such contextual factors, decompiler annotation metrics could better align with human comprehension and usability, ultimately providing a more accurate assessment of the effectiveness of annotation techniques.

We recommend that future work in reverse engineering consider developing and validating metrics that capture these dimensions of code readability. Metrics that incorporate both structural and semantic relevance—possibly through embedding-based similarity measures or by mapping variable interactions across the code—could offer more insight into the practical value of decompiler augmentations. Such an approach could also reveal specific conditions under which annotations are most beneficial, supporting the design of augmentation methods that optimize accuracy and utility for software reverse engineers.

VI. THREATS TO VALIDITY

One threat to the validity of this study is that our technique was tested exclusively on open-source code snippets. Open-source software is generally designed with readability and logical structure in mind. This is rarely the case for real-world malware, which often uses techniques to conceal the functionality of their binaries. Additionally, our dataset, sampled from GitHub, is inherently biased toward well-known projects, and our experiments were limited to a finite set of code snippets. However, to address this limitation, we selected snippets from diverse domains and difficulty levels to ensure as broad a representation as possible within these constraints.

Our study also exclusively measured the outputs of the Hex-Rays decompiler and the DIRTY tool for variable and type name recovery, without considering alternatives such as Ghidra [7] or deGPT [13]. DIRTY is specifically tailored to Hex-Rays’ binary representation [2], while alternative tools, such as deGPT, perform additional augmentations outside the scope of our study, such as structural simplification and comment generation [13]. These augmentations would act as confounding variables for our experiment that would prevent us from establishing a causative link between variable and type renamings and developer comprehension. Other tools either do not perform as well as DIRTY or are not focused on both variable and type name recovery [11, 12].

Next, our study was constrained to four code snippets. While additional snippets could provide further insights and reduce the influence of outliers like AEEK, doing so would require additional participants to maintain statistical power. Given the difficulty of recruiting qualified reverse engineers, we opted for a study design that balanced feasibility with robust analysis. Future work could explore alternative designs, such as randomizing a larger pool of snippets per participant.

Furthermore, our participant pool consisted of self-selected reverse engineers from industry and academia rather than a fully randomized sample of a defined population. As a result, statistical significance should be interpreted with caution. While our statistical tests provide useful insights into observed trends, they should not be taken as definitive evidence of universal laws. Instead, they indicate what might be expected in a similar population under comparable conditions. Additionally, we acknowledge growing concerns about strict reliance on the arbitrary 0.05 p-value threshold and encourage future work to consider Bayesian or effect-size-based approaches for deeper statistical interpretation.

Additionally, our study compared decompiler output with AI-augmented annotations but did not include human-generated annotations. This could introduce biases, as participants’ trust or skepticism toward AI might influence their engagement with the annotations. Future work comparing AI annotations with the original source code and human-generated annotations could better isolate the impact of annotation provenance on comprehension.

Finally, our selected code snippets represent the entire workflow. The constraint that each code sample be self-contained on a single page is particularly limiting. So, direct integration with a decompiler would likely yield more comprehensive results.

VII. KEY TAKEAWAYS

This paper presents a novel human study designed to evaluate the impact of the Decompiled variable ReTYper (DIRTY) on performance in reverse engineering tasks. In this study, both professional and amateur reverse engineers were asked to answer questions that simulated real-world reverse engineering challenges. We recorded participants’ correctness and response times, and gathered qualitative feedback on their perceptions of the quality of DIRTY’s variable renaming, retyping, and the overall code structure.

Our findings challenge several assumptions about the effectiveness of machine-learning-based decompiler augmentation:

- **ML Performance Metrics Do Not Predict Comprehension Gains:** We find no significant correlation between commonly used similarity metrics (e.g., BLEU, Jaccard similarity) and actual improvements in program comprehension. This suggests that traditional evaluation methods may not fully capture the nuances of readability and cognitive load in reverse engineering.
- **Users Prefer AI-Augmented Outputs, Even Without Performance Gains:** Despite no measurable improvements in correctness or efficiency, participants reported a strong preference for AI-enhanced annotations, indicating a perceived value that is not reflected in objective comprehension metrics.
- **Enriched Decompiler Output Does Not Necessarily Improve Performance:** Our study found no significant evidence that AI-generated variable and type names improved task performance. In some cases, misleading annotations correlated with errors, highlighting the need for user skepticism when working with AI-augmented tools.
- **User Trust and Skepticism Play a Key Role:** Participants who fully trusted AI-generated annotations were more likely to answer incorrectly, while those who critically analyzed the code structure performed better. This underscores the importance of training reverse engineers to verify AI-generated annotations rather than accepting them at face value.
- **Future Research Should Refine Metrics and Baselines:** The lack of alignment between similarity metrics and human comprehension highlights the need for more robust evaluation techniques.

Overall, our results suggest that while AI-assisted tools may enhance usability, they do not necessarily translate into improved comprehension, calling for more research into how to bridge this gap effectively.

VIII. DATA AVAILABILITY

All data and scripts are available at https://osf.io/wusmj/?view_only=915e96b79ba44a178a2a8a95c9fa4c5c. This repository includes comprehensive documentation to facilitate replication and extension of our research.

IX. ETHICAL CONSIDERATIONS

All experiments described in this study were reviewed and exempted by our institution’s Institutional Review Board (IRB), confirming that the study posed minimal risk to participants. Recruitment targeted professionals and students with verified experience in reverse engineering, and participation was entirely voluntary and anonymous. No personally identifiable information was collected, and participants had the right to withdraw at any point without consequence. Participant recruitment and experience are detailed in Section III-E.

REFERENCES

- [1] Hex-Rays, “The hex-rays decompiler (v8.2.230124),” 2019. [Online]. Available: <https://hex-rays.com/decompiler>
- [2] Q. Chen, J. Lacomis, E. J. Schwartz, C. Le Goues, G. Neubig, and B. Vasilescu, “Augmenting decompiler output with learned variable names and types,” in *31st USENIX Security Symposium (USENIX Security 22)*, 2022, pp. 4327–4343.
- [3] L. Durlina, J. Kfoustek, and P. Zemek, “Psybot malware: A step-by-step decompilation case study,” in *2013 20th Working Conference on Reverse Engineering (WCORE)*. IEEE, 2013, pp. 449–456.
- [4] K. Yakdan, S. Eschweiler, E. Gerhards-Padilla, and M. Smith, “No more gotos: Decompilation using pattern-independent control-flow structuring and semantic-preserving transformations,” in *NDSS*. Citeseer, 2015.
- [5] K. Yakdan, S. Dechand, E. Gerhards-Padilla, and M. Smith, “Helping johnny to analyze malware: A usability-optimized decompiler and malware analysis user study,” in *2016 IEEE Symposium on Security and Privacy (SP)*. IEEE, 2016, pp. 158–177.
- [6] M. J. Van Emmerik, *Static single assignment for decompilation*. University of Queensland, 2007.
- [7] N. S. Agency, “The ghidra decompiler,” 2019. [Online]. Available: <https://ghidra-sre.org/>
- [8] Z. L. Basque, A. P. Bajaj, W. Gibbs, J. O’Kain, D. Miao, T. Bao, A. Doupe, Y. Shoshitaishvili, and R. Wang, “Ahoy sailor! there is no need to dream of c: A compiler-aware structuring algorithm for binary decompilation,” in *Proceedings of the USENIX Security Symposium*, 2024.
- [9] C. Cifuentes and K. J. Gough, “Decompilation of binary programs,” *Software: Practice and Experience*, vol. 25, no. 7, pp. 811–829, 1995.
- [10] E. Schulte, J. Ruchti, M. Noonan, D. Ciarletta, and A. Loginov, “Evolving exact decompilation,” in *Workshop on Binary Analysis Research (BAR)*, 2018.
- [11] J. Lacomis, P. Yin, E. Schwartz, M. Allamanis, C. Le Goues, G. Neubig, and B. Vasilescu, “Dire: A neural approach to decompiled identifier naming,” in *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 2019, pp. 628–639.
- [12] V. Nitin, A. Saieva, B. Ray, and G. Kaiser, “Direct: A transformer-based model for decompiled identifier renaming,” in *Proceedings of the 1st Workshop on Natural Language Processing for Programming (NLP4Prog 2021)*, 2021, pp. 48–57.
- [13] P. Hu, R. Liang, and K. Chen, “Degpt: Optimizing decompiler output with llm,” in *Proceedings 2024 Network and Distributed System Security Symposium (2024)*. <https://api.semanticscholar.org/CorpusID>, vol. 267622140, 2024.
- [14] K. Cho, “Learning phrase representations using rnn encoder-decoder for statistical machine translation,” *arXiv preprint arXiv:1406.1078*, 2014.
- [15] D. Lawrie, C. Morrell, H. Feild, and D. Binkley, “What’s in a name? a study of identifiers,” in *14th IEEE international conference on program comprehension (ICPC’06)*. IEEE, 2006, pp. 3–12.
- [16] D. Votipka, S. Rabin, K. Micinski, J. S. Foster, and M. L. Mazurek, “An observational investigation of reverse {Engineers’} processes,” in *29th USENIX Security Symposium (USENIX Security 20)*, 2020, pp. 1875–1892.
- [17] S. Stapleton, Y. Gambhir, A. LeClair, Z. Eberhart, W. Weimer, K. Leach, and Y. Huang, “A human study of comprehension and code summarization,” in *Proceedings of the 28th International Conference on Program Comprehension*, 2020, pp. 2–13.
- [18] P. Tonella, M. Torchiano, B. Du Bois, and T. Systä, “Empirical studies in reverse engineering: state of the art and future trends,” *Empirical Software Engineering*, vol. 12, pp. 551–571, 2007.
- [19] J. Siegmund, N. Peitek, C. Parnin, S. Apel, J. Hofmeister, C. Kästner, A. Begel, A. Bethmann, and A. Brechmann, “Measuring neural efficiency of program comprehension,” in *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 140–150.
- [20] J. Siegmund, “Program comprehension: Past, present, and future,” in *2016 IEEE 23rd international conference on software analysis, evolution, and reengineering (SANER)*, vol. 5. IEEE, 2016, pp. 13–20.
- [21] A. Von Mayrhauser and A. M. Vans, “Program comprehension during software maintenance and evolution,” *Computer*, vol. 28, no. 8, pp. 44–55, 1995.
- [22] A. J. Ko, B. A. Myers, M. J. Coblenz, and H. H. Aung, “An exploratory study of how developers seek, relate, and collect relevant information during software maintenance tasks,” *IEEE Transactions on software engineering*, vol. 32, no. 12, pp. 971–987, 2006.
- [23] N. Pennington, “Stimulus structures and mental representations in expert comprehension of computer programs,” *Cognitive psychology*, vol. 19, no. 3, pp. 295–341, 1987.
- [24] V. Arunachalam and W. Sasso, “Cognitive processes in program comprehension: An empirical analysis in the context of software reengineering,” *Journal of Systems and Software*, vol. 34, no. 3, pp. 177–189, 1996.
- [25] F. D  tienne, “Expert programming knowledge: A schema based approach,” *Psychology of Programming/Academic Press*, 1990.
- [26] Y. Cao, R. Zhang, R. Liang, and K. Chen, “Evaluating the effectiveness of decompilers,” in *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2024, pp. 491–502.
- [27] Z. Karas, A. Bansal, Y. Zhang, T. Li, C. McMillan, and Y. Huang, “A tale of two comprehensions? analyzing student programmer attention during code summarization,” *ACM Transactions on Software Engineering and Methodology*, 2024.
- [28] D. Roy, S. Fakhoury, and V. Arnaoudova, “Reassessing automatic evaluation metrics for code summarization tasks,” in *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2021, pp. 1105–1116.
- [29] E. Shi, Y. Wang, L. Du, J. Chen, S. Han, H. Zhang, D. Zhang, and H. Sun, “On the evaluation of neural code summarization,” in *Proceedings of the 44th international conference on software engineering*, 2022, pp. 1597–1608.
- [30] A. Jbara and D. G. Feitelson, “On the effect of code regularity on comprehension,” in *Proceedings of the 22nd international conference on program comprehension*, 2014, pp. 189–200.
- [31] Z. P. Fry, B. Landau, and W. Weimer, “A human study of patch maintainability,” in *Proceedings of the 2012 International Symposium on Software Testing and Analysis*, 2012, pp. 177–187.
- [32] J. Sillito, G. C. Murphy, and K. De Volder, “Questions programmers ask during software evolution tasks,” in *Proceedings of the 14th ACM SIGSOFT international symposium on Foundations of software engineering*, 2006, pp. 23–34.
- [33] M. J. Pacione, M. Roper, and M. Wood, “A novel software visualisation model to support software comprehension,” in *11th working conference on reverse engineering*. IEEE, 2004, pp. 70–79.
- [34] A. Jaffe, J. Lacomis, E. J. Schwartz, C. Le Goues, and B. Vasilescu, “Meaningful variable names for decompiled code: A machine translation approach,” in *Proceedings of the 26th Conference on Program Comprehension*, 2018, pp. 20–30.
- [35] A. Gelman, *Data analysis using regression and multilevel/hierarchical models*. Cambridge university press, 2007.
- [36] S. Nakagawa, P. C. Johnson, and H. Schielzeth, “The coefficient of determination r^2 and intra-class correlation coefficient from generalized linear mixed-effects models revisited and expanded,” *Journal of the Royal Society Interface*, vol. 14, no. 134, p. 20170213, 2017.
- [37] K. Charmaz, *Constructing grounded theory: A practical guide through qualitative analysis*. sage, 2006.
- [38] J. C. De Winter, S. D. Gosling, and J. Potter, “Comparing the pearson and spearman correlation coefficients across distributions and sample sizes: A tutorial using simulations and empirical data,” *Psychological methods*, vol. 21, no. 3, p. 273, 2016.
- [39] L. Yujian and L. Bo, “A normalized levenshtein distance metric,” *IEEE transactions on pattern analysis and machine intelligence*, vol. 29, no. 6, pp. 1091–1095, 2007.
- [40] S. Niwattanakul, J. Singthongchai, E. Naenudorn, and S. Wanapu, “Using of jaccard coefficient for keywords similarity,” in *Proceedings of the international multicongference of engineers and computer scientists*, vol. 1, no. 6, 2013, pp. 380–384.
- [41] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, “Bleu: a method for automatic evaluation of machine translation,” in *Proceedings of the 40th annual meeting of the Association for Computational Linguistics*, 2002, pp. 311–318.
- [42] S. Ren, D. Guo, S. Lu, L. Zhou, S. Liu, D. Tang, N. Sundaresan, M. Zhou, A. Blanco, and S. Ma, “Codebleu: a method for automatic evaluation of code synthesis,” *arXiv preprint arXiv:2009.10297*, 2020.
- [43] T. Zhang, V. Kishore, F. Wu, K. Q. Weinberger, and Y. Artzi, “Bertscore: Evaluating text generation with bert,” *arXiv preprint arXiv:1904.09675*, 2019.
- [44] Q. Chen, J. Lacomis, E. J. Schwartz, G. Neubig, B. Vasilescu, and C. Le Goues, “VarCLR: Variable semantic representation pre-training via contrastive learning,” in *International Conference on Software Engineering*, ser. ICSE ’22, 2022.

- [45] K. L. Gwet, "On the krippendorff's alpha coefficient," *Manuscript submitted for publication*. Retrieved October, vol. 2, no. 2011, p. 2011, 2011.
- [46] Y. Zhang, J. Li, Z. Karas, A. Bansal, T. J.-J. Li, C. McMillan, K. Leach, and Y. Huang, "Eyetrans: Merging human and machine attention for neural code summarization," *Proceedings of the ACM on Software Engineering*, vol. 1, no. FSE, pp. 115–136, 2024.
- [47] A. Mastropaolo, M. Ciniselli, M. Di Penta, and G. Bavota, "Evaluating code summarization techniques: A new metric and an empirical characterization," in *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, 2024, pp. 1–13.