R2I: A Relative Readability Metric for Decompiled Code

HAEUN EOM, Sungkyunkwan University, South Korea
DOHEE KIM, Sungkyunkwan University, South Korea
SORI LIM, Sungkyunkwan University, South Korea
HYUNGJOON KOO*, Sungkyunkwan University, South Korea
SUNGJAE HWANG*, Sungkyunkwan University, South Korea

Decompilation is a process of converting a low-level machine code snippet back into a high-level programming language such as C. It serves as a basis to aid reverse engineers in comprehending the contextual semantics of the code. In this respect, commercial decompilers like Hex-Rays have made significant strides in improving the readability of decompiled code over time. While previous work has proposed the metrics for assessing the readability of source code, including identifiers, variable names, function names, and comments, those metrics are unsuitable for measuring the readability of decompiled code primarily due to i) the lack of rich semantic information in the source and ii) the presence of erroneous syntax or inappropriate expressions. In response, to the best of our knowledge, this work first introduces R2I, the Relative Readability Index, a specialized metric tailored to evaluate decompiled code in a relative context quantitatively. In essence, R2I can be computed by i) taking code snippets across different decompilers as input and ii) extracting pre-defined features from an abstract syntax tree. For the robustness of R2I, we thoroughly investigate the enhancement efforts made by (non-)commercial decompilers and academic research to promote code readability, identifying 31 features to yield a reliable index collectively. Besides, we conducted a user survey to capture subjective factors such as one's coding styles and preferences. Our empirical experiments demonstrate that R2I is a versatile metric capable of representing the relative quality of decompiled code (e.g., obfuscation, decompiler updates) and being well aligned with human perception in our survey.

CCS Concepts: • **General and reference** → **Metrics**; **Measurement**; • **Software and its engineering** → *Software development techniques.*

Additional Key Words and Phrases: Code Readability, Code Metric, Decompiled Code, Decompiler

ACM Reference Format:
Haeun Eom, Dohee Kim, Sori Lim, Hyungjoon Koo, and Sungjae Hwang. 2024. R2I: A Relative Readability Metric for Decompiled Code. *Proc. ACM Softw. Eng.* 1, FSE, Article 18 (July 2024), 23 pages. https://doi.org/10.1145/3643744

1 INTRODUCTION

Today, the software is commonly distributed in the form of (stripped) executable binaries, which often lacks most of the high-level information available in the source code. A wide spectrum of scenarios can encounter application bugs [Bessey et al. 2010], program crashes, security vulnerabilities [Chandramohan et al. 2016; David et al. 2016; Eschweiler et al. 2016; Gao et al. 2018], or the

*Corresponding author.

Authors' addresses: Haeun Eom, Sungkyunkwan University, Suwon, South Korea, e0mh4@g.skku.edu; Dohee Kim, Sungkyunkwan University, Suwon, South Korea, dohui7557@g.skku.edu; Sori Lim, Sungkyunkwan University, Suwon, South Korea, drwho@g.skku.edu; Hyungjoon Koo, Sungkyunkwan University, Suwon, South Korea, kevin.koo@skku.edu; Sungjae Hwang, Sungkyunkwan University, Suwon, South Korea, sungjaeh@skku.edu.

This work is licensed under a Creative Commons Attribution 4.0 International License.
© 2024 Copyright held by the owner/author(s).
ACM 2994-970X/2024/7-ART18
https://doi.org/10.1145/3643744

need for malware analysis [Bruschi et al. 2006; Cesare et al. 2013; Yakdan et al. 2016]. To handle such cases, comprehending the inner workings of a binary (*i.e.*, binary reverse engineering; binary reversing) becomes essential. However, binary reversing without appropriate tools is exceptionally challenging because contextual meanings must be inferred from machine-interpretable code alone. Decompilation stands as a fundamental technique for gaining insights because it involves converting low-level machine code into a (C-like) high-level programming language. A decompiler is a program designed for the decompilation, comprising a set of components, including disassembly, control, and data flow analyses, type inference, lifting to intermediate representations, and varying optimizations, collectively producing a decompiled code.

Reversing practitioners largely rely on the quality of a decompiler's output [Vector35 2023b], namely the decompiled code, to comprehend a binary effectively. Hence, the quality of a decompiler-producing code is crucial. In this context, commercial decompilers like Hex-Rays [Hex-Rays 2023b] and Binary Ninja [Vector35 2023a] have been consistently enhancing a decompilation output [Hex-Rays 2023a; Wiens et al. 2023], making it more accurate and readable. In a similar vein, open-source decompilers like Ghidra [NSA 2023b], angr [Shoshitaishvili et al. 2016], and RetDec [Křoustek et al. 2017] put considerable efforts on improving the readability of their outputs. Furthermore, a plethora of study [Chen et al. 2010; Enders et al. 2023; Schulte et al. 2018; Yakdan et al. 2015] focus on improving the readability of decompiled code by reducing the gap between the original code and decompiled code.

To evaluate the readability of a given code, one needs a metric that can quantitatively measure it. Early work like Buse et al. [Buse and Weimer 2008] and Posnett et al. [Posnett et al. 2011] propose the readability metrics for source code, considering varying features like comments, function names, variable names, and the number of identifiers. Later, Scalabrino et al. [Scalabrino et al. 2018, 2016] attempt to include meaningful textual features of the source code. However, applying the aforementioned metrics for source code to decompiled code is unsuitable because ① decompiler-producing code does not contain rich semantic information that is available in the source, and ② the code may introduce erroneous syntax or inappropriate expressions. Until today, the readability metrics for decompiled code [Wirtanen 2022] is absent, necessitating a dedicated readability metric for decompiled code.

In this work, to the best of our knowledge, we first devise the readability metrics tailored to decompiled code, dubbed R2I (relative readability index). Simply put, given multiple (*i.e.*, two or more) decompiler-generating code snippets, the metric is designed for producing a relatively quantitative value, ranging from 0 to 1. For handy computation, we develop a full-fledged readability assessment system for decompiled code with R2I, which consists of the following four phases: ① taking a binary as input and producing a decompiled code from a list of decompilers for comparison; ② constructing an abstract syntax tree (AST); ③ extracting pre-defined features from the AST; and ④ calculating R2I with feature weights. To define appropriate features, we thoroughly investigate the improvement efforts to promote decompiled code readability from both existing decompilers and academic work, identifying 31 features that assist to obtain a reliable and robust index. Besides, as measuring code readability inherently entails subjective criteria such as one's coding styles or preferences, we conduct a user survey (22 participants), thereby empirically arranging feature weights. Note that defining those features and their weights for R2I is a one-time processing.

To demonstrate the practicality and the usefulness of R2I, we prepare 5,305 functions that have been successfully decompiled from six decompilers of our choice (*i.e.*, Hex-Rays [Hex-Rays 2023b], Binary Ninja [Vector35 2023a], Ghidra [NSA 2023b], Radare2 [Radare 2023; Wargio et al. 2023a] [1], RetDec [Křoustek et al. 2017], angr [Shoshitaishvili et al. 2016]). We confirm that the results of

[1]Radare2 has been implemented as the r2dec-plugin [Wargio et al. 2023a], which we refer to as Radare2 in our paper.

R2I with our dataset are well aligned with those of our survey. Additionally, we applied R2I to the outputs from different versions of the same decompiler and from obfuscated binaries, adequately reflecting the differences.

The contributions of our paper are summarized as follows.

- We present R2I, the first Relative Readability Index for a decompiler-producing code snippet, which is capable of measuring quantitative code readability (*i.e.*, [0,1] range).
- We define 31 features for R2I comprehensively by considering readability-enhancing efforts from existing decompilers and previous studies to promote source code readability. We also conducted a user survey for the features that entail subjective criteria.
- We develop an end-to-end system to evaluate R2I, demonstrating its usefulness and practicality with varying experiments (*e.g.*, obfuscated binary). We make our system publicly available (§11) for future research in the realm of decompiled code readability.

2 BACKGROUND

This section describes the background of decompilers and the existing readability metrics.

Decompilation Utilities. A decompilation process [Wiki 2023] aims to convert a low-level machine code snippet (*e.g.*, assembly function) or bytecode (*e.g.*, Java, C#, Python) back into a high-level human-readable one, which aids reverse engineers in comprehending the underlying code semantics [Emmerik and James 2007]. Such a process can serve various purposes, such as ① understanding or recovering legacy code in case that the source is unavailable [Park et al. 2023], ② analyzing potential security vulnerabilities [Shejwal et al. 2023], ③ identifying unexpected system behaviors like bugs and malware [Yakdan et al. 2016], ④ facilitating software porting between environments or platforms [Troshina et al. 2010], and ⑤ investigating the legality pertaining to intellectual property violations or software copyright infringement [Melling and Zeidman 2012].

Decompilers. A decompiler is a program tailored to decompilation, producing a high-level programming language like C and C++. Although decompilation predominately relies on the design of a decompiler engine, in general, it goes through a series of (complex) reverse steps of compilation in joint, including ① disassembly, ② control flow analyses, ③ data flow analyses, ④ intermediate representation (IR) generation, ⑤ data type inference, ⑥ code semantic analysis and construction, and ⑦ built-in optimizations for better readability. Theoretically, a complete decompiled code includes function parameters, variable types, invocations to another function with suitable parameters, and a return value, which renders the recovered code readable and informative (putting its accuracy aside). However, the product of each decompiler may significantly vary depending on its own unique analysis algorithms and peculiar optimization strategies. To exemplify, Figure 8b and Figure 8d present the outputs from two distinct decompilers, Hex-Rays [Hex-Rays 2023b] and Radare2 [Radare 2023; Wargio et al. 2023a]. Both follow C-like syntax, but the latter is close to an assembly-like code because it directly utilizes a register as a variable name (*e.g.*, `rax`, `esi`). This work proposes a *relative readability metric* that can quantitatively compare a decompiled code snippet with others.

Source Readability Metrics. Buse et al. [Buse and Weimer 2008] and Posnett et al. [Posnett et al. 2011] propose the assessment metrics for the readability of source code. They define varying features, including comments, number of identifiers, function names, and variable names, which help help assess the source code's readability level. Dorn [Dorn 2012] presents a generalizable model of code readability based on a large-scale of human study (*i.e.*, 5,000 participants) by harnessing all tokens as identifiers for capturing rich semantics. However, those metrics are inapplicable to a decompiled code snippet because of the absence of high-level information. For example, as the original function name is unavailable in a stripped binary, it is common to name after a

memory address (*e.g.*, `0x40EDD0` → `sub_40EDD0`). Likewise, variables and parameters are named with meaningless labels like `v13` and `a1`. Besides, decompilers may generate syntactically invalid code, making the decompiled code less readable. Unlike prior work, we focus on the readability metric for *decompiled codes.*

3 DECOMPILED CODE READABILITY

This section briefly outlines the problem, the motivation, the objective, and the approach of a readability metric for decompiled code, then delineates how we define the features of the metric.

3.1 Problem Statement and Our Approaches

Problem and Motivation. In general, decompiled code reconstructed from an executable binary considerably differs from source code written in a programming language. Namely, the local features of previous metrics for source code readability [Buse and Weimer 2008; Posnett et al. 2011] cannot be directly applicable mainly because ① decompiled code has been generated by a decompiler (rather than a human programmer): *e.g.*, keywords may be decompiler-specific; indentations are unintentional; ② semantic information has been disappeared:*e.g.*, no comment remains; function and variable names are randomly labeled; and ③ seemingly high-level code (*i.e.*, C-like language) may not obey a valid program expression or statement. However, to the best of our knowledge, little has been conducted on a readability metric for assessing decompiled code quality.

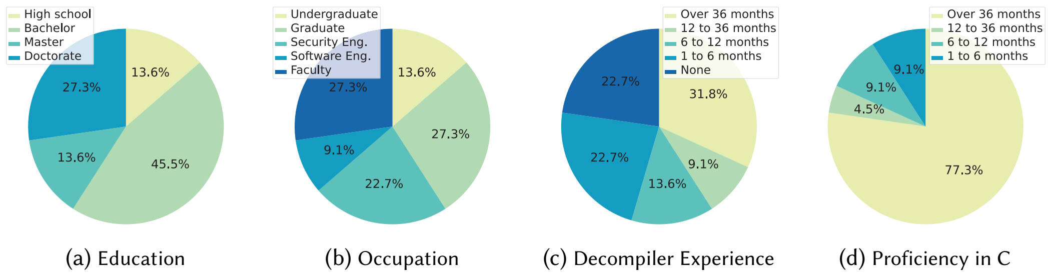
Goal and Scope. We aim to devise a dedicated metric that allows one to relatively and quantitatively evaluate decompiled code. In essence, our metric computes a relative readability index using two or C-like decompiled codes across different decompilers, expressing that one is more or less readable than another. We assume that a stripped binary has been written in a compiled language such as C, C++, Golang [Meyerson 2014], Rustlang [Matsakis and Klock 2014], or Nimlang [Rumpf 2022] [2]. We design the metric to be calculated based on a (decompiler-recognized) function because it represents a chunk of logical abstraction that performs an independent task. Identifying a function in a binary is orthogonal to our work.

Our Approaches. Measuring code readability inherently involves subjective criteria because it may rely on one's coding style or preference. As such, we collect features to yield a reliable metric as follows. First, we thoroughly investigate prior efforts on improving decompiled code readability from existing works [Hex-Rays 2023a; Křoustek et al. 2023; NSA 2023a; Shoshitaishvili et al. 2023a; Wargio et al. 2023a; Wiens et al. 2023], which aids in defining a set of features. For instance, one of the industry-leading decompilers, Hex-Rays [Hex-Rays 2023b], constantly enhances the quality of decompiled code from varying aspects (Table 2). Besides, we study academic contributions to promote code readability [Buse and Weimer 2008; Chen et al. 2010; Dorn 2012; Enders et al. 2023; Posnett et al. 2011; Scalabrino et al. 2018, 2016; Schulte et al. 2018; Yakdan et al. 2015]. Second, we explore varying syntactic errors that hamper building an AST, extracting additional features affecting code readability. Third, we conduct a user study that considers empirical preferences, particularly when reading decompiler-producing code, assisting in adjusting relative weights between features. It is worth noting that a relative readability metric fits into our goal under the assumption that no original code is available (*i.e.*, no absolute accuracy).

3.2 User Survey

We conducted two user surveys to account for subjective factors of the above features like code styles, which play a crucial role in understanding coding preferences. The first survey served as a

[2]Note that we have yet tested on virtual-machine-based (*e.g.*, Java [Oracle 2023]) or interpreted languages (*e.g.*, Python [Van Rossum et al. 2022], Ruby [Matsumoto 2022]).



(a) Education (b) Occupation (c) Decompiler Experience (d) Proficiency in C

Fig. 1. Participants' demographics in our user survey. We consider various factors including education, occupation, experience on decompilers, and C language proficiency.

Table 1. Summary of survey questions. Note that we request consent to collect private information from a participant, including a name, phone number, and email. Additionally, we explicitly specify the purpose, items, and retention period of the collected information.

(1) Participant Background
  - *What is the highest level of education you have completed?*
    `High school, Bachelor, Master, Doctorate`
  - *What is your occupation?*
    `Undergraduate, Graduate, Security engineer, Software engineer, Faculty member`
    - Describe the main area of your expertise and career (in the case of an engineer or faculty)
    - Describe the years of your experience.
  - *How long have you been programming in C?*
    `None, Less than six months, 6 - 12 months, 12 - 36 months, More than 36 months`
  - *Which decompiler do you mainly use?*
  - *How long have you been using a decompiler?*
    `None, Less than six months, 6 - 12 months, 12 - 36 months, More than 36 months`

(2) User Preference & Conflicting Features
  - *Which of the two code snippets from the same source exhibits better readability?*
    - `longer if condition and line length, shorter code lines, less nested if` vs `shorter if condition and line length, longer code lines, more nested if`
    - `for` vs `while`
    - `while` vs `do-while`
    - `switch` vs `if`
    - `if` vs `ternary`
    - `!strcmp` vs `strcmp` in condition

(3) Comparison of the decompiled outputs generated by six decompilers
  - Choose the most readable code excerpt.
  - Choose the least readable code excerpt.

pilot study aimed at refining the appropriate design of the survey questions while the second focuses on assessing how well the readability metric for decompiled code aligns with user perceptions.

Participants. We recruited 22 participants from various sources: vulnerability discovery organizations, universities, and IT industries. Figure 1a depicts their educational backgrounds, ranging from high school diplomas to doctorate degrees. Figure 1b represents varying work experiences: 27.3% are computer science professors engaged in the security field, 9.1% work as security engineers specializing in areas such as binary analysis and vulnerability discovery, 22.7% are software engineers, developing embedded software such as automotive and smartphones, while 13.6% and 27.3% are undergraduate and graduate computer science students, respectively. Additionally, as shown

Table 2. Summary of the 31 features for decompiled code readability. The symbols (↑, ↓) represent that higher and lower values for better readability. The origin column indicates that each feature has been carefully elected from the existing [D]ecompiler efforts to enhance code readability [Hex-Rays 2023a; Křoustek et al. 2023; NSA 2023a; Shoshitaishvili et al. 2023a; Wargio et al. 2023a; Wiens et al. 2023], [P]rior work [Buse and Weimer 2008; Chen et al. 2010; Dorn 2012; Enders et al. 2023; Posnett et al. 2011; Scalabrino et al. 2018, 2016; Schulte et al. 2018; Yakdan et al. 2015], or [S]yntatic errors. Note that (✳) means a feature that leads to indirect improvement. A,B,G,H, R2, and RD represent Angr [Shoshitaishvili et al. 2016], Binary ninja [Vector35 2023a], Ghidra [NSA 2023b], Hex-Rays [Hex-Rays 2023b], Radare2 [Radare 2023; Wargio et al. 2023a], and RetDec [Křoustek et al. 2017], respectively.

Class	No	Origin	Feature	Description	Decompiler
Code Quality	F1	D/P	[↑] # of array detections	Improvement on identifying arrays and structures	B, G, H, RD
	F2	D/P	[↓] # of operators	Improvement on identifying 64-bit variables and bit operations	H
				Improvement on identifying compiler idioms (✳)	H
				Simplification of bit arithmetic expressions	A, H, RD
				Simplification of multi-part boolean expressions	G
				Optimization of modulo/remainder calculations	G
	F3	D/P	[↓] # of comma operators in conditions	Elimination of comma operators in conditions	G, H
	F4	D	[↓] # of goto statements	Elimination of `goto` statements to reduce redundant call invocations	H
				Reduction of `goto` statements	A, H
	F5	D	[↓] # of inline assembly	Improvement on identifying floating points, SSE operations, and intrinsic functions	H
	F6	D	[↓] # of missing conditions	Clarification on conditions: *e.g.*, `while(true)` → `while(condition)`	H, RD
				Clarification on conditions: *e.g.*, `for(i=0; ;++i)` → `for(i=0;condition;++i)`	H
	F7	D/P	[↓] # of nested casting operators	Elimination of unnecessary casts	G, RD
				Improvement on identifying 64bit variables and bit operations (✳)	H
				Improvement on identifying data type through TEK/KPCR reference support (✳)	H
				Improvement on identifying `typedef` relationships between data and types	G
	F8	D	[↓] # of references/dereferences	Improvement on identifying arrays and structures (✳)	B, G, H, RD
				Elimination of dereferences of array arguments & addresses	A, RD
	F9	D	[↓] # of unnecessary `goto` labels	Elimination of unnecessary `goto` labels	A, RD
	F10	D/P	[↓] # of variables	Aggressive variable elimination and propagation	A, H, RD
User Preference	F11	D	[↑] Ratio of conditional statements	Preference of `switch` over `if` statements	A, B, RD
				Preference of ternary operators over `if` to reduce the line of code	A
	F12	D	[↑] Ratio of loop statements	Preference of `for` over `while` or `do-while` loops	A, H, RD
				Preference of `while` over `do-while` loops	H
	F13	D	[↑] Ratio of `!strcmp` in conditions	Preference of `!strcmp` over `strcmp` in conditions	H
Conflicting Features	F14	D	[↓] Max # of conditions in if statements	Increase `if` statements to reduce the length of a line	H
	F15	D/P	[↓] Max # of nested if statements	Increase the length of `if` conditions to reduce nested `if` statements	A, H, RD
	F16	D/P	[↓] Max length of a line	Reduction of the length of a line by decreasing cast and bitwise operators (✳)	A, G, H, RD
	F17	D/P	[↓] Line of code	Elimination of LLVM intrinsics: *e.g.*, `llvm.ctpop.*` (✳)	RD
				Improvement on identifying inlined and built-in functions (✳)	B, H
				Reduction of line of code by removing dead or redundant code	A, B, G, RD
Erroneous Syntax	F18	S	[↓] # of multiple types	A data type has been unclearly defined (*e.g.*, unknown, undefined, or multiple type)	A, G, H
	F19	S	[↓] # of invalid `goto` statements	A `goto` statement has been erroneously labelled	B, G, R2
	F20	S	[↓] # of invalid `do-while` loops	A `do-while` statement has been inaccurate (*e.g.*, boundary)	R2
	F21	S	[↓] # of invalid function calls	A function invocation has been invalid or unclear	R2
	F22	S	[↓] # of remaining IRs	A decompiled code contains intermediate representations	A
	F23	S	[↓] # of unimplemented parts	A code decompilation has been explicitly unimplemented	B
	F24	S	[↓] # of unknown expressions	A conditional expression or variable has been undefined or unclear	A, R2
	F25	S	[↓] # of invalid argument	A function argument has been invalid	RD
	F26	S	[↓] # of unknown operators	A conditional expression contains an unknown operator	R2
General Features	F27	P	[↓] # of tokens	Total number of tokens	N/A
	F28	P	[↓] # of conditions	Total number of conditional statements	N/A
	F29	P	[↓] # of loops	Total number of loop statements	N/A
	F30	P	[↓] # of assignments	Total number of assignments	N/A
	F31	P	[↓] Max # of nested loop statements	Maximum number of nested loop statements	N/A

in Figure 1c, 77.3% of participants have decompiler experiences, with 9.1% having 1 to 3 years of experience and 31.8% over three years, mainly for security research and vulnerability analysis. Others have used decompilers for educational purposes, such as class projects. Among the various decompilers, 76.5% have experience with Hex-Rays [Hex-Rays 2023b], 11.8% with Ghidra [NSA 2023b], and another 11.8% with JD-GUI [Dupuy 2023]. All participants confirmed their proficiency in the C programming language, while 77.3% utilize it in their work (Figure 1d).

Pilot Study. Prior to the primary survey, we conducted a pilot survey involving five participants, two of whom possess significant expertise in binary analysis and extensive experience with decompilers. The survey mainly includes the outputs from the six decompilers of our choice, asking their rankings (*i.e.*, 1st, 2nd, ..., 6th). The pilot survey yielded valuable insights and feedback: ① determining the rankings of all decompiled codes is challenging; ② the examples are a little too long to read the codes; ③ a direct code-length comparison is ambiguous due to the absence of line numbers. In response, we ① simplify the ranking questions for participants to select the most and least comprehensible code alone from the provided examples; ② select a set of brief example code snippets for participants to discern code features clearly; ③ insert line numbers in decompiled code for better readability.

Survey Design. The main survey presents the pairs of decompiled code, each exemplifying specific features from the *User Preference* and *Conflicting Features* categories in Table 3. Participants were instructed to choose one of the two options based on their preferences. To ensure clarity, comprehensive information about the features represented by each decompiled code was provided. Furthermore, participants were asked to provide supplementary comments in case they selected a particular option. Subsequently, we selected decompiled code from six decompilers (as detailed in §6) and requested the participants to identify the best and the worst decompiled code in terms of readability. The six decompiled code excerpts were presented side by side, enabling the participants to clearly observe the code differences between each decompiler. We ensure the chosen decompiled codes to encompass all the features that are taken into account for our metric. Table 1 provides some example questions from our survey. In appreciation for their time and efforts, we offered a $23 gift card (in USD) to the participants.

3.3 Features for Decompiled Code Readability

Feature Exploration. Table 2 summarizes the 31 factors (*i.e.*, features) that impact on decompiled code readability. The features originate from the improvements of existing decompilers, prior work, and syntactic errors with the following considerations. First, most decompilers continuously strive to improve the readability of their outputs, producing concise but accurate expressions by recovering the original program logic. Hence, we analyze the decompiler's efforts to enhance code readability, including the official website [Hex-Rays 2023a; Wiens et al. 2023] and change logs [Avast 2023; NSA 2023a; Shoshitaishvili et al. 2023b; Wargio et al. 2023b]. Furthermore, we investigate decompilers' source code [Křoustek et al. 2023; Shoshitaishvili et al. 2023a; Wargio et al. 2023a] (in the case of open source), specifically focusing on the optimization module to improve readability levels. Our findings show that Hex-Rays [Hex-Rays 2023a] has the most substantial number of improvements, totaling 105, followed by Binary Ninja [Wiens et al. 2023], Ghidra [NSA 2023a], RetDec [Křoustek et al. 2023], and Angr [Shoshitaishvili et al. 2023a] with 18, 51, 43, and 21 enhancements, respectively. Second, we study prior work on code readability metrics [Buse and Weimer 2008; Chen et al. 2010; Dorn 2012; Enders et al. 2023; Posnett et al. 2011; Scalabrino et al. 2018, 2016; Schulte et al. 2018; Yakdan et al. 2015]. Third, we extract peculiar features that harm readability (*i.e.*, by increasing ambiguity and obscurity) while correcting syntax errors, which have not been previously taken into account in source code. Of all, as in Table 2, we define the 31 features in total, including 17 features (F1-F17) from decompilers' efforts, 13 features (F[1-3,7,10,15-17,27-31]) from previous studies, and nine features (F18-26) from erroneous syntax at decompiled code.

Feature Categorization. Over the 31 features from various sources in Table 3, we largely categorize them into five feature groups: code quality, user preference, conflicting features, erroneous syntax, and general features. First, the ten features are grouped to the *Code Quality* class, which directly improves the readability of decompiled code, including simplifying control flows, eliminating

```
1  //Hex-Rays
2  // F16 vs F17
3  return sub_40EDD0(
4    v9,
5    v8,
6    *(&off_419580[8 * (unsigned int)(v7 + v13)]
7    + 4 * v6
8    + 2 * (unsigned __int8)byte_6256EF
9    + (unsigned __int8)byte_6256B4));
10
11 //Angr
12 unsigned long | unsigned int v4; // F18
13 xmm1<8> = Conv(32->128, Load(addr=rdx<8>, size=4,
   ↪  endness=Iend_LE)); // F22
14 if ((char)((unsigned long long)(unsigned int)(BinaryOp
   ↪  CmpF & 69) >> 6)) & 1) != 1) // F7, F22
15 {
16   v3 = v2[1]; // F1
17 }
```

```
1  //Binary Ninja
2  // F2, F16 vs F17
3  rax_2 = sub_40edd0(rdi_1, rsi_1, *(((data_6256b4 +
   ↪   ((data_6256ef + ((rax_5 + ((rdx_3 + r8_1) << 1)) <<
   ↪   1)) << 1)) << 3) + 0x419580));
4
5
6  //Radare
7  if (rax <= 0x41d810) {
8    goto label_0;  // F4
9  }
10 __asm ("comiss xmm1, dword [0x0041d82c]"); // F5
11 if (rax < 0x41d810) {
12   goto label_0;  // F4
13 }
14 label_0: // F4
15 *(rdi) = 0x41d810;
16 eax = 0;
17 rax = rcx + rax*2;
```

Fig. 2. Illustration of decompiled code excerpts from Hex-Rays and Angr (left), and Binary Ninja and Radare2 (right). The example codes help us to determine the common features of F1-2, F4-5, F7, F16-18, and F22.

unnecessary variables, and reducing nested casts. Second, we put three features into the *User Preference* category because they can rely on one's coding style or individual preferences. For instance, one prefers to choose between `for`, `while`, or `do-while` when dealing with a loop statement (F11 in Table 2), which cannot tell absolutely better readability. To accommodate the features pertaining to user preference, we further subdivide the F11 feature into F11-`switch`, F11-`if`, and F11-ternary, and assign different weights based on the survey results. In the same vein, we subdivide F12 into F12-`for`, F12-`while`, F12-`dowhile`, and F13 into F13-`!strcmp` and F13-`strcmp`. Third, we separately group the four features in conflict with each other because improving one feature may have a negative impact on others. For instance, reducing `nested-if` statements by combining multiple `if` statements could increase the number of conditions within those `if` statements. Such features are categorized under the *Conflicting features* category. Fourth, nine additional features are associated with syntax errors, which are grouped under the *Erroneous Syntax* class. Decompilers frequently generate code with invalid syntax due to their inability to decompile or when essential information, such as variable types and function names, is missing from the binary. Lastly, the remaining five features have been obtained from the previous work on readability metrics for source code, grouping them as the *General features* category. We conducted a thorough examination of these metrics and incorporated the features that are applicable to a decompiler's output.

Feature Examples. Figure 2 presents code excerpts from the Hex-Rays, Binary Ninja, Angr, and Radare2 decompilers, containing the selected features. Notably, for the same function call statement on line 3 (*e.g.*, `0x40EDD0`), Hex-Rays and Binary Ninja generates different codes. Hex-Rays tend to create more lines of code (F17) for readability whereas Binary Ninja does longer lines (F16) with relatively more operators (F2) on line 3 than Hex-Rays'. Radare2 commonly utilizes `goto` statements (F4) on lines 8, 12, and 14, along with frequent inline assembly (F5) on line 10. Angr's code exhibits unclear elements, such as declaring variables of unknown type (F18) on line 12, using intermediate representations like `Conv` and `BinaryOp` (F22) on line 14, and employing multiple nested casts (F7) on line 14.

Relative Feature Weights. We design our readability metric so that it can parameterize the relative feature weights that account for individual subjective factors. In this study, we initially assigned an equal weight of 0.0323, calculated as 1/31, to each feature, given that there are 31 features in total. For features within the *User-Preference* and *Conflicting* categories, we assigned different weights based on the results of our user survey. Regarding conditional statements, 90.9% of participants favored `switch` over `if`, as indicated in Figure 3 (F11). Moreover, 59.1% of participants preferred `if` over ternary. Consequently, we assigned a higher weight (0.0161) to `switch` (F11-`switch`), calculated

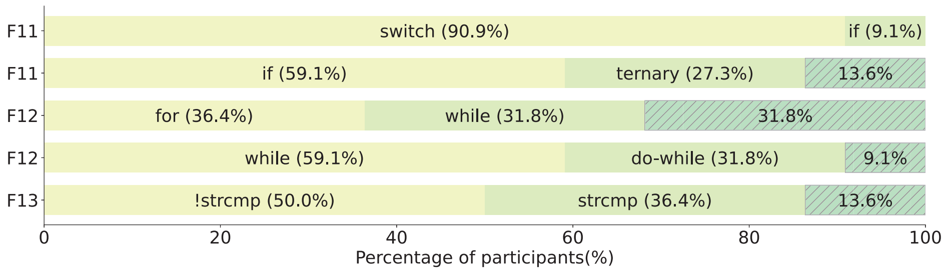


Fig. 3. Survey results of participants' preferences for the features (F11-13 in Table 3). The participants responded that code readability increases in the order of `switch`, `if`, and `tenary` for conditional statements; `for`, `while`, and `do-while` for loops; and `!strcmp` over `strcmp` for string comparisons. The hatched pattern represents no preference.

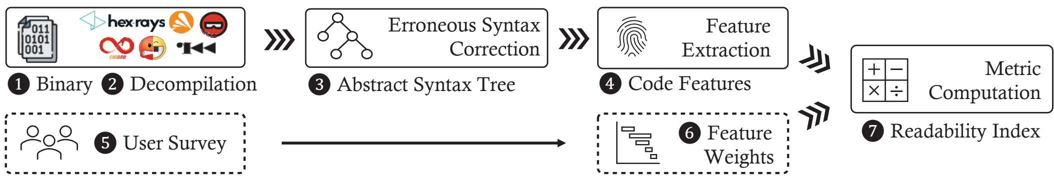


Fig. 4. Overview of our readability assessment system for decompiled code. With a given binary (❶), target decompilers produce decompiled code (❷). As a pre-processing, we correct syntactic errors to generate an abstract syntax tree (❸; §4.1) that assists in extracting code features. Meanwhile, we pre-define useful features and their weights (❻; §3.3) from a user survey (❺; §3.2), which is a one-time process (*i.e.*, dotted box). Finally, we compute a relative readability index (❼; §4.2) from the features and weights.

as $((1/31) * (3/6))$. For `if` (F11-`if`), we applied a weight of 0.0108 $((1/31) * (2/6))$, and for ternary (F11-ternary), a weight of 0.0054 $((1/31) * (1/6))$ was assigned. Similarly, based on participant preferences indicated in Figure 3 (F12), we assigned a weight of 0.0161 to `for` (F12-`for`), 0.0108 to `while` (F12-`while`), and 0.0054 to `do-while` (F12-`dowhile`), with most participants favoring `for`, followed by `while`, and then `do-while`. Lastly, our survey demonstrated that code using `!strcmp` is perceived as more readable than code using `strcmp` when comparing strings. Additionally, all participants agreed that having fewer conditions in `if` statements (F14) is preferable to `nested if` statements (F15). Furthermore, all participants favored shorter line length (F16) over longer lines of code (F17). We assigned different weights to these features based on these survey results, similar to our approach for F11-if, F11-switch, and F11-ternary. It is important to note that these weights can be customized flexibly according to the specific context.

4 RELATIVE READABILITY INDEX FOR DECOMPILED CODE

This section defines a relative readability index for decompiled code. For handy comparison, we develop a system that can assess decompiled code readability that takes a binary as input and computes the individual indexes for target decompilers.

Overview. Figure 4 depicts the overview of our code readability evaluation process tailored to decompiled code. First, we have a list of decompilers that produce (binary-function-based) decompiled code snippets (❷) from a given binary (❶). A naïve decompiler-generated code typically entails a distinct high-level code, which often contains an erroneous syntax or expression. This is

Table 3. Various syntax error examples in C-like decompiled code outputs. We fix those codes so that one can construct an AST for further computing a code readability index. We selectively define features ("Erroneous Syntax" category in Table 2) that hurt the readability, excluding trivial errors like a missing parenthesis (*e.g.*, dash(-) in the feature column). The description column indicates a regular express pattern to detect each example (unless otherwise stated).

<table>
<tr><th>Error Type</th><th>Feature</th><th>Example</th><th>Correction</th><th>Description</th></tr>
<tr><td>Declarations</td><td>F18</td><td>unsigned int | char v0;</td><td>undefined v0;</td><td>/^[\w\s*(\[\d+\])]+(\|[\w\s*(\[\d+\])]+)+/</td></tr>
<tr><td></td><td>F18</td><td>undefined v1;</td><td>typedef int undefined;</td><td>(Defining a new type in a custom header)</td></tr>
<tr><td></td><td>F18</td><td>(_UNKNOWN *)v19</td><td>typedef void _UNKNOWN;</td><td>(Defining a new type in a custom header)</td></tr>
<tr><td></td><td>F18</td><td>code **ppcVar1;</td><td>typedef int code;</td><td>(Defining a new type in a custom header)</td></tr>
<tr><td></td><td>-</td><td>signed int64_t var;</td><td>int64_t var;</td><td>/[\s\(]signed\sint[^\s\)]+/</td></tr>
<tr><td>Structures</td><td>F19</td><td>LAB_004c8dba: }</td><td>INVAL_LAB;</td><td>(Devising a detection routine)</td></tr>
<tr><td></td><td>F20</td><td>do{ .. } .. }while(..)</td><td>INVAL_DOWHILE;</td><td>(Devising a detection routine)</td></tr>
<tr><td>Identifiers</td><td>F21</td><td>void (*0x401350)()();</td><td>INVAL_FUNCALL();</td><td>/\)\([\s\S,()*+]*\)\s\([\s\S,()*+]*\);/</td></tr>
<tr><td></td><td>-</td><td>rdx:rax</td><td>rdx_rax</td><td>/(rdx:rax)|(edx:eax)/</td></tr>
<tr><td></td><td>-</td><td>rsp<8> = 5;</td><td>rsp = 5;</td><td>/(?<![a-z])[a-z]{2,3}<\d+>/</td></tr>
<tr><td></td><td>-</td><td>v6 = ::s1;</td><td>v6 = s1;</td><td>/\S*::\S*/</td></tr>
<tr><td>Expressions</td><td>F24</td><td>if(...)</td><td>if(unknown)</td><td>/if\s?\(\.\.\.\)/</td></tr>
<tr><td></td><td>F24</td><td>(? > ?) ? 1 : 0;</td><td>(unknown) ? 1 : 0;</td><td>/\(\s*\?\s*[!<>=]+\s*\?\)/</td></tr>
<tr><td></td><td>F24</td><td>? = fp_stack[0]</td><td>(unknown) = fp_stack[0]</td><td>/^\s*\?\s\=\s/</td></tr>
<tr><td></td><td>F25</td><td>setjmp({(struct{ })</td><td>setjmp(INVAL_FORM)</td><td>/jmp\(\s*\{/</td></tr>
<tr><td></td><td>-</td><td>&&var5[0]</td><td>&(&var5[0])</td><td>/&&\S/</td></tr>
<tr><td></td><td>-</td><td>__asm { }</td><td>__asm ( )</td><td>/__asm\s{.*}/</td></tr>
<tr><td></td><td>-</td><td>assert("!\"invalid\");</td><td>assert("!\"invalid\"");</td><td>/[^\\]\\"(,|\)*;$)/</td></tr>
<tr><td></td><td>-</td><td>int start(..) __noreturn</td><td>noreturn int start(..)</td><td>/_{0,2}noreturn/</td></tr>
<tr><td></td><td>-</td><td>v1 = 10f</td><td>v1 = 10.0f</td><td>/[\s\(]-?\d+f/</td></tr>
<tr><td>Operators</td><td>F26</td><td>if(ebp overflow 0)</td><td>if(UNKNOWN_OP)</td><td>/(if|while).*overflow/</td></tr>
<tr><td>Eccentricities</td><td>F22</td><td>Conv(16 ->128, di);</td><td>INVALID_IR;</td><td>/Conv\(|BinaryOp\s[A-Z]/</td></tr>
<tr><td></td><td>F23</td><td>x = /*x = unimplemented { }*/;</td><td>x=UNIMPL;</td><td>//*.**/(?=\)*;)/</td></tr>
</table>

mostly because a decompiler does not aim to produce re-compilable code but to focus on human-readable code generation (*i.e.*, de-prioritizing syntactic errors). Second, we correct syntactic errors so that one can generate an abstract syntax tree (❸; §4.1). Next, we extract the predefined features (❹) in Table 2 from each tree. Note that the feature weights (❻; §3.3) are pre-computed via a user survey (❺; §3.2). Finally, using the features and their relative weights, we compute a relative readability index (❼; §4.2) across target decompilers.

4.1 Feature Extraction with ASTs

Once a decompiled code snippet (*i.e.*, function-based code identified by a decompiler) is obtained, we build an AST from the code to extract the pre-defined features. As the decompilers in our experiments follow C-like syntax, we construct a C-based AST. However, the original decompiled code contains a variety of erroneous syntax, which cannot be directly built to an AST form.

Syntactic Error Types. We categorize (C-like; delimited by a semi-colon) statements that incorporate syntactic errors into six error types: ① declarations (*e.g.*, wrong data types), ② structures (*e.g.*, invalid structures), ③ identifiers (*e.g.*, incorrect identifiers), ④ expressions (*e.g.*, inappropriate expressions), ⑤ operators (*e.g.*, unknown operators), and ⑥ other compiler-specific eccentricities (*e.g.*, unexpected handling routines, missing implementation like `unimplemented`). Table 3 summarizes such error types, corresponding features (if any), and how to resolve erroneous syntax with a variety of instances.

Erroneous Syntax Correction. In most cases, we can address the above issues either ① by defining new types with `typedefs` in a custom header or ② by replacing erroneous cases with the forms that can be readily built into an AST using regular expressions. For example, we define a new data type in the custom header in case that a variable has been declared with a wrong type like `signed`, `undefined`, or `UNKNOWN`. Similarly, we utilize a regular expression to capture incorrect identifiers (*e.g.*,

`void (*0x401350)`, `rsp<8>`, `::s1`), inappropriate expressions (*e.g.*, `...` in a conditional statement, `__asm{}`, `? > ?`), and unknown operators (*e.g.*, `ebp overflow 0`). However, it requires ③ logical and deductive approaches to fix a malformed structure. For example, `do{ .. } .. } while(..)` in Table 3 does not hold the even number of parentheses (*e.g.*, opening/closing) in a statement. As a final note, we fix a trivial syntax error for building an AST rather than defining it as a feature (*e.g.*, `__asm{}` → `__asm()`).

4.2 Relative Readability Index for Decompiled Code

Metric Design. The readability metric for decompiled code is designed with the following functionalities in mind, which can ① provide a handy readability index that can be determined with a straightforward range (*i.e.*, $(0, 1)$), ② represent a readability difference in both relative and quantitative manners, and ③ process decompiler-generated code appropriately to compute the index.

Readability Index for Decompiled Code. We introduce the *relative readability index (R2I)* for decompilers' outputs, which is capable of representing quantitative differences between the decompiled code. By design, R2I can be computed merely when there are multiple decompilers (*i.e.*, $n >= 2$). Suppose that we have n different decompilers, m features that are obtained from AST, and relative feature weights ($\sum_{i=1}^{m} w_i = 1$) from our user study. Let $\mathcal{A}_{n\times m}$ be a matrix that holds every value of a jth feature for an ith decompiler. Then, we apply a transformation on the matrix ($T(\mathcal{A})$) with the following two processes: ① reducing the number of features (*i.e.*, $m \rightarrow m'$) by eliminating all feature values that are the same across decompilers (*i.e.*, $f_{max} - f_{min} = 0$), and ② taking the delta per feature (*i.e.*, $\Delta_j = f_j - f_{min}$), followed by computing an exponential decay function (*i.e.*, e^{-x}). We denote $r \in \{0, 1\}$ according to two distinct feature groups: ① the smaller value, the better readability ($r = 1$; *e.g.*, the number of tokens, line of code) whereas ② the larger, the better ($r = 0$; *e.g.*, the number of array). Note that this term assists in handling both feature groups so that the final index preserves the same direction. Now we can obtain $\mathcal{A}'_{n\times m'}$ throughout an element-wise computation as follows:

$$\mathcal{A}' = T(\mathcal{A}) : a'_{ij} = r \cdot e^{-\log_{10}(1+\Delta_{ij})} + (1-r) \cdot (1 - e^{-\log_{10}(1+\Delta_{ij})}) \tag{1}$$

Because the number of features has relatively decreased, we adjust the feature weights (W') accordingly.

$$W' : w'_i = \frac{w_i}{\sum_{i=1}^{m'} w_i} \text{ s.t.} \sum_{i=1}^{m'} w'_i = 1 \tag{2}$$

Finally, we can compute R2I with the following matrix multiplication, resulting in a relative readability metric for each decompiler.

$$R2I = \mathcal{A}' \cdot W'^T \tag{3}$$

The metric ranges from 0 to 1 where a value that is close to 1 indicates relatively good code readability, or 0 otherwise. The benefit of this approach is to *relatively reveal readability-favored features* with adjusted weights while ignoring identical features. It is worth noting that the R2I metric represents the relative differences based on a given list of decompiled codes (*i.e.*, rankings in the list), not the gaps on a different list (*i.e.*, rankings between the list) because R2I stays away from an absolute metric.

5 IMPLEMENTATION

This section describes the implementation of our R2I-based assessment system in detail.

Recognizing Target Functions. As described in §3.1, we compute R2I at the granularity of a function. We rely on each decompiler's capability of identifying a function boundary, followed by

Table 4. We curate six popular decompilers, most of which are under active development or improvement.

	Decompiler	IR	Version	Release
Commercial	Hex-Rays [Hex-Rays 2023b]	Microcode	8.2	Jan. 08, 2023
	Binary Ninja [Vector35 2023a]	BNIL	3.3	Jan. 26, 2023
Open-source	RetDec [Křoustek et al. 2017]	LLVM IR	5.0	Dec. 08, 2022
	Ghidra [NSA 2023b]	p-code	10.2.3	Feb. 09, 2023
	Angr [Shoshitaishvili et al. 2016]	VEX IR	9.2.40	Mar. 01, 2023
	Radare2 [Radare 2023; Wargio et al. 2023a]	ESIL	5.8.4	Mar. 15, 2023

including the functions in a code region (*i.e.*, `.text` section) while excluding system-linker-inserted ones (*e.g.*, `_start`, `deregister_tm_clones`, `register_tm_clones` etc.). Note that we associate a function with an address because of variations in function naming schemes across decompilers.

Building ASTs. We utilize `pycparser` [Eliben 2023], one of the popular open-source parsers for C code, to construct an AST. Given that decompiled code may not strictly adhere to C syntax, we modified the parser to deal with syntactic errors (§4.1).

Extracting Features. We wrote a script that can extract the pre-defined features on top of an AST by counting the occurrences of relevant nodes or sub-trees that hold the feature. Note that we directly obtain the features of a code length and a line length from the original decompiled code.

6 EVALUATION

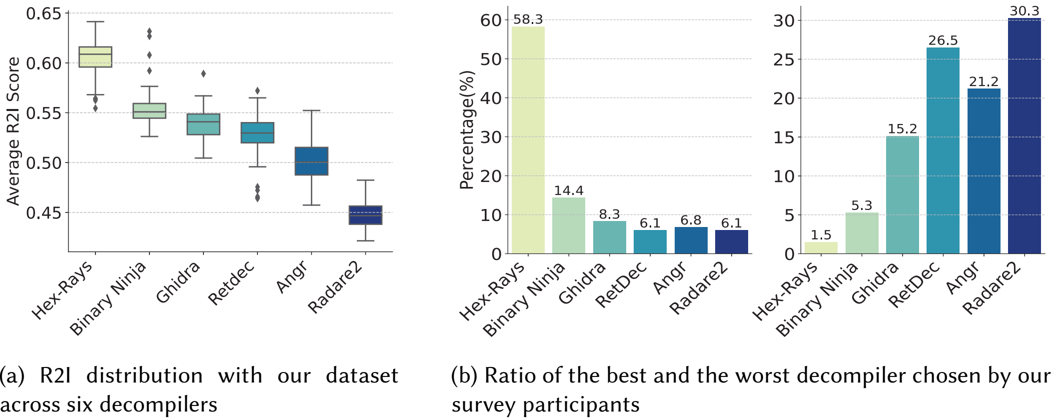
This section defines the following three research questions, demonstrating the usefulness of R2I.

- **RQ1: Practicality of R2I.** How well is R2I aligned with human perception in our survey?
- **RQ2: Effectiveness of R2I.** How effective is R2I in evaluating the readability of decompiled code (in terms of applicability)?
- **RQ3: Correlations of Features.** How are the features and R2I correlated with each other?

Dataset and Environmental Setup. We evaluate the R2I metric using GNU `Coreutils` 8.29 [GNU 2017] and `Findutils` 4.6.0 [GNU 2015], compiled with `GCC` 8.2.0 at the optimization level `O2`. We remove debug symbols from binaries to reflect real-world scenarios. Moreover, the decompilation-failing binaries (*e.g.*, size > 150 KB) by (at least) one of the six decompilers have been excluded [3], resulting in a corpus containing 103 `Coreutils` binaries and four `Findutils` binaries. With our corpus, decompilers identify 68,464 functions as a whole, out of which 5,305 functions (*i.e.*, 7.75%) are detected by all decompilers in our experiments. To compute relative metrics, we utilize these 5,305 functions to assess the effectiveness of R2I. Note that we utilize the LLVM obfuscator [Junod et al. 2017] to generate obfuscated binaries. The evaluation was conducted on Ubuntu 20.04, equipped with an Intel(R) Core(TM) i9-11900 @ 2.50GHz processor with 16 cores and 96 GB RAM.

Selective Decompilers. For this study, we choose six decompilers from the (non-comprehensive) list [Vector35 2023b] that ① support the decompilation of 64-bit executables in ELF (Executable and Linking Format) [Committee 1995], and ② are under active development (*i.e.*, updates within the last six months as of writing). Table 4 summarizes the popular decompilers, including two commercial-off-the-shelf (COTS) products (*i.e.*, Hex-Rays [Hex-Rays 2023b] version 8.2, Binary Ninja [Vector35 2023a] version 3.3), and four open-source software (*i.e.*, Ghidra [NSA 2023b] version 10.2.3, Radare2 [Radare 2023; Wargio et al. 2023a] version 5.8.4, RetDec [Křoustek et al. 2017] version 5.0, and angr [Shoshitaishvili et al. 2016] version 9.2.40). Note that most of them are designed as

[3] `du`, `tr` of `Coreutils`, and `find`, `locate` of `Findutils` have been ruled out.



(a) R2I distribution with our dataset across six decompilers

(b) Ratio of the best and the worst decompiler chosen by our survey participants

Fig. 5. (a) The boxplot shows that decompiled code by Hex-Rays has better readability (*i.e.*, median R2I = 0.608) than others. (b) We asked the participants to choose the best (Hex-Rays; left) and the worst (Radare2; right) decompilers for code readability, which is well-aligned with the R2I ranking of the decompilers in (a).

a binary analysis framework, thus encompassing other features like disassembly and symbolic execution as well as decompilation.

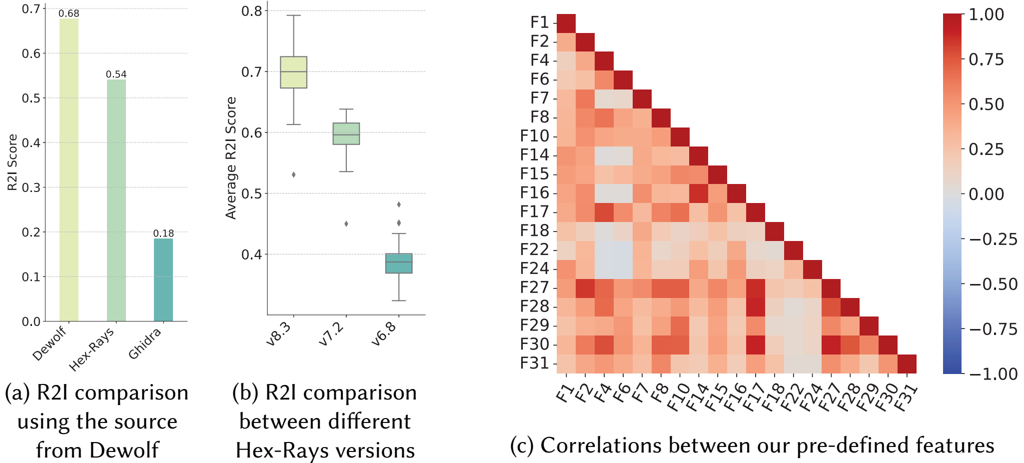
6.1 RQ1: Practicality of R2I.

To examine the feasibility of R2I as a code readability assessment metric, we compute R2I to evaluate the readability of decompiled code produced by six decompilers. We then compare these results with those obtained from the second user survey, seeking to determine the alignment between the R2I metric and user perceptions of decompiled code.

R2I Results. We compute R2I for individual functions within the binaries and subsequently average them per binary, ranking the decompilers. Figure 5a depicts the average R2I scores for 107 binaries from `Coreutils` and `Findutils`. Hex-Rays attains the highest R2I of 0.604 on average, followed by Binary Ninja at 0.553, and Ghidra at 0.539. RetDec and Angr achieve the indexes of 0.528 and 0.501, respectively, while Radare2 has the lowest index of 0.448.

Our Survey Results. In our main survey, the participants provided feedback on the most and least comprehensible decompiled code among the six decompilers. Figure 5b illustrates the results, with 58.3% of participants selecting Hex-Rays' decompiled code as the most readable. Binary Ninja ranks second with 14.4%, closely followed by Ghidra at 8.3%. The remaining decompilers, Angr, RetDec, and Radare2, received 6.8%, 6.1%, and 6.1% of participant preferences, respectively. Conversely, as depicted in Figure 5b, 30.3% of participants identified Radare2's decompiled code as the least comprehensible, while very few participants (1.5%) indicated that Hex-Rays' decompiled code was the least comprehensive.

Dewolf Survey Results. While the alignment of R2I with the preferences of the participants of our choice is noteworthy, the results may vary with different participant groups. Recent work [Enders et al. 2023] introduces another decompiler (Dewolf) aimed at enhancing code readability. The authors evaluated the readability of Dewolf's decompiled code compared to Hex-Rays and Ghidra through a user survey involving 53 participants. Thus, we applied R2I to the same decompiled code from the survey, and compared the results with the prior survey findings. Figure 6a presents the R2I scores with the Dewolf decompiler achieving the highest score (0.68), followed by Hex-Rays (0.54), and Ghidra (0.18). These results align closely with the survey findings, where 83% of participants



(a) R2I comparison using the source from Dewolf

(b) R2I comparison between different Hex-Rays versions

(c) Correlations between our pre-defined features

Fig. 6. To show the practicality of R2I, (a) we apply R2I to the source provided by Dewolf [Enders 2021] that proposes a readability-enhancing decompiler. The R2I metric of Dewolf shows a better index than that of Hex-Rays [Hex-Rays 2023b], reflecting the improving efforts on decompiled code. Similarly, (b) we compute R2I to see the enhancements across different Hex-Rays versions, demonstrating that the higher version produces better decompiled code. Meanwhile, (c) we investigate the correlations between the features with a heatmap, revealing positive relationships of several features (*e.g.*, the line of code (F17) and the number of conditions (F28)). Any features producing the correlations with p-values under 0.01 have been eliminated.

favored Dewolf's decompiled code as the most comprehensible, while 13% selected Hex-Rays and 4% chose Ghidra. This demonstrates that R2I aligns with other surveys as well.

Summary 1: R2I's practicality is evident, as its best and worst scores align closely with users' preferences. The R2I scores closely correspond to the second to fifth users' preferences. The slight variance in rankings between Binary Ninja and Ghidra is negligible, given the R2I metric's indication of nearly identical levels of readability for both decompilers.

6.2 RQ2: Effectiveness of R2I

We now assess the effectiveness of R2I metric through two different applications; ① different versions of the same decompiler, and ② decompiled code from obfuscated binaries.

6.2.1 R2I with Evolving Decompilers. Decompilers, including Hex-Rays and Binary Ninja, keep advancing code readability through updates. To assess the effectiveness of R2I metrics in capturing these improvements, we select three versions of Hex-Rays: v6.8 (initial x64 decompiled code with limited readability improvements), v7.2 (introduction of microcode intermediate language), and v8.3 (the latest version at the time of writing).

Results. Figure 6b illustrate the average R2I across all binaries within our dataset for three different decompiler versions. Note that the R2I consistently increases as the decompiler version advances. Hex-Rays v8.3, the latest version, achieves the highest readability with a median R2I of 0.698, followed by Hex-Rays v7.2 at 0.595, and Hex-Rays v6.8 at 0.386. Moreover, the score gap between

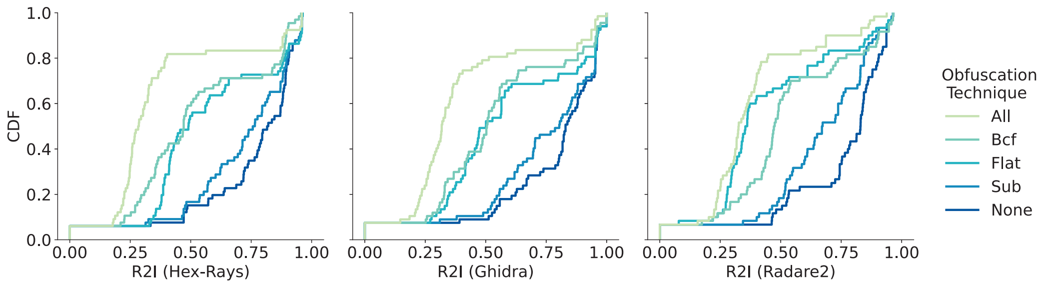


Fig. 7. R2I CDF comparisons from the (non-)obfuscated binaries (*e.g.*, `whoami` from `coreutils`). R2I shows a clear distinction between a non-obfuscated binary (*i.e.*, None) and obfuscation-technique-applied binaries (*e.g.*, Bcf: bogus control flow, Flat: control flow flattening, Sub: instruction substitution). Note that the decompiled code readability with all obfuscation techniques ranks the lowest with R2I.

Hex-Rays v6.8 and v7.2 is relatively more significant than v7.2 and v8.3, indicating more substantial readability improvements from v6.8 to v7.2.

> **Summary 2**: R2I effectively reflects improvements in decompiler versions, evidenced by higher scores in the latest versions and highlighting the differences between versions.

6.2.2 R2I with Obfuscated Binaries. An obfuscation technique [Junod et al. 2017] typically degrades code readability. To demonstrate the applicability of R2I, we apply various obfuscated techniques to binaries to see how R2I reflects the degree of obfuscation. We examine the decompiled code from each binary with no obfuscation (none) and those with different obfuscation techniques, including bogus control flow (bcf), control flow flattening (flat), and instruction substitutions (sub). Our evaluation comprises Hex-Rays and Radare2, known for producing the most and least readable decompiled code according to the R2I metric, respectively. Additionally, we include Ghidra, recognized for generating the most readable code among open-source decompilers. Unlike prior evaluations, we use non-stripped binaries because obfuscations often rearrange function locations, making it infeasible to identify identical functions across different binaries. It is noteworthy to mention that R2I represents relative gaps between obfuscation techniques, rather than between decompilers.

Results. Figure 7 shows that a binary obfuscation tends to lower R2I, highlighting the R2I metric's ability to reflect different obfuscation techniques. We confirm that R2I significantly drops when all obfuscation techniques are applied compared to their non-obfuscated counterparts. Decompiled code with bcf and flat obfuscations results in a similar R2I. Interestingly, we sometimes see minor differences or no difference in R2I between non-obfuscated code and the one with a sub-obfuscation, hypothesizing that the impact of instruction substitutions limits the readability of decompiled code. Notably, we observe a few instances where binaries with all obfuscations have higher R2I than those not. Our in-depth investigation shows that a combined obfuscation technique sometimes complicates the whole function but a few simple ones (*e.g.*, control flow flattening), bringing about generating a simplified function while rendering others exceptionally challenging to understand.

> **Summary 3**: The R2I metric adequately reflects obfuscation techniques applied to a binary, maintaining a high index for non-obfuscated code while a low index for obfuscated code.

Table 5. Selective (weighted) features that affect R2I for the function at `0x401b20` from the `hostid` binary in our corpus. Figure 8a contains the source and the decompiled outputs. The RetDec's `switch-case` gains a relatively high score (*e.g.*, 0.0097) based on one of the user-preference features (*e.g.*, F11-switch). However, other features like inline assembly (F5) and the number of conditions (F28), drop the overall R2I. Meanwhile, line of code (F17), the lower number of conditions (F28) and assignments (F30) in Hex-Rays than others benefit overall readability, leading the highest R2I of 0.7055.

Decompiler	F4	F5	F10	F11-switch	F11-if	F15	F17	F18	F21	F28	F30	R2I
Hex-Rays	0.0582	0.0582	0.0582	0.0000	0.0129	0.0129	**0.0388**	0.0582	0.0582	**0.0291**	**0.0582**	**0.7055**
Binary Ninja	0.0582	0.0582	0.0029	0.0000	0.0129	0.0129	0.0017	0.0582	0.0582	0.0194	0.0044	0.5368
Ghidra	0.0582	0.0582	0.0041	0.0000	0.0129	0.0129	0.0013	0.0019	0.0582	0.0194	0.0044	0.5192
RetDec	0.0582	**0.0064**	0.0116	**0.0097**	0.0194	0.0388	0.0019	0.0582	0.0582	**0.0145**	0.0116	0.5099
Angr	0.0582	0.0582	0.0027	0.0000	0.0129	0.0194	0.0010	0.0582	0.0582	0.0194	0.0041	0.4370
Radare2	0.0145	0.0582	0.0027	0.0000	0.0129	0.0388	0.0008	0.0582	0.0291	0.0582	0.0016	**0.4332**

6.3 RQ3: Correlations for Features and R2I

This section explores the correlations between features and between R2I and the features.

R2I and Features. To identify the features that significantly influence the R2I metric, we conduct a Pearson correlation test with the initial hypothesis of no correlation between each feature and R2I (Figure 5a). The test results unveil that the following features exhibit the most significant negative correlation with R2I, consequently hurting code readability (*i.e.*, rejecting the hypothesis with a low p-value of $p < 0.01$ or 99% confidence level):

- Number of tokens (F27; $r = -0.6993, p < 10^{-100}$)
- Line of code (F17; $r = -0.5171, p < 10^{-100}$)
- Number of assignments (F30; $r = -0.5430, p < 10^{-100}$)
- Number of operators (F2; $r = -0.5103, p < 10^{-100}$)
- Number of references/dereferences (F8; $r = -0.4453, p < 10^{-100}$)

Correlations between Features. Subsequently, we conduct additional Pearson correlation tests to assess the relationships between R2I features. The outcomes of this test indicate that while most features lack definitive relevance, a subset of features exhibits distinct positive correlations, as depicted in Figure 6c. Notably, we observe a statistically significant positive correlation between the features related to the number of lines of code (F17) and the following features:

- Number of tokens (F27; $r = 0.8736, p = 2.89^{-19}$)
- Number of assignments (F30; $r = 0.8917, p = 3.80^{-26}$)
- Number of conditions (F28; $r = 0.8839, p = 3.49^{-24}$)

This implies that an increase in the number of tokens, conditions, and assignments is associated with longer lines of code. It is worth noting that the remaining features do not exhibit significant positive or negative correlations with each other (*i.e.*, independent features).

Summary 4: We identify a strong negative correlation between R2I and certain features, including code size (F17, F27) and code complexity (F2, F8, F30), which impair code readability. Additionally, we observe the (statistically meaningful) mutual feature correlations only between the size of the code and three features F17, F28, and F30.

7 CASE STUDY

In this section, to gain deeper insights into our metrics, we delve into R2I with the source code in Figure 8a and the outputs of three decompilers. In particular, we look into how R2I reflects

```
1  void parse_long_options (int argc, /* omitted */,void
↪   (*usage_func) (int), ...)
2  {
»
8    if (argc == 2 && (c = getopt_long (argc, argv, "+",
↪     long_options, NULL)) != -1)
9    {
10     switch (c)
11     {
12       case 'h':
13         (*usage_func) (EXIT_SUCCESS);
14         break;
15       case 'v':
16       {
17         va_list authors;
18         va_start (authors, usage_func);
19         version_etc_va (stdout, command_name, package,
↪           version, authors);
20         exit (0);
21       }
22       default:
23         break;
24     }
25   }
»
29 }
```

(a) Source code

```
1  void sub_401B20(__int64 a1, /* omitted */, void
↪   (__fastcall *a6)(_QWORD), ...)
2  {
3    int v6; // ebx
4    int v11; // eax
5    gcc_va_list va;
»
9    if ( (_DWORD)a1 == 2 )
10   {
11     v11 = sub_404DF0(a1, a2, "+", &off_405A40, 0LL);
12     if ( v11 != -1 )
13     {
14       if ( v11 == 104 )
15       {
16         a6(0LL);
17       }
18       else if ( v11 == 118 )
19       {
20         va_start(va, a6);
21         sub_403C70(stdout, a3, a4, a5, va);
22         exit(0);
23       }
24     }
25   }
»
28 }
```

(b) Hex-Rays

```
1  int64_t function_401b20(int64_t a1, int64_t a2, char *
↪   a3, char * a4, int64_t a5, int64_t a6) {
»
4    if ((char)v1 != 0) {
»
12     __asm_movaps(v2);
13     __asm_movaps(v2);
14     __asm_movaps(v2);
15   }
»
18   if ((int32_t)a1 != 2) {
19     //0x401b7f
20     g27 = v3;
21     g28 = 0;
22     int64_t result; // 0x401b20
23     return result;
24   }
25   int64_t result2 = function_404df0(a1, a2, &g3,
↪     (int64_t *)&g4, 0, a6); // 0x401bb9
26   int32_t v4 = result2; // 0x401bbe
27   switch (v4) {
28     default: {
29       // 0x401bc8
30       if (v4 == 118) {
31         int64_t v5 = 48; // bp-240, 0x401bea
32         function_403c70((int64_t)g30, (int64_t)a3,
↪           (int64_t)a4, a5, &v5, a6);
33         exit(0);
34         // UNREACHABLE
35       }
36     }
37     case -1: {
38     }
39     case 104: {
40       //0x401b7f
41       g27 = v3;
42       g28 = 0;
43       return result2;
44     }
45   }
46 }
```

(c) RetDec

```
1  uint64_t fcn_00401b20 (int64_t arg_100h, /* omitted */,
↪   int64_t arg9) {
2    int64_t var_8h;
»
14   int64_t var_c0h;
15   rdi = arg1;
»
25   if (al != 0) {
26     *((rsp + 0x50)) = xmm0;
»
33     *((rsp + 0xc0)) = xmm7;
34   }
»
37   while (eax == 0xffffffff) {
38 label_0:
»
45     ecx = 0x405a40;
46     edx = 0x405a18;
47     r14 = r9;
48     eax = fcn_00404df0 ();
49   }
50   if (eax == 0x68) {
51     goto label_1;
52   }
53   if (eax != 0x76) {
54     goto label_0;
55   }
56   rcx = r13;
57   rdx = r12;
58   rsi = rbp;
59   r8 = rsp + 8;
60   rax = rsp + 0x100;
»    *((rsp + 8)) = 0x30;
66
67   fcn_00403c70 (*(obj.stdout));
68   exit (0);
69 label_1:
70   edi = 0;
71   void (*r14)() ();
72   goto label_0;
73 }
```

(d) Radare2

Fig. 8. (a) Source code of the function `parse_long_options` and three decompiled code snippets generated by (b) Hex-Rays, (c) RetDec, and (d) Radare2. The representations of conditional statements (*e.g.*, `if`, `switch-case` in grey) vary depending on each decompiler. The R2I metrics are (b) 0.7055, (c) 0.5099, and (d) 0.4332, respectively. Note that part of the code has been omitted for brevity.

code readability with actual decompiled codes. Table 5 is in the process of calculating our metric, where each entry represents the weighted value of a feature for the `parse_long_options` function from the `hostid` binary in our corpus. Hex-Rays has the highest R2I of 0.7055, while Radare2 has the lowest R2I of 0.4332. Binary Ninja, Ghidra, and RetDec have R2I of 0.5368, 0.5192, and 0.5099, respectively.

Comparison between Decompiled Outputs. The original function (Figure 8a) takes a variable number of parameters, and in its body, there is a single `switch-case` statement within an `if` condition. In Figure 8b, Hex-Rays generates nested `if` statements (F15) rather than the `switch` statement (F11-`switch`). The overall concise code with relatively fewer local variables (F10) and assignments (F30) than other decompilers looks the most similar output to the original. Meanwhile, RetDec (Figure 8c) produces a switch statement to express multiple conditional statements, similar to the source. However, it becomes less readable than Hex-Rays because of the inline assembly (F5) (*i.e.*, representing the variable number of arguments) and lines of code (F17). Radare2 (Figure 8d) generates the least readable code, which significantly differs from the original code. It embeds `goto` statements (F4) that complicate the program's control flow. Moreover, Radare2 generates incomprehensible function calls (F21) such as `(*r14)()()` and numerous assignments that reuse register-based-naming variables (*e.g.*, `rax` and `eax`), significantly impairing code readability. Note that R2I incorporates the above properties of each decompiler as features well, reflecting them to a readability index.

8 THREATS TO VALIDITY

This section discusses the limitations of our work and future directions.

Decompilation Capability and Accuracy. Our code readability system predominately relies on the capability of multiple decompilers' decompilation, constraining that R2I could be computed solely with produced outputs. For example, if there were a single decompiler to successfully generate a (decompiled) code snippet while all others failed, it would be insufficient to calculate R2I. Additionally, our evaluation assumes that the original source (*i.e.*, ground truth) is unavailable, hence the accuracy (*i.e.*, absolute quality) of decompiled code is orthogonal to our approach.

Function Boundary Detection. Although discovering a function boundary by target decompilers is another research problem [Alves-Foss and Song 2019; Andriesse et al. 2017; Bao et al. 2014; Koo et al. 2023; Shin et al. 2015], it may limit the computation of R2I. A different capability of recognizing a function may give rise to failing to compute R2I: *e.g.*, our experiments show that less than 10% functions have been identified for all six decompilers in our corpus.

Semi-automation for Syntax Correction. Recall that the features are extracted atop an AST while (C-like) decompiled code sometimes may not obey a programming language grammar. We empirically repair such an erroneous syntax problem by generating a regular-expression-based pattern and inserting a fake header, however, such semi-automated fixes still hardly cover every decompiler-producing output. To address this issue, we remain relaxing AST generation by patching `pycparser` [Eliben 2023] as part of our future work.

Semantic Features. The R2I metric focuses solely on syntactic features from ASTs and does not incorporate semantic code improvements, like meaningful constants. For instance, the recent efforts to enhance readability from Hex-Rays include a fruitful constant; *e.g.*, `operation == WriteKey` instead of `operation == 2` where "WriteKey" conveys better contextual meaning. To account for such improvements, we plan to explore semantic code analysis in our future work.

Representativeness of Survey Participants. The user survey results with 22 participants may reflect their preferences in a *limited* context or could differ with a different group of participants. Hence, the R2I system has been implemented with adjustable feature weights. As part of our future work, we plan to conduct a large-scale user survey to assess R2I within the decompiler community.

Decompiler-specific Common ASTs. It is possible for a decompiler to facilitate the creation of a decompiler-specific AST, distinct from a C-based AST. For instance, Hex-Rays offers a plugin [Bachaalany 2007] that can represent a decompiled function as a proprietary AST. One potential

avenue for future development could involve establishing a standardized, common AST format that is interoperable with various decompilers.

9 RELATED WORK

This section outlines three related areas to our work: source code readability models, readability-affecting factors, and decompiler-enhancing efforts.

Source Code Readability Models. Buse et al. [Buse and Weimer 2008] propose a readability model (*i.e.*, logistic regression) based on a set of syntactical features, which are obtained from subjective ratings of the readability of code snippets by human annotators. Posnett et al. [Posnett et al. 2011] improve (*i.e.*, simplify) the model with Buse et al.'s mean scores and the Halstead [Halstead 1977] metrics. Similarly, Dorn [Dorn 2012] introduces a general software readability model by additionally including features that capture structural patterns, visual perception, natural language notions, and alignments. On the other hand, Scalabrino et al. [Scalabrino et al. 2018, 2016] emphasize the textual aspects of source code in identifiers and comments for measuring code readability. Recent advances utilize deep neural networks to enhance code readability: *e.g.*, Convolutional Neural Networks (ConvNets) [Mi et al. 2018]. Later, Mi et al. [Mi et al. 2022, 2021] leverage a hybrid neural network to extract features from visual, semantic, and structural aspects of source code. Meanwhile, Fakhoury et al. [Fakhoury et al. 2019] claim the limitations of popular readability metrics like Scalabrino et al. [Scalabrino et al. 2018] or Dorn [Dorn 2012] because it does not necessarily align with actual code readability improvements. Note that our work focuses on the readability index for (machine-produced) decompiled code.

Readability-affecting Factors. Tashtoush et al. [Tashtoush et al. 2013] investigate various programming features that affect code readability, and evaluate it through a survey. In a similar vein, Lee et al. [Lee et al. 2013] study the impact of programming style on code readability. Mannan et al. [Mannan et al. 2018] conduct a research to understand the impact of code readability on the overall quality of software. Meanwhile, Johnson et al. [Johnson et al. 2019] explore the importance of code readability (*e.g.*, nested loop) in software development. Alawad et al. [Alawad et al. 2019] empirically reveal a negative correlation between readability and complexity. Oliveira et al. [Oliveira et al. 2020] investigate various factors that impact code readability like programming constructs and naming conventions. Beyer et al. [Beyer and Fararooy 2010] examine program dependency (deep degree) to assess software quality. Pecorelli et al. [Pecorelli et al. 2019] investigate the influence of code smells on improving code readability. Zhang et al. [Zhang et al. 2013] explore the impact of six contextual factors on software metrics.

Decompiler-enhancing Efforts. To improve decompiled code readability, C-decompiler [Chen et al. 2010] presents a practical decompiler that reduces redundant variables with data flow analysis and register propagation. Similarly, Dream [Yakdan et al. 2015] attempts to reduce `goto` statements that impair readability. The Byte-Equivalent Decompiler (BED) [Schulte et al. 2018] proposes a new decompilation technique that persistently recombines and recompiles source excerpts from a huge code database. Lately, Dewolf [Enders et al. 2023] introduces another decompiler that focuses on the enhancement of the readability and comprehensibility of decompiled code. We thoroughly study the previous readability-affecting factors and existing decompiler-improving efforts for code readability, defining the features for R2I.

10 CONCLUSION

Decompilation serves as a basis for reverse engineers to aid in understanding the underlying code semantics in a binary. Because of a myriad of decompilation applications such as security vulnerability analysis, malware behavior identification, and infringement detection of software

copyright, COTS decompilers like Hex-Rays have constantly been evolving for better-decompiled-code readability. Although previous studies propose several metrics for evaluating the readability of source code, little work has introduced that of decompiler-producing code. To the best of our knowledge, this work first suggests the Relative Readability Index, dubbed R2I, a specialized metric tailored to evaluate decompiled code in a relative context quantitatively. Our empirical evaluations, conducted using six widely used decompilers and user surveys, demonstrate the versatility of the R2I metric, not only representing the relative quality of decompiled code but also aligning with human perception in our user survey.

11 DATA-AVAILABILITY STATEMENT

To foster further readability research for decompiled code, we disclose all of our evaluation datasets to the public. We have opened the datasets on a preserved digital repository [Haeun et al. 2024a] and source code [Haeun et al. 2024b] to reproduce our work.

ACKNOWLEDGMENTS

We thank the anonymous reviewers for their constructive feedback. This work was partly supported by the two Institute of Information & Communications Technology Planning & Evaluation (IITP) grants funded by the Korean government (MSIT; Ministry of Science and ICT) (No. 2022-0-00688; AI Platform to Fully Adapt and Reflect Privacy-Policy Changes, No. 2022-0-01199; Graduate School of Convergence Security, Sungkyunkwan University), the Basic Science Research Program through the National Research Foundation of Korea (NRF) grant funded by the Ministry of Education of the Government of South Korea (No. NRF-2022R1F1A1074373), and the commissioned research project supported by the affiliated institute of Electronics and Telecommunications Research Institute (ETRI) (No. 2022-017). Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the sponsor.

REFERENCES

Duaa Alawad, Manisha Panta, Minhaz Zibran, and Md Rakibul Islam. 2019. An Empirical Study of the Relationships between Code Readability and Software Complexity. arXiv.

Jim Alves-Foss and Jia Song. 2019. Function Boundary Detection in Stripped Binaries. In *Proceedings of the 35th Annual Computer Security Applications Conference (ACSAC)*. San Juan, Puerto Rico.

Dennis Andriesse, Asia Slowinska, and Herbert Bos. 2017. Compiler-Agnostic Function Detection in Binaries. In *Proceedings of the 2nd*. Paris, France.

Avast. 2023. Retdec : Changelog. https://github.com/avast/retdec/blob/master/CHANGELOG.md

Elias Bachaalany. 2007. Hex-Rays Decompiler primer. https://hex-rays.com/blog/hex-rays-decompiler-primer/

Tiffany Bao, Jonathan Burket, Maverick Woo, Rafael Turner, and David Brumley. 2014. ByteWeight: Learning to Recognize Functions in Binary Code. In *Proceedings of the 23rd USENIX Security Symposium (Security)*. San Diego, CA.

Al Bessey, Ken Block, Ben Chelf, Andy Chou, Bryan Fulton, Seth Hallem, Charles Henri-Gros, Asya Kamsky, Scott McPeak, and Dawson Engler. 2010. A Few Billion Lines of Code Later: Using Static Analysis to Find Bugs in The Real World. *Commun. ACM* 53, 2 (2010), 66–75.

Dirk Beyer and Ashgan Fararooy. 2010. A Simple and Effective Measure for Complex Low-level Dependencies. In *Proceedings of the IEEE 18th International Conference on Program Comprehension (ICPC)*. IEEE, 80–83.

Danilo Bruschi, Lorenzo Martignoni, and Mattia Monga. 2006. Detecting Self-mutating Malware Using Control-flow Graph Matching. In *Detection of Intrusions and Malware & Vulnerability Assessment: Third International Conference, DIMVA 2006, Berlin, Germany, July 13-14, 2006. Proceedings 3*. Springer, 129–143.

Raymond PL Buse and Westley R Weimer. 2008. A Metric for Software Readability. In *Proceedings of the 2008 International Symposium on Software Testing and Analysis (ISSTA)*. ACM, 121–130.

Silvio Cesare, Yang Xiang, and Wanlei Zhou. 2013. Control Flow-based Malware Variantdetection. *IEEE Transactions on Dependable and Secure Computing (TDSC)* 11, 4 (2013), 307–317.

Mahinthan Chandramohan, Yinxing Xue, Zhengzi Xu, Yang Liu, Chia Yuan Cho, and Hee Beng Kuan Tan. 2016. Bingo: Cross-architecture Cross-os Binary Search. In *Proceedings of the 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering (FSE)*. 678–689.

Gengbiao Chen, Zhengwei Qi, Shiqiu Huang, Kangqi Ni, Yudi Zheng, Walter Binder, and Haibing Guan. 2010. A Refined Decompiler to Generate C Code with High Readability. *Special Issue: Focus Section on Selected Papers from the 2010 Conference on Cloud Computing and Virtualization (CCV)* 43, 11 (2010), 150–154.

TIS Committee. 1995. Executable and Linking Format (ELF) Specification. https://refspecs.linuxfoundation.org/elf/elf.pdf

Yaniv David, Nimrod Partush, and Eran Yahav. 2016. Statistical Similarity of Binaries. *Acm Sigplan Notices* 51, 6 (2016), 266–280.

Jonathan Dorn. 2012. *A General Software Readability Model.* Master's thesis. Department of Computer Science, University of Virginia.

Emmanuel Dupuy. 2023. JD-GUI : Java Decompiler. https://java-decompiler.github.io/

Eliben. 2023. Pycparser : Github. https://github.com/eliben/pycparser

Van Emmerik and Michael James. 2007. *Static Single Assignment for Decompilation.* Ph. D. Dissertation. University of Queensland.

Steffen Enders. 2020-2021. Dewolf Survey : Github. https://github.com/steffenenders/dewolf-surveys

Steffen Enders, Eva-Maria C Behner, Niklas Bergmann, Mariia Rybalka, Elmar Padilla, Er Xue Hui, Henry Low, and Nicholas Sim. 2023. Dewolf: Improving Decompilation by Leveraging User Surveys. In *Proceedings of The Workshop on Binary Analysis Research (BAR).*

Sebastian Eschweiler, Khaled Yakdan, Elmar Gerhards-Padilla, et al. 2016. discovRE: Efficient Cross-Architecture Identification of Bugs in Binary Code. In *Proceeding of the 23rd Annual Network and Distributed System Security Symposium (NDSS)*, Vol. 52. 58–79.

Sarah Fakhoury, Devjeet Roy, Adnan Hassan, and Vernera Arnaoudova. 2019. Improving Source Code Readability: Theory and Practice. In *Proceeding of the 27th IEEE/ACM International Conference on Program Comprehension (ICPC).* IEEE, 2–12.

Jian Gao, Xin Yang, Ying Fu, Yu Jiang, and Jiaguang Sun. 2018. VulSeeker: A Semantic Learning Based Vulnerability Seeker for Cross-platform Binary. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE)*. 896–899.

Free Software Foundation GNU. 2015. GNU Core Utilities : Findutils. https://ftp.gnu.org/gnu/findutils/

Free Software Foundation GNU. 2017. GNU Core Utilities : Coreutils. https://ftp.gnu.org/gnu/coreutils/

Eom Haeun, Kim Dohee, Lim Sori, Koo Hyungjoon, and Hwang Sungjae. 2024a. R2I: A Relative Readability Metric for Decompiled Code. https://doi.org/10.5281/zenodo.10684856

Eom Haeun, Kim Dohee, Lim Sori, Koo Hyungjoon, and Hwang Sungjae. 2024b. R2I: A Relative Readability Metric for Decompiled Code. https://github.com/e0mh4/R2I.git

Maurice H Halstead. 1977. *Elements of Software Science (Operating and Programming Systems Series).* Elsevier Science Ltd.

Hex-Rays. 2023a. Hex-Rays Decompiler : Comparisons of Decompilation Across Decompiler Version. https://hex-rays.com/decompiler/

Hex-Rays. 2023b. IDA Pro : A Powerful Disassembler and A Versatile Debugger. https://hex-rays.com/ida-pro/

John Johnson, Sergio Lubo, Nishitha Yedla, Jairo Aponte, and Bonita Sharif. 2019. An Empirical Study Assessing Source Code Readability in Comprehension. In *Proceedings of the 35th IEEE International Conference on Software Maintenance and Evolution (ICSME).* IEEE, 513–523.

Pascal Junod, Julien Rinaldini, Johan Wehrli, and Julie Michielin. 2017. LLVM Obfuscator. https://github.com/obfuscator-llvm/obfuscator/tree/llvm-4.0

Hyungjoon Koo, Soyeon Park, and Taesoo Kim. 2023. A Look Back on a Function Identification Problem. In *Proceedings of the 37th Annual Computer Security Applications Conference (ACSAC).* Austin, TX.

Jakub Křoustek, Peter Matula, and P Zemek. 2017. Retdec: An Open-Source Machine-Code Decompiler. In *Proceeding of the 6th Botnet and Malware Ecosystems Fighting Conference (BotConf).*

Jakub Křoustek, Peter Matula, and P Zemek. 2023. Retdec : Github. https://github.com/avast/retdec

Taek Lee, Jung Been Lee, and Hoh Peter In. 2013. A Study of Different Coding Styles Affecting Code Readability. *International Journal of Software Engineering and Its Applications* 7, 5 (2013), 413–422.

Umme Ayda Mannan, Iftekhar Ahmed, and Anita Sarma. 2018. Towards Understanding Code Readability and Its Impact on Design Quality. In *Proceedings of the 4th ACM SIGSOFT International Workshop on NLP for Software Engineering (NL4SE).* ACM, 18–21.

Nicholas D. Matsakis and Felix S. Klock. 2014. The Rust Language. *ACM SIGAda Ada Letters* 34, 3 (2014), 103–104.

Yukihiro Matsumoto. 2022. Ruby Programming Language. https://www.ruby-lang.org/.

Larry Melling and Bob Zeidman. 2012. Comparing Android Applications to Find Copying. *Journal of Digital Forensics, Security and Law* 7, 1 (2012), 4.

Jeff Meyerson. 2014. The Go Programming Language. *IEEE Software* 31, 5 (2014), 104–104.

Qing Mi, Yiqun Hao, Liwei Ou, and Wei Ma. 2022. Towards Using Visual, Semantic and Structural Features to Improve Code Readability Classification. *Journal of Systems and Software* 193 (2022), 111454.

Qing Mi, Jacky Keung, Yan Xiao, Solomon Mensah, and Yujin Gao. 2018. Improving Code Readability Classification Using Convolutional Neural Networks. *Information and Software Technology* 104 (2018), 60–71.

Qing Mi, Yan Xiao, Zhi Cai, and Xibin Jia. 2021. The Effectiveness of Data Augmentation in Code Readability Classification. *Information and Software Technology* 129 (2021), 106378.

NSA. 2023a. Ghidra : Ghidra Change History. https://github.com/NationalSecurityAgency/ghidra/blob/Ghidra_10.3.1_build/Ghidra/Configurations/Public_Release/src/global/docs/ChangeHistory.html

NSA. 2023b. Ghidra Decompiler. https://ghidra-sre.org/

Delano Oliveira, Reydne Bruno, Fernanda Madeiral, and Fernando Castor. 2020. Evaluating Code Readability and Legibility: An Examination of Human-Centric Studies. In *Proceedings of the 36th IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 348–359.

Oracle. 2023. Java. https://www.java.com.

Jihee Park, Sungho Lee, Jaemin Hong, and Sukyoung Ryu. 2023. Static Analysis of JNI Programs via Binary Decompilation. *Journal of the IEEE Transactions on Software Engineering* 49 (2023), 3089–3105.

Fabiano Pecorelli, Fabio Palomba, Dario Di Nucci, and Andrea De Lucia. 2019. Comparing Heuristic and Machine Learning Approaches for Metric-based Code Smell Detection. In *Proceedings of the IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*. IEEE, 93–104.

Daryl Posnett, Abram Hindle, and Premkumar Devanbu. 2011. A Simpler Model of Software Readability. In *Proceedings of the 8th Working Conference on Mining Software Repositories (MSR)*. ACM, 73–82.

Radare. 2023. Radare : Libre and Portable Reverse Engineering Framework. https://www.radare.org/n/

Andreas Rumpf. 2022. A Statically Typed Compiled Systems Programming Language. https://nim-lang.org/.

Simone Scalabrino, Mario Linares-Vásquez, Rocco Oliveto, and Denys Poshyvanyk. 2018. A Comprehensive Model for Code Readability. *Journal of Software: Evolution and Process* 30, 6 (2018), 1958.

Simone Scalabrino, Mario Linares-Vasquez, Denys Poshyvanyk, and Rocco Oliveto. 2016. Improving Code Readability Models with Textual Features. In *Proceeding of the 24th IEEE International Conference on Program Comprehension (ICPC)*. IEEE, 1–10.

Eric Schulte, Jason Ruchti, Matt Noonan, David Ciarletta, and Alexey Loginov. 2018. Evolving Exact Decompilation. In *Proceedings of The Workshop on Binary Analysis Research (BAR)*.

Yash Shejwal, Virat Tiwari, Rewa Wader, and Aditya Warghane. 2023. Decompilers in Reverse Engineering. https://medium.com/@raw.rewa10/decompilers-and-reverse-engineering-6b4acf3f76ff

Eui Chul Richard Shin, Dawn Song, and Reza Moazzezi. 2015. Recognizing Functions in Binaries with Neural Networks. In *Proceedings of the 24th USENIX Security Symposium (Security)*. Washington, DC.

Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2016. SoK: (State of) The Art of War: Offensive Techniques in Binary Analysis. In *Proceeding of the 37th IEEE Symposium on Security and Privacy (SP)*.

Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2023a. Angr : Github. https://github.com/angr/angr/tree/master

Yan Shoshitaishvili, Ruoyu Wang, Christopher Salls, Nick Stephens, Mario Polino, Audrey Dutcher, John Grosen, Siji Feng, Christophe Hauser, Christopher Kruegel, and Giovanni Vigna. 2023b. Angr Documentation : Angr Changelog. https://docs.angr.io/en/latest/appendix/changelog.html

Yahya Tashtoush, Zeinab Odat, Izzat M Alsmadi, and Maryan Yatim. 2013. Impact of programming features on code readability. *International Journal of Software Engineering and Its Applicati* 7, 6 (2013), 441–458.

Katerina Troshina, Yegor Derevenets, and Alexander Chernov. 2010. Reconstruction of Composite Types for Decompilation. In *Proceeding of the 10th IEEE Working Conference on Source Code Analysis and Manipulation*. IEEE, 179–188.

Guido Van Rossum, Fred L Drake, et al. 2022. Python Reference Manual. https://www.python.org/psf-landing/

Vector35. 2023a. Binary Ninja. https://binary.ninja

Vector35. 2023b. Decompiler Explorer : A Web Front-end to a Number of Decompilers. https://dogbolt.org/

Wargio et al. 2023a. R2dec-js : Github. https://github.com/wargio/r2dec-js

Wargio et al. 2023b. R2dec-js : Github : Releases. https://github.com/wargio/r2dec-js/releases

Jordan Wiens, Kyle Martin, Peter LaFosse, Alexander Taylor, Xusheng Li, Rusty Wagner, Andrew Lamoureux, Jon Palmisciano, Stephen Tong, and Brian Potchik. 2023. Binary Ninja : Binary Ninja Change History Blog. https://binary.ninja/blog/

The Program Transformation Wiki. 2023. The Decompilation Wiki. https://www.program-transformation.org/Transform/DeCompilation.html.

Erik Wirtanen. 2022. NSF-funded Project Aims to Mitigate Malware and Viruses by Making Them Easily Understandable. https://fullcircle.asu.edu/faculty/know-thy-enemy/

Khaled Yakdan, Sergej Dechand, Elmar Gerhards-Padilla, and Matthew Smith. 2016. Helping Johnny to Analyze Malware: A Usability-optimized Decompiler and Malware Analysis User Study. In *Proceeding of the 37th IEEE Symposium on Security and Privacy (SP)*. IEEE, 158–177.

Khaled Yakdan, Sebastian Eschweiler, Elmar Gerhards-Padilla, and Matthew Smith. 2015. No More Gotos: Decompilation Using Pattern-Independent Control-Flow Structuring and Semantic-Preserving Transformations. In *Proceeding of the 22nd Annual Network and Distributed System Security Symposium (NDSS)*.

Feng Zhang, Audris Mockus, Ying Zou, Foutse Khomh, and Ahmed E Hassan. 2013. How Does Context Affect the Distribution of Software Maintainability Metrics?. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM)*. IEEE, 350–359.

Received 2023-09-29; accepted 2024-01-23