Automatic Fixation of Decompilation Quirks Using Pre-Trained Language Model

Ryunosuke Kaichi, Shinsuke Matsumoto, Shinji Kusumoto

Graduate School of Information Science and Technology, Osaka University, Japan

Abstract. Decompiler is a system for recovering the original code from bytecode. A critical challenge in decompilers is that the decompiled code contains differences from the original code. These differences not only reduce the readability of the source code but may also change the program's behavior. In this study, we propose a deep learning-based quirk fixation method that adopts grammatical error correction. One advantage of the proposed method is that it can be applied to any decompiler and programming language. Our experimental results show that the proposed method removes 55% of identifier quirks and 91% of structural quirks. In some cases, however, the proposed method injected a small amount of new quirks.

Keywords: decompiler · fine-tuning · deep learning · quirk · grammatical error correction

1 Introduction

Decompiler is one of the reverse engineering systems that translate low-level program representation (e.g., binary or bytecode) to human-readable language (e.g., source code) [2]. Decompiler is expected to be applied to various purposes. One of the major applications is to understand the program behavior in an environment where source code cannot be accessed. Famous IDE tools, such as Eclipse and IntelliJ, have a decompiler feature in default. This feature helps developers to analyze the inside of dependent libraries without their source code. Furthermore, a decompiler is one of the important techniques for binary security analysis [3]. Several decompiler-based malware detection methods have been proposed for Android applications [1] [11].

A critical challenge in decompilers is that the decompiled code contains differences from the original code. This paper calls these difference *quirk*. Low-level program languages do not contain identifier information written in the original code. So, the complete identifier reconstruction is fundamentally impossible [8]. It is known that source code identifiers play an important role in source code comprehension [6]. Therefore, identifier quirks become obstacles to applying decompiler for the scenario of program comprehension. Decompilation quirks occur not in identifiers but rather in program structure. It is because a single program instruction in low-level language does not always correspond to high-level language instruction. For example, iteration instruction in low-level language can

be translated to both `for` and `while` statements. This translation is considered an inference problem. Harrand et al. [5] have reported that the decompiled code sometimes behaves differently from the original code by the structural quirks.

The study aims to provide a method to fix decompilation quirks. We propose a deep learning-based quirk fixation method that adopts grammatical error correction (GEC) to achieve this goal. GEC is a well-known technique for detecting and fixing grammatical errors, including in natural language sentences. Our method assumes quirks in the decompiled code, one of the grammatical errors against the original code. The proposed method has a significant advantage in applicability, which can be applied to any decompilers and programming languages. Also, the method has high compatibility with deep learning. A large learning dataset (i.e., a set of pairs of decompiled and original code) can be easily generated with a fully automated. We apply our method to ReCa [10], a program competition dataset, as an evaluation. Evaluation results show that the proposed method removes 55% of identifier quirks and 91% of structural quirks.

2 Decompilation Quirk

This section illustrates decompilation quirks with concrete source code examples. Fig. 1 shows quirk examples with famous Java decompilers, CFR. We can see various quirks in decompiled code. This paper broadly classifies quirks into two types: identifier quirk and structural quirk.

There are two identifier quirks for local variables. As explained in the first section, bytecode does not contain identifier information, especially in local and temporal variables. So, the reconstruction of local variable identifiers is a challenging task. Only the loop index `i` is reconstructed correctly. Probably, CFR has a specific reconstruction rule that follows common sense in which loop indexes should be named `i`, `j`, and `k`. However, we lost almost identifiers, such as `occurrence` and `numbers`, that help program comprehension.

Next, we focus on structural quirks. The `final` modifier for the method parameter has been lost. We cannot grasp the programmer's intent that the parameter is immutable from the decompiled code. Those variables and method modifiers are used to declare constraints for the compiler. So, bytecode does not contain this information. The guard clause to find a sentinel in a loop is merged into the loop condition in `for` statement. This merge causes a further quirk: the sentinel condition is reversed from `==` to `!=`. Postfix operator `++` becomes prefix operator. In summary, though overall behavior is the same as the original code, the decompiled code is slightly difficult to read to grasp its intent.

3 Proposed Quirk Fixation Method

3.1 Overview

This paper proposes an automated fixation method for decompilation quirks. The fundamental idea of our method is to adopt deep learning-based GEC

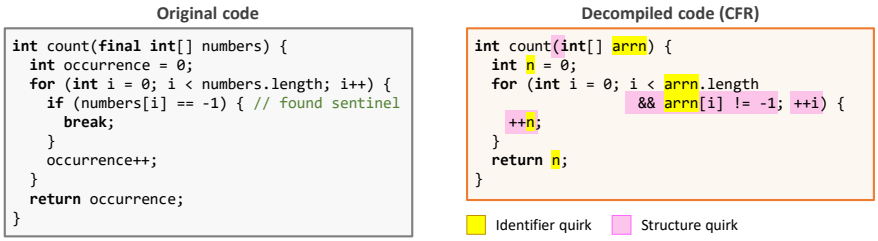

Fig. 1: Example of decompilation quirks in Java

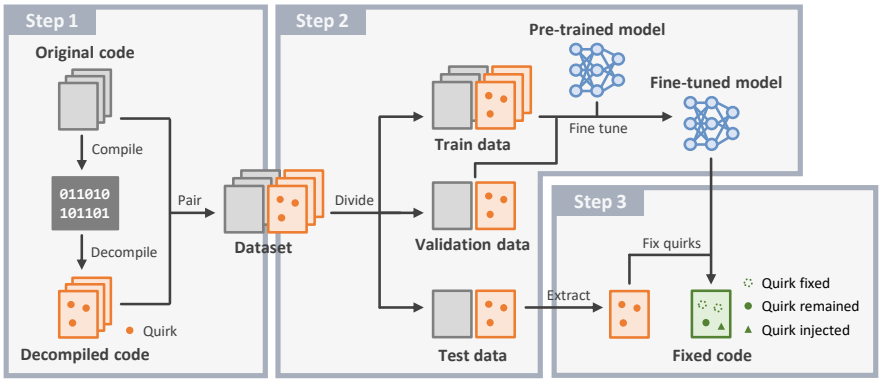

Fig. 2: Flow of quirk fixation by the proposed method

by assuming a decompilation quirk as a syntax error. Our method leverages CodeT5 [13], a Transformer-based pre-trained model, applying fine-tuning as a translation task using decompiled and original code pairs. The generated model attempts to translate code with quirks to code without quirks.

3.2 Procedure

Fig. 2 shows the overall procedure of quirk fixation by the proposed method. The method consists of the following three steps.

Step 1. Dataset Generation: In this step, we create a learning dataset of original and decompiled code pairs. First, we obtain source code from an arbitrary data source such as GitHub or a public dataset. Next, all source code is compiled to generate byte code or machine code, depending on the language. Then, we recover the source code from the bytecode using a decompiler. At this point, decompiled code contains certain quirks. Finally, original and decompiled codes are paired as a dataset.

Step 2. Fine-Tuning: Next, we generate a quirk fixation model by fine-tuning. First, we divide the dataset generated in Step 1 into three types: training data, validation data, and test data. The split rates are 80%, 15%, and 5%, respectively. Then, CodeT5 is fine-tuned by using training data and validation

data. The learning task is a translation task using a paired dataset. Finally, we obtain a model that considers quirks as grammatical errors and corrects them.

Step 3. Quirk Fixation: Finally, we attempt to fix decompilation quirks using the generated model. The input to the model generated by fine-tuning is decompiled codes contained in test data. As a result, the model generates fixed codes with certain quirks removed.

4 Experimental Setup

4.1 Purpose

The experiment aims to confirm the extent to which identifier quirks and structural quirks in the decompiled code have been fixed.

This experiment focuses on Java as the programming language and CFR as the decompiler. The decompiled code by CFR has the slightest difference from the AST of the original code and the second-highest compilability rate among Java decompilers [5]. As described in Section 3.1, our method can be applied to any programming language and decompiler. Further experiments will be conducted for several decompilers and programming languages.

4.2 Definition of Decompilation Quirks

We define quirks as differences in the AST between the original code and the decompiled code. Quirk is defined as the following two types.

Identifier Quirks: Identifier quirks are differences related to changes in identifier names. We classify the updates of nodes whose labels are SimpleName or QualifiedName in the AST as identifier quirks. Quirks highlighted in yellow in Fig. 1 are one of the identifier quirks. Since identifier names are lost at compile time, it is difficult for the decompiler to recover them. Even if the variables have meaningful names in the original code, the names are changed after decompiling. It leads to a decrease in program comprehension.

Structural Quirks: Structural quirks are differences related to changes in the syntax of the source code. All differences excluding identifier quirks, are classified as structural quirks. The reversal of the finding sentinel condition presented in Fig. 1 is one of the structural quirks. Structural quirks lead to reduced readability as with identifier quirks. It could also affect the behavior of the program.

4.3 Quirk Evaluation

As an experiment procedure, we first detect the difference in AST between the original code and the decompiled code and between the original code and the fixed code. These differences are considered a set of quirks. Then, we confirm how

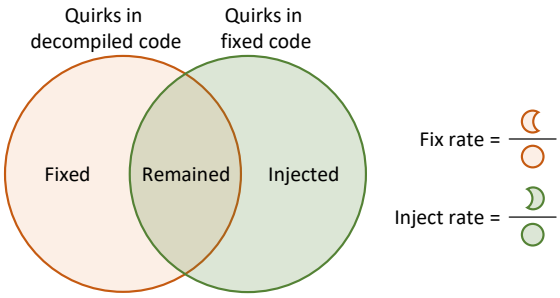

Fig. 3: Inclusion relation of two quirk sets

many quirks in the decompiled code have been fixed by analyzing the inclusion relationship between the two quirk sets. We use GumTree [4] to detect differences in AST. GumTree is a tool that can identify changes between two programs.

Fig. 3 shows a Venn diagram representing the inclusion relationship between the set of quirks in the decompiled code and the set of quirks in the fixed code. We define the fix rate as the percentage of fixed quirks in the decompiled code. The fix rate is calculated as fixed/fixed+remained. The proposed method may inject new quirks. The percentage of injected quirks among the quirks in the fixed code is defined as the inject rate. The inject rate is calculated as injected/injected+remained.

4.4 Dataset

In the experiment, we use ReCa [10], a program competition dataset. ReCa contains source code for four programming languages, C, C++, Python, and Java, but we only use Java. The bytecode generated by the compiler is required to obtain the decompiled code. Therefore, we extracted only compilable source code. Furthermore, only codes with a file size of 2 KB or less were subjected to the experiment. Initially, the experiment was conducted without selection by file size. As a result, the fine-tuning of the model stopped halfway through due to memory problems. We considered both whether it would not cause memory problems and whether the number of data used for fine-tuning was sufficient. Finally, the experimental target was set to 2 KB or less source code. As a result of extracting codes that satisfy the above conditions, the number of source codes gathered was 17,220. These were divided in the ratio of 80:15:5 and used as training, validation, and test data, respectively.

4.5 Pre-Trained Model

As the pre-trained model, we use CodeT5 [13], proposed by Wang et al. CodeT5 is a Transformer-based model pre-trained on the CodeSearchNet [7] dataset. It can multitask, including code generation, transformations, and modification. CodeT5 has several models of different sizes. We use CodeT5-small due to memory problems. Fine-tuning took approximately 90 minutes.

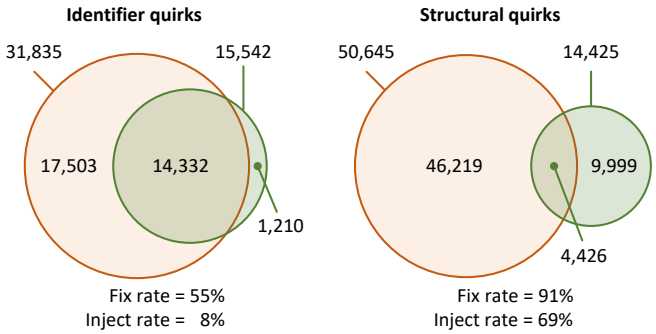

Fig. 4: Venn diagram of identifier quirks and structural quirks

5 Results and Discussion

5.1 Results

Fig. 4 shows Venn diagrams for the set of identifier quirks and the set of structural quirks. The numbers in the Venn diagram represent the total quirk of all 861 source codes in test data. Focusing on identifier quirks, the total number in the decompiled code was 31,835. Of these, 17,503 were removed by the proposed method, and the fix rate of identifier quirks was about 55%. On the other hand, the proposed method injected a relatively small number of identifier quirks, with an inject rate of 8%. Next, the total number of structural quirks in the decompiled code was 50,645. The proposed method removed 46,219, which is about 91%. On the other hand, new structural quirks injected by the proposed method were 9,999. The inject rate was 69%, which was higher than identifier quirks.

A more detailed analysis can be shown in Fig. 5. This figure represents an actual example of quirk fixation by the proposed method. A part of the source code has been abbreviated to make it easier to understand the fixation's effect. The structural quirks highlighted in pink is the most important one to focus on. The original code uses a variable of type `boolean` as a flag, determining the final output. In contrast, the decompiled code uses the value of the variable `n` of type `int` as a flag. The role of variable `n` is difficult to understand intuitively, reducing readability. Furthermore, the conditional expression of the `if` statement in the `for` statement is reversed. Usually, humans code differently. It is precisely the cause of reduced readability. In contrast, the fixed code uses variables of type `boolean` as flags, making their role clear. The `if` statement has also been fixed to be written in the same way as the original code, which is easier to understand and more readable.

Next, note the identifier quirks highlighted in yellow. The variable names for arrays and flags in the fixed code have been changed to intuitively understandable names. Since it is not the same name as the original code, it is a modification failure in this definition. However, the fixed code is relatively superior in terms of readability. It is one of the strengths of the proposed method.

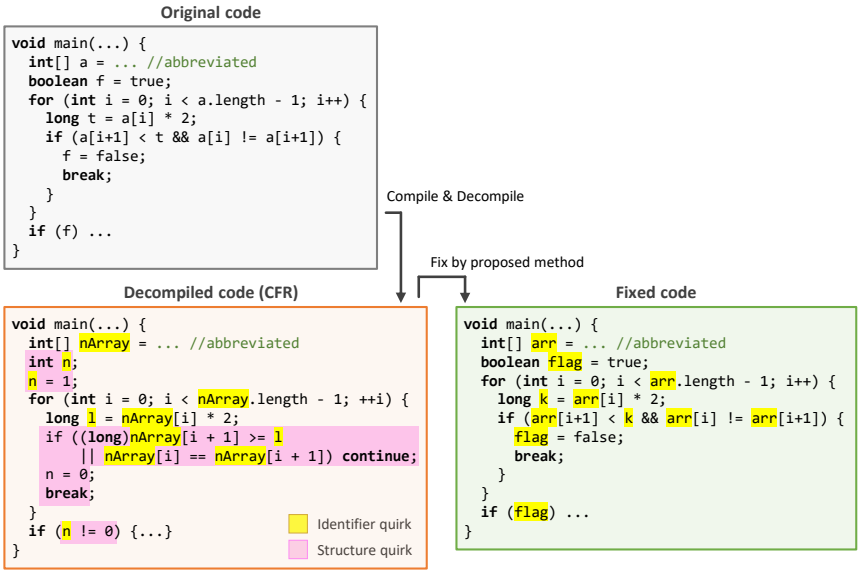

Fig. 5: Example of quirk fixation in a subject with ReCa

5.2 Discussion

We discuss why the fix rate of structural quirks was significantly higher than that of identifier quirks. The pre-trained model used in this study was pre-trained and fine-tuned on a dataset consisting of source code created by multiple developers. The naming of identifiers is highly dependent on the developer, although there is a certain degree of shared understanding. Variables with the same meaning are often given different names, so the correct answer to identifier quirks can vary widely from data to data. However, since the syntax is developer-independent, the correct answer to structural quirks is almost uniquely determined. For these reasons, we consider that the fix rate of structural quirks was significantly higher than that of identifier quirks.

6 Conclusion

We proposed an automated fixation method for two types of quirks in the decompiled code. As a result, we confirmed that the proposed method could fix 55% of identifier quirks and 91% of structural quirks in the decompiled code.

For future work, we first try to conduct experiments with multiple decompilers. The proposed method can fix quirks without depending on decompilers or programming languages. Using more evaluation metrics is also an important task. For practicality, it is necessary to evaluate the fixed code in terms of whether it can be compiled and whether it passes the test cases of the original code. Finally, we will compare with existing methods focusing only on identifier quirks [9] [12].

Acknowledgements

This research was partially supported by JSPS KAKENHI Japan (Grant Number: JP21H04877, JP20H04166, JP21K18302, and JP21K11829)

References

1. Cen, L., Gates, C.S., Si, L., Li, N.: A probabilistic discriminative model for android malware detection with decompiled source code. Transactions on Dependable and Secure Computing (TDSC) **12**(4), 400–412 (2014)
2. Cifuentes, C., Gough, K.J.: Decompilation of binary programs. Software: Practice and Experience **25**(7), 811–829 (1995)
3. Cifuentes, C., Waddington, T., Van Emmerik, M.: Computer security analysis through decompilation and high-level debugging. In: Working Conference on Reverse Engineering (WCRE). pp. 375–380 (2001)
4. Falleri, J., andXavier Blanc, F.M., Martinez, M., Monperrus, M.: Fine-grained and accurate source code differencing. In: International Conference on Automated Software Engineering (ASE). pp. 313–324 (2014)
5. Harrand, N., Soto-Valero, C., Monperrus, M., Baudry, B.: The strengths and behavioral quirks of java bytecode decompilers. In: International Working Conference on Source Code Analysis and Manipulation (SCAM). pp. 92–102 (2019)
6. Hofmeister, J., Siegmund, J., Holt, D.: Shorter identifier names take longer to comprehend. In: International Conference on Software Analysis, Evolution and Reengineering (SANER). pp. 217–227 (2017)
7. Husain, H., Wu, H.H., Gazit, T., Allamanis, M., Brockschmidt, M.: Codesearchnet challenge: Evaluating the state of semantic code search. arXiv preprint arXiv:1909.09436 (2019)
8. Jaffe, A., Lacomis, J., Schwartz, E.J., Goues, C.L., Vasilescu, B.: Meaningful variable names for decompiled code: A machine translation approach. In: International Conference on Program Comprehension (ICPC). pp. 20–30 (2018)
9. Lacomis, J., Yin, P., Schwartz, E., Allamanis, M., Le Goues, C., Neubig, G., Vasilescu, B.: Dire: A neural approach to decompiled identifier naming. In: International Conference on Automated Software Engineering (ASE). pp. 628–639 (2019)
10. Liu, H., Shen, M., Zhu, J., Niu, N., Li, G., Zhang, L.: Deep learning based program generation from requirements text: Are we there yet? Transactions on Software Engineering (TSE) **48**(4), 1268–1289 (2022)
11. Milosevic, N., Dehghantanha, A., Choo, K.K.R.: Machine learning aided android malware classification. Computers and Electrical Engineering **61**, 266–274 (2017)
12. Nitin, V., Saieva, A., Ray, B., Kaiser, G.: Direct: A transformer-based model for decompiled identifier renaming. In: Workshop on Natural Language Processing for Programming (NLP4Prog). pp. 48–57 (2021)
13. Wang, Y., Wang, W., Joty, S., Hoi, S.C.: CodeT5: Identifier-aware unified pre-trained encoder-decoder models for code understanding and generation. arXiv preprint arXiv:2109.00859 (2021)