

CODE CLONES: A NOVEL APPROACH TO DETECTING PLAGIARISM IN BINARY DECOMPOSITION OF C PROGRAMS

Marek HORVÁTH, Filip GURBAL'

Department of Computers and Informatics, Faculty of Electrical Engineering and Informatics,
Technical University of Košice, Letná 9, 042 00 Košice, Slovak Republic, Tel. +421 55 602 3175,
E-mails: marek.horvath@tuke.sk, filip.gurbal@tuke.sk

ABSTRACT

The escalation of code plagiarism in computer science education has necessitated the development of more sophisticated detection methods, particularly for binary files which pose a unique challenge. This study introduces a novel approach to plagiarism detection through binary decomposition, which decompiles binary files back into high-level code to reveal similarities that may be hidden at the binary level. The methodology is crucial for scenarios where students submit compiled programs, enabling the application of traditional code similarity analysis methods to detect instances of plagiarism. Statistical data on code similarity among student submissions in a C programming course are presented, highlighting the urgent need for robust detection strategies. This paper explores the ethical implications of plagiarism, the technical challenges of binary file analysis, and the potential for applying this technique across various programming languages and educational contexts.

Keywords: Plagiarism Detection, Binary Decomposition, Computer Science Education, Algorithmic Analysis, Pattern Recognition, Academic Integrity,

1. INTRODUCTION

In the evolving landscape of computer science education, a significant increase in student enrollment in introductory programming courses, particularly in the C language, has presented distinct challenges for educators, especially in maintaining academic integrity. With the widespread availability of online resources and shared code repositories, the boundaries of code originality often become vague, leading to incidents of plagiarism, whether intentional or accidental. However, the issue becomes more complex when dealing with binary files, which require a different approach than standard source code plagiarism detection.

The phenomenon of "code clones", or sections of code that are either identical or highly similar, is a familiar concept in software development. However, its application in the context of educational settings, especially for binary files, is still a relatively uncharted area. Our study introduces a new methodology for detecting plagiarism through binary decomposition. This method involves decompiling binary files back into a higher-level code format, enabling the detection of similarities that might be hidden in the binary state.

This method is particularly crucial in scenarios where students are asked to submit compiled programs. Decompiling these binaries allows for the application of traditional code similarity analysis methods to detect potential instances of plagiarism.

In this paper, we demonstrate the urgency and relevance of this approach by presenting statistical data on the occurrence of similar code patterns among student submissions in a basic C programming course. This data does not originate from an experiment with the students but rather serves to demonstrate the widespread nature and seriousness of plagiarism issues in such foundational courses.

Besides presenting our new detection method, this

paper discusses the ethical implications of plagiarism in academic settings, the unique challenges of identifying it in binary files, and the prospects of applying this technique across different programming languages and educational scenarios. Our objective is to contribute to the ongoing conversation about academic honesty in computer science education and to provide educators with more effective tools for maintaining it.

Through the dissemination of our methodology and insights, we aim to promote further research and progress in this field, leading to the creation of stronger plagiarism detection strategies and a deeper understanding of how students engage with programming assignments.

2. RELATED WORK

The challenge of plagiarism in computer programming has been extensively researched. Muxin's study on code cloning highlights the widespread occurrence of identical or similar source code fragments and the various methods students use to hide plagiarism [1]. This includes changing variable names, modifying comments, and reformatting code.

Research conducted by Liu and colleagues focuses on plagiarism detection in the online education context, noting that students codes often have similar structures due to identical solutions to course tasks [2]. Their findings suggest that detection should concentrate on superficial changes rather than different control statements and logic.

Yu et al. examine the cultural and educational differences in perceptions of plagiarism, focusing on a significant case of academic dishonesty at RMIT University [3]. This study underlines the need for a more thorough understanding of the reasons behind plagiarism among different student groups.

Karnalim's review summarizes various automated similarity detection methods, tracing their evolution and effectiveness over time [4]. The study highlights JPlag

and MOSS as important tools in the field, owing to their practical utility.

Ragkhitwetsagul et al. introduce the concept of normalization by decompilation as a means to enhance similarity measurements in extensively modified source codes [5]. Hrkút and his team investigate the application of natural language processing methods in analyzing source code [6], emphasizing the potential of graph representations for detecting code similarities.

Fonseca and Macedo discuss the use of existing plagiarism detection methods for real-time application in programming classes [7]. Their study stresses the significance of early detection to assist teachers in addressing students learning needs effectively.

Ljubovic's research on evaluating plagiarism detection tools highlights the challenges in establishing similarity thresholds and the importance of correctly identifying both false positives and negatives [8]. Aniceto's work provides an overview of the increasing number of studies in source code plagiarism detection [9], showcasing the varied approaches and international research efforts in this area.

These studies collectively emphasize the complexity of plagiarism detection in computer programming. They highlight the need for ongoing improvements in detection methodologies and a detailed understanding of the various factors that contribute to plagiarism.

3. ADDRESSING PLAGIARISM CHALLENGES IN MASSIVE COURSES

Plagiarism in academic work, especially within massive computer science courses, is a significant concern. This issue challenges the integrity of academic standards and impedes the genuine development of problem-solving and coding skills among students. The nature of these courses, often covering fundamental concepts of algorithmization, combined with the easy access to similar code and solutions online, increases the likelihood of plagiarism.

The concern is grounded in reality, as evident from solid evidence. Table 1 shows an analysis of plagiarism rates in freshman-level assignments within these courses, revealing a worrying trend. The assignments, integral to basic algorithmization education, consistently demonstrate instances of plagiarism over the years. This highlights the severity of the issue, particularly among new students.

The data indicates a growing trend in plagiarism, pointing not only to a lack of comprehension of core concepts but also to a larger issue of academic dishonesty. The challenge is further amplified in the context of binary files, where conventional plagiarism detection methods may fall short. As such, developing and implementing effective detection tools and methodologies becomes essential to uphold the academic integrity and educational quality in these massive courses.

Ethically, plagiarism undermines the educational process and devalues the efforts of honest students. It is unethical for students to cheat by submitting someone else's work as their own. This practice not only slows down the learning process but also supports an underground market where assignments are bought and sold, allowing students

to bypass genuine learning. In large courses, the sheer volume of submissions makes it impractical to manually evaluate each one for plagiarism, necessitating automated solutions. Our approach, through binary decomposition, helps address this need by providing a more effective means of detecting plagiarism, ensuring that students are evaluated fairly and accurately based on their own work.

Currently, our focus is primarily on the C programming language. This is because most of the fundamental courses in our department, such as fundamentals of algorithmization, programming, data structures, and operating systems, are taught in C. The problem of plagiarism is most prevalent in these beginner courses, which is why our priority has not extended to other programming languages. Students typically encounter other languages at higher levels of their education, where plagiarism is less of an issue compared to these introductory courses.

The next section discusses the specific tools and decompilers selected for our study, aimed at effectively addressing these challenges.

4. TOOLS AND DECOMPILERS

In our study, we carefully selected a set of tools for code similarity checking and binary decompiling based on their efficiency, popularity in academic and professional environments, and their sophisticated algorithmic capabilities. For code similarity checking, we continue to rely on JPlag and Sherlock. Additionally, for the crucial task of decompiling, we have chosen a range of tools: Angr, BinaryNinja, Ghidra, Hex-rays, Reko, Relyze, and RetDec. This diverse selection ensures a comprehensive approach to analyzing binary files.

4.1. Plagiarism detectors - JPlag and Sherlock

JPlag and Sherlock remain our primary tools for code similarity checking. JPlag offers an effective syntax-based analysis and supports multiple programming languages, including Java, C, C++, and Python. It utilizes the Running Karp-Rabin Greedy String Tiling algorithm, known for its effectiveness in detecting code similarities. Sherlock operates on a tokenization algorithm, making it universally applicable across various programming languages, although it may not analyze the structural nuances of the code.

4.2. Decompilers

The selection of decompilers is pivotal in our methodology, as it allows us to reconstruct high-level code from binary files, which is essential for our plagiarism detection process.

Angr - A dynamic binary analysis tool known for its versatility in handling different binary formats. Angr employs symbolic execution and control flow analysis, crucial for reconstructing high-level representations of binary code.

BinaryNinja - Known for its user-friendly interface and comprehensive analysis capabilities. BinaryNinja provides

in-depth insights into binary code, essential for accurate decompiling and analysis of complex software.

Ghidra - Developed by the National Security Agency (NSA), Ghidra is a comprehensive suite of software reverse engineering tools, capable of handling various platforms and processors, making it invaluable for decompiling binaries into readable formats.

Hex-rays - Acclaimed for its accuracy in decompilation, Hex-rays translates binary programs into high-level representations with precision, aiding significantly in the analysis of compiled code.

Reko - An open-source decompiler focusing on .NET assemblies and machine code. Reko's simplicity and effectiveness in converting binaries into a high-level, readable format make it a valuable addition to our toolset.

Relyze - Offers an interactive approach to software reverse engineering, incorporating modern decompiling techniques and providing detailed insights into the structure and behavior of binary code.

RetDec - Notable for its universality and open-source nature, RetDec supports a wide range of executable formats and is capable of decompiling binaries into a C-like language, making it highly effective for our purposes.

The integration of these tools allows us to establish a robust framework for detecting plagiarism in binary files. Each tool contributes its unique strengths, enhancing the reliability of our findings and the practical applicability of our study.

5. ANALYZING DECOMPILERS AND SIMILARITY CHECKERS THROUGH A FACTORIAL PROGRAM EXPERIMENT

In this experiment, we aimed to evaluate the performance of various decompilers and code similarity checkers using simple programs written in C, including a factorial program, a prime numbers program, and a Fibonacci sequence program. The process involved compiling the programs, decompiling them using different tools, and then analyzing the decompiled code for similarities.

The C program used for this example is a basic implementation of a factorial calculation:

Factorial Program

```
#include <stdio.h>

int main() {
    int n, i;
    unsigned long long factorial = 1;

    printf("Enter an integer: ");
    scanf("%d", &n);

    if (n < 0)
        printf("Error! Factorial of a negative number doesn't exist.");
    else {
        for (i = 1; i <= n; ++i) {
            factorial *= i;
        }
        printf("Factorial of %d = %llu",
            n, factorial);
    }
}
```

```
}

return 0;
}
```

After compiling this program, we used various decompilers, including Relyze, to convert the compiled binaries back to high-level code. Below is a snippet of the decompiled factorial program from Relyze:

Snippet of decompiled factorial program from Relyze

```
void deregister_tm_clones() {
    if (&g_b4010 == &g_b4010)
        return;
}

void register_tm_clones() {
    if (&g_b4010 == &g_b4010)
        return;
}

void __do_global_dtors_aux() {
    if (g_b4010 != 0)
        return;
    deregister_tm_clones();
    g_b4010 = 1;
}

void frame_dummy() {
    register_tm_clones();
}

void fn00000000000001080(char *rdi) {
    if (rdi != NULL) {
        printf("%s", rdi);
    }
}

void fn00000000000001090(const char *format, ...)
{
    va_list args;
    va_start(args, format);
    vsprintf(format, args);
    va_end(args);
}
```

After compiling this program, we used various decompilers, including Angr, BinaryNinja, Ghidra, Hex-rays, Reko, Relyze, and RetDec, to convert the compiled binaries back to high-level code. We then analyzed the decompiled results for similarity using JPlag and Sherlock, two established code similarity checkers.

The process of decompilation introduces several structural changes that can alter the original code's structure and logic. These changes are primarily due to the different ways decompilers interpret and reconstruct the binary code back into high-level code. For instance, variable names are often replaced with generic identifiers, additional functions and control structures that manage low-level operations are introduced, and some decompilers may insert placeholder code for error handling or system calls. Moreover, the sequence and formatting of the reconstructed code can vary significantly from the original, making it difficult for tools like JPlag to recognize patterns and similarities. These structural modifications, combined with potential differences in how loops and conditionals are represented, lead to significant discrepancies that prevent effective similarity detection by JPlag. This is why matches are not

found, as the algorithmic essence of the original program becomes obfuscated through these transformations.

Figures 1, 2, and 3 show the similarity comparison results using JPlag for the factorial program, the prime numbers program, and the Fibonacci sequence program, respectively. The matrices illustrate the extent of similarity detected between the original and decompiled codes by different tools. It is interesting to note the level of similarity in the structural interpretation of the binary by these decompilers, as reflected in their outputs.

Jplag	Original	Angr	BinaryNinja	Ghidra	Hex-rays	Reko	Relyze	RetDec
Original	0	0	0	0	0	0	0	0
Angr	0	0	0	0	0	0	0	0
BinaryNinja	0	0	0	0	0	0	0	15.3
Ghidra	0	0	0	0	21.1	0	0	0
Hey-rays	0	0	0	21.1	0	0	0	0
Reko	0	0	0	0	0	0	100	27.3
Relyze	0	0	0	0	0	100	0	27.3
RetDec	0	0	15.3	0	0	27.3	27.3	0

Fig. 1 Decompile Similarity Comparison Using JPlag for Factorial Program

Jplag	Original	Angr	BinaryNinja	Ghidra	Hex-rays	Reko	Relyze	RetDec
Original	0	0	0	0	0	28.8	0	0
Angr	0	0	0	0	0	0	0	0
BinaryNinja	0	0	0	0	0	0	0	10
Ghidra	0	0	0	0	0	0	0	0
Hey-rays	0	0	0	0	0	0	0	0
Reko	28.8	0	0.1	0	0	0	0	0
Relyze	0	0	0	0	0	0	0	0
RetDec	0	0	10	0	0	0	0	0

Fig. 2 Decompile Similarity Comparison Using JPlag for Prime Numbers Program

Jplag	Original	Angr	BinaryNinja	Ghidra	Hex-rays	Reko	Relyze	RetDec
Original	0	0	0	25.1	0	0	0	0
Angr	0	0	0	0	0	0	0	0
BinaryNinja	0	0	0	0	0	0	0	0
Ghidra	25.1	0	0	0	0	12	0	0
Hey-rays	0	0	0	0	0	0	0	0
Reko	0	0	0	12	0	0	0	0
Relyze	0	0	0	0	0	0	0	0
RetDec	0	0	0	0	0	0	0	0

Fig. 3 Decompile Similarity Comparison Using JPlag for Fibonacci Sequence Program

A comparable analysis was performed using Sherlock, as shown in Figure 4. Sherlock's method of similarity checking, which focuses on tokenization, provided alternative insights. For example, the comparison between the original and decompiled codes under Sherlock's analysis revealed a more varied range of similarity scores, indicating a different perspective in detecting code similarities based on token patterns.

Sherlock	Original	Angr	BinaryNinja	Ghidra	Hex-rays	Reko	Relyze	RetDec
Original	0	6	9	10	15	22	22	9
Angr	6	0	6	10	13	10	10	10
BinaryNinja	9	6	0	7	13	11	11	7
Ghidra	10	10	7	0	17	14	14	7
Hey-rays	15	13	13	17	0	17	17	14
Reko	22	10	11	14	17	0	100	21
Relyze	22	10	11	14	17	100	0	21
RetDec	9	10	7	7	14	21	21	0

Fig. 4 Decompile Similarity Comparison Using Sherlock

These findings emphasize the different effectiveness of various decompilers and the subtle differences in how code similarity is identified by JPlag and Sherlock. The study highlights the importance of using diverse methods for plagiarism detection in binary decomposition, crucial in educational contexts where the originality of code submissions is critical.

6. DISCUSSION

The results of our experiment, as shown in the provided matrices (see Figures 1, 2, 3 and 4), indicate several noteworthy trends and patterns in the similarity scores produced by JPlag and Sherlock when analyzing the decompiled outputs of a factorial C program.

JPlag's results show that most decompiled outputs do not have significant similarities with each other, as indicated by the mostly zero scores across the matrix. This implies that the structural changes during decompilation were significant enough to prevent detection by JPlag's algorithm. However, notable exceptions were seen with Reko, Relyze, and RetDec, where similarity percentages reached as high as 100% and 27.3% respectively. These high scores may suggest that certain decompilers produce outputs with structural patterns or code sequences that are more recognizable as similar.

In contrast, Sherlock's tokenization-based similarity checking provided a broader range of similarity scores across all decompiled outputs. Unlike JPlag, Sherlock detected some level of similarity even between the original code and the decompiled versions, with scores ranging from 6% to 22%. These results highlight Sherlock's sensitivity to token-level changes and its ability to detect more subtle forms of similarity that might be missed by structural analysis alone.

The high similarity scores between the outputs of Reko and Relyze (100%) suggest that these tools might use similar decompiling strategies or that their outputs preserve a significant part of the original program's structure. Additionally, the consistent similarity scores among various decompiler pairs point to possible areas for improvement for these tools to vary their decompiling approaches and further decrease the chance of producing similar outputs.

The year-over-year increase in plagiarism percentages in student submissions for 'Assignment A' from 4.7% to 7.8%, as seen in the table, highlights the increasing challenge of plagiarism detection. It indicates that despite advancements in detection tools, the occurrence of

plagiarism is rising, perhaps due to easier access to similar code through various resources.

Evaluating the false-positive rate is crucial to understanding the effectiveness and reliability of plagiarism detection methods. The false-positive rate can be calculated as follows:

$$\begin{aligned}\text{FalsePositiveRate} &= \frac{\text{FalsePositive}}{\text{FalsePositive} + \text{TrueNegative}} \\ &= \frac{6}{6 + 34} \\ &= \frac{6}{40} \\ &= 0.15\end{aligned}$$

This rate indicates that 15% of the non-plagiarized cases were incorrectly identified as plagiarized. Sometimes there are few correct solutions, making it difficult to assess borderline cases. A positive result does not necessarily mean that the code must be a plagiarism, but it is an indicator that it could be.

These findings emphasize the value of using a mix of similarity detection tools to thoroughly analyze code for potential plagiarism. The combined strengths of JPlag and Sherlock, when used together, offer a more effective mechanism for detecting a broad range of similarities, from token-level to structural ones.

7. CONCLUSION

In conclusion, the findings from our study highlight the complex nature of plagiarism detection in binary files and emphasize the need for a comprehensive approach. The comparative analysis using well-known tools such as JPlag and Sherlock showed that while some decompilers produce outputs with distinct structural patterns, others may result in similar structures. This highlights the importance of having a variety of decompilation techniques. The increase in plagiarism rates over the years necessitates ongoing improvements in detection processes and the integration of advanced technologies. Moving forward, it's crucial for educational institutions to embrace these thorough strategies to safeguard the integrity of academic submissions and uphold the high standards of computer science education. The insights gained from this study into decompiler performance and similarity detection are a significant contribution to the field, providing direction for future research and the advancement of more effective plagiarism detection methods.

ACKNOWLEDGEMENTS

We extend our sincere thanks to the Department of Computers and Informatics (DCI) at the Technical University of Košice (TUK) for providing access to their comprehensive dataset of student assignments. This support has been crucial in enhancing the scope and impact of our research.

Table 1 Plagiarism Statistics for Course Fundamentals of Algorithms and Programming Assignments for the Years 2021/2022/2023

Assignment	Year	Submissions	Plagiarisms	Plagiarism Percentage (%)
Assignment A	2021	817	39	4.7
	2022	1018	124	12.2
	2023	909	71	7.8
Assignment B	2021	732	91	12.4
	2022	930	94	10.1
	2023	824	40	4.9
Assignment C	2021	734	42	5.7
	2022	921	55	6
	2023	837	64	7.6
Assignment D	2021	414	25	6
	2022	435	17	3.9
	2023	860	33	3.8
Assignment E	2021	587	36	6.1
	2022	803	50	6.2
	2023	682	36	5.3

REFERENCES

- [1] LI, M.: "Research on Code Plagiarism Detection Based on Code Clone Detection Technologies," in *2021 2nd International Conference on Big Data and Informatization Education (ICBDIE)*, 2021, pp. 274-277. doi: 10.1109/ICBDIE52740.2021.00068
- [2] LIU, T. – ZHAO, Z. – FANG, H. – HUANG, Q. – ZHANG, W.: "Design and Implementation of Code Plagiarism Detection System," in *2023 4th International Seminar on Artificial Intelligence, Networking and Information*

- Technology (AINIT)*, 2023, pp. 188-195. doi: 10.1109/AINIT59027.2023.10212887
- [3] YU, L. – JIANG, H. – ZHU, H. – ZHAO, Q. – CHEN, J.: “Investigating the Understanding of Plagiarism: A Case Study of Code Plagiarism in China,” in *2020 15th International Conference on Computer Science & Education (ICCSE)*, 2020, pp. 176-181. doi: 10.1109/ICCSE49874.2020.9201827
- [4] KARNALIM, O. – CHIVERS, W. S.: “Similarity Detection Techniques for Academic Source Code Plagiarism and Collusion: A Review,” in *2019 IEEE International Conference on Engineering, Technology and Education (TALE)*, 2019, pp. 1-8. doi: 10.1109/TALE48000.2019.9225953
- [5] RAGKHITWETSAGUL, CH. – KRINKE, J. – CLARK, D.: “Similarity of Source Code in the Presence of Pervasive Modifications,” in *2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, 2016, pp. 117-126. doi: 10.1109/SCAM.2016.13
- [6] HRHÚT, P. – ĎURAČÍK, M. – TOTH, Š. – MEŠKO, M.: “Current Trends in the Search for Similarities in Source Codes with an Application in the Field of Plagiarism and Clone Detection,” in *2023 33rd Conference of Open Innovations Association (FRUCT)*, 2023, pp. 77-84. doi: 10.23919/FRUCT58615.2023.10143064
- [7] FONSECA, N. G. – MACEDO, L. – MENDES, A. J.: “Using early plagiarism detection in programming classes to address the student’s difficulties,” in *2018 International Symposium on Computers in Education (SIIE)*, 2018, pp. 1-6. doi: 10.1109/SIIE.2018.8586700
- [8] LJUBOVIC, V. – PAJIC, E.: “Plagiarism Detection in Computer Programming Using Feature Extraction From Ultra-Fine-Grained Repositories,” *IEEE Access*, vol. 8, 2020, pp. 96505-96514. doi: 10.1109/ACCESS.2020.2996146
- [9] ANICETO, R. C. – HOLANDA, M. – CASTANHO, C. – DA SILVA, D.: “Source Code Plagiarism Detection in an Educational Context: A Literature Mapping,” in *2021 IEEE Frontiers in Education Conference (FIE)*, 2021, pp. 1-9. doi: 10.1109/FIE49875.2021.9637155

Received January 15, 2024, accepted June 26, 2024

BIOGRAPHIES

Marek Horváth graduated with a Master’s degree in 2023 at the Department of Computers and Informatics, Technical University of Košice, Slovakia. Currently, he is pursuing his Ph.D. at the same department. The main topic of his research is static code analysis and plagiarism detection.

Filip Gurbál is Ph.D. student at the Department of Computers and Informatics, Technical University of Košice, Slovakia. He graduated at the university in computer science in 2020. The subject of his research is bridging the understanding gap between testers and programmers. He also focuses on software testing methods and tools.