UNIVERSIDADE FEDERAL DO RIO GRANDE DO SUL
INSTITUTO DE INFORMÁTICA
CURSO DE CIÊNCIA DA COMPUTAÇÃO

ROBERTA ROBERT

`ROBiT`: Um Método de Detecção de Plágio Baseado em Otimizações de Arquivos Binários

Monografia apresentada como requisito parcial para a obtenção do grau de Bacharel em Ciência da Computação

Orientador: Prof. Dr. Jeferson de Campos Nobre
Co-orientador: Prof. Dr. Bruno Castro da Silva

Porto Alegre
2024

“ If you steal from one author it’s plagiarism. If you steal from many it’s research.“

—(WILSON MIZNER, 1953)

AGRADECIMENTOS

Eu nunca tive uma conquista pessoal que tenha sido fácil, ou que tenha sido feita completamente sozinha - a todas as pessoas que me ajudaram nestes 14 anos de UFRGS e 32 anos de vida, estes agradecimentos são para vocês.

Aos meus pais, Rogério e Mirian, pela paciência de ter uma filha que decidiu seguir caminhos alternativos durante toda a graduação (talvez vida): obrigada pelo amor cuidadoso, por apoiarem com convicção todos os movimentos arriscados de vida que decidi tomar.

Ao meu irmão, Rodrigo, grande inspiração em muito do que eu faço e sou, muito obrigada por decidir fazer Ciência da Computação em 2000. Obrigada principalmente por estimular e nunca ter invalidado uma irmã de 10 anos que decidiu que queria ser hacker.

À minha cunhada, Tiane, pós-doutora e fonte infinita de inspiração acadêmica: obrigada por sempre me mostrar a verdade sobre a academia, por me fazer vivenciar a parte mágica de se apaixonar por um tema de pesquisa. Este trabalho nunca teria saído sem você.

Ao meu marido, Robson, por ter carinho e paciência infinitas, por sempre ser meu porto seguro e meu lar.

Às minhas amigas, Luiza, Giuli e Vivian, por serem minha inspiração, meu conforto - por sempre estarem presentes na minha vida.

Ao meu orientador, Jeferson, por ser um psicólogo além de orientador, me dando todo o suporte que eu precisava em momentos muito difíceis, não deixando que as dificuldades que foram surgindo tomassem o controle do trabalho, e de mim.

Ao meu co-orientador, Bruno, por ser uma companhia infinita de madrugadas e fins de semana na composição deste trabalho, por não deixar nenhuma dúvida em pé. Obrigada pela paciência.

Ao meu primeiro chefe, Luis Otávio, por ter me dado o meu primeiro emprego dentro da área de segurança - finalmente vou devolver o combinado e me formar!

E por último, ao professor Weber, por ser a principal figura de suporte, apoio e força dentro da área de segurança, e dentro da minha carreira. Por nunca duvidar da minha capacidade, sempre ter me instigado a aprender e me aprofundar nos assuntos, sempre ter me apoiado a sonhar alto. Por ter me ensinado a nunca deixar de ser curiosa e de me divertir.

RESUMO

O aumento significativo da utilização de plataformas remotas de ensino de programação desde a crise do COVID-19, aliado aos recentes progressos na área de Inteligencia Artificial (IA) generativa, trouxeram novos desafios à forma como códigos fonte são tipicamente avaliados em busca de plágios. O esforço necessário para cometer-se plágio diminuiu, seja utilizando técnicas automatizadas de transformação sintática, ou gerando versões alternativas do código fonte através de IA generativa. A maioria das abordagens atuais usadas na construção de ferramentas anti-plágio não são robustas o suficiente para lidar com o novo escopo de tais avanços tecnológicos. Trabalhos existentes na literatura sobre detectores de clonagem de código demonstraram que existem transformações sintáticas que alteram o código de tal forma que os algoritmos de representação sintática/semântica usados na avaliação de plágio (e.g., AST—árvore de sintaxe abstrata; CFG—grafo de controle de fluxo; PDG—grafo de dependência de programa; e Tokenização) não reconhecem o código alterado como um possível clone. Atualmente, a abordagem de otimização de código binário já é reconhecida como uma técnica de ofuscação e manipulação de códigos maliciosos na área de evasão de antivírus, mas pouca atenção tem sido dada ao seu potencial na construção de ferramentas anti-plágio. Neste trabalho, iremos propor um método para detecção de plágio que estende as capacidades demonstradas por abordagens existentes. Nosso método consiste em uma técnica que não apenas analisa códigos fonte, ou apenas os arquivos binários correspondentes resultantes da compilação, e sim efetua uma análise do binário resultante da compilação *com otimizações*. Intuitivamente, o uso de otimizações de compilação serve como um "filtro reverso", removendo do código binário possíveis fontes de ofuscação implementadas pelo plagiador em sua modificação de um código fonte original. Nossos experimentos confirmam esta correlação entre código binário otimizado e seu código fonte original. A avaliação empírica de nosso método demonstra uma utilização para calcular níveis de similaridade entre códigos binários, tendo boa eficácia na detecção de plágios mesmo quando os códigos fontes correspondentes tenham sofrido alterações sintáticas complexas. Dessa forma, o método aqui proposto pode ser utilizado de forma efetiva como uma ferramenta complementar às existentes para efetuar análises de plágio.

Palavras-chave: Anti-Plágio. Ofuscação. Segurança.

ROBiT: Reverse Optimization Binary Transformation Anti-Plagiarism System

ABSTRACT

The significant increase in the use of remote programming education platforms since the COVID-19 crisis, coupled with recent advances in the field of generative Artificial Intelligence (AI), has brought new challenges to the way source codes are typically evaluated for plagiarism. The effort required to commit plagiarism has decreased, whether by using automated syntactic transformation techniques or by generating alternative versions of the source code through generative AI. Most of the current approaches used in the construction of anti-plagiarism tools are not robust enough to deal with the new scope of such technological advances. Existing works in the literature on code cloning detectors have demonstrated that there are syntactic transformations that change the code in such a way that the syntactic/semantic representation algorithms used in plagiarism assessment (e.g., AST—abstract syntax tree; CFG—control flow graph; PDG—program dependency graph; and Tokenization) do not recognize the altered code as a possible clone. Currently, the approach of binary code optimization is already recognized as an obfuscation technique and manipulation of malicious codes in the area of antivirus evasion, but little attention has been given to its potential in the construction of anti-plagiarism tools. In this work, we propose a method for plagiarism detection that extends the capabilities demonstrated by existing approaches. Our method consists of a technique that not only analyzes source codes, or just the corresponding binary files resulting from compilation but also performs an analysis of the binary resulting from the compilation *with optimizations*. Intuitively, the use of compilation optimizations serves as a "reverse filter," removing from the binary code possible sources of obfuscation implemented by the plagiarist in their modification of an original source code. Our experiments confirm this correlation between optimized binary code and its original source code. The empirical evaluation of our method demonstrates a use for calculating levels of similarity between binary codes, having good efficiency in detecting plagiarism even when the corresponding source codes have undergone complex syntactic changes. Thus, we believe that the method proposed here can be effectively used as a complementary tool to existing ones for conducting plagiarism analysis.

Keywords:

.

LISTA DE FIGURAS

LISTA DE TABELAS

SUMÁRIO

1 INTRODUÇÃO

A preocupação com potenciais casos plágio, em aulas do curso de computação, não são recentes. Um dos artigos descrevendo as possíveis implicações de plágio, a apresentando uma das primeiras técnicas de análise de similaridade, datam de 1986 (FAIDHI; ROBINSON, 1987). A progressão de cursos e disciplinas para o formato remoto, principalmente após a crise sanitária do COVID-19, resultou em significativas mudança no mundo da tecnologia. A principal delas talvez seja o sistema de avaliação e revisão de códigos em cursos relacionadas à programação: devido ao crescimento do número de pessoas matriculadas por turma, o tempo médio disponível para a revisão manual de exercícios caiu (DURACIK et al., 2020), fazendo com que a adoção de analisadores automatizados de plágio aumentasse de forma correspondente (AIKEN, 2004).

A pesquisa e adoção de *Large Language Models* (LLM) avança rapidamente, atingindo novos patamares com a popularização do produto ChatGPT—um poderoso gerador textual capaz (dentre outras coisas) de fornecer códigos para uma grande gama de problemas computacionais (YENDURI et al., 2023).

Estas duas condições fazem com que estejamos vivendo uma tempestade perfeita: um momento no tempo no qual técnicas para automatizar plágio são facilmente replicáveis e passível de utilização em larga escala, combinado com o conhecimento técnico relativamente limitado necessário para utilização das ferramentas correspondentes. Mesmo em casos onde ocorre correção manual de códigos fonte, de forma adicional a analisadores automatizados, a detecção de plágio é desafiadora. Já existem frameworks, por exemplo, como o Mossad (DEVORE-MCDONALD; BERGER, 2020), que criam modificações do código fonte inicial sem alterar a sua legibilidade ou a funcionalidade/semântica do programa resultante.

Além de abordagens de IA generativa para geração de código fonte, partindo-se de um código inicial, há também exemplos na literatura que demonstram que LLMs conseguem gerar códigos inatos (GENG et al., 2023), o que significa que um aluno sem conhecimento prévio sobre o problema em questão é capaz de facilmente criar uma versão independente e inata do código necessário para resolução de tal problema, utilizando para isso uma engenharia de prompt simples.

Infelizmente, apesar da popularização e da larga adoção de analisadores antiplágio, em especial do `MOSS` (AIKEN, 2004), seu funcionamentos é comumente baseado em algoritmos de análise sintática (SCHLEIMER; WILKERSON; AIKEN, 2003; LIU et

al., 2006; WISE, 1996; PRECHELT; MALPOHL; PHILIPPSEN, 2002; NICHOLS et al., 2019). De fato, uma ampla gama de técnicas de evasão foram propostas contra a classe de algoritmos anti-plágio baseados en reconhecimento sintático (DEVORE-MCDONALD; BERGER, 2020; ZHANG et al., 2023; BIDERMAN; RAFF, 2022). As técnicas de mitigação de tais riscos normalmente apontam para uma solução multidisciplinar ao invés de explorar algoritmos ou metodologias mais robustas relacionadas a técnicas de diferenciação/comparação de códigos fonte.

Um dos principais desafios enfrentados pelas abordagens atuais adotadas por analisadores anti-plágio é que não há consenso sobre como diferenciar os três tipos de escopo com as quais elas têm que lidar: detecção de plágio no caso de códigos modificados criados por usuários reais; no caso de códigos fonte gerados por uma LLM (BIDERMAN; RAFF, 2022); e no caso de códigos produzido por técnicas automatizadas de sofisticadas transformações sintáticas (ZHANG et al., 2021).

Além de técnicas de evasão de detectores de plágio tais como as discutidas até este momento, outra classe de métodos se baseia na produção de códigos binários cuja diferença em relação a um dado arquivo binário original é minimizada via métodos de ofuscação aplicados no nível de códigos binários pós-compilação (REN et al., 2021). Nesse caso, um parâmetro comum de ofuscação pré-determinado por compiladores (tal como a flag de otimização *-O3* do compilador GCC) tem como comportamento conhecido a simplificação de alterações diversas efetuadas em nível sintático. Intuitivamente, isso "reverte" alterações feitas no código fonte original para seu estado inicial, efetivamente "desfazendo" modificações no programa sendo plagiado (ZHANG et al., 2021), (REN et al., 2021), (DEVORE-MCDONALD; BERGER, 2020).

O presente trabalho explora um novo método para detecção de plágios baseada no uso de técnicas de otimização durante o processo de compilação. Iremos demonstrar que o método resultante é capaz de identificar casos de plágio envolvendo modificações sintáticas complexas no código fonte original—as quais, conforme discutido na seção de Resultados, atualmente conseguem evadir a maioria dos analisadores anti-plágio existentes. Nossos experimentos confirmam uma correlação entre a funcionalidade efetiva do código binário otimizado e seu código fonte correspondente—independentemente de *como* tal funcionalidade foi implementada na linguagem de alto nível. O método aqui proposto pode ser utilizado de forma efetiva e eficaz como uma ferramenta complementar às existentes para efetuar análises de plágio.

Este trabalho está estruturado da seguinte forma. No Capítulo 2, será feita a apre-

sentação de nossa fundamentação teórica, abordando os conceitos básicos necessários para situar o leitor dentro do escopo da pesquisa aqui desenvolvida. Isso possibilitará posicionar as nossas motivações e hipóteses técnicas frente ao estado da arte e a trabalhos relacionados, apresentados no Capítulo 3. No Capítulo 4, apresentaremos o método desenvolvido neste trabalho (`ROBiT`). Após, no Capítulo 5, discutiremos a metodologia experimental utilizada na condução de nossos estudos de caso para avaliação de desempenho da técnica proposta. Por fim, o Capítulo 6 apresentará e interpretará os resultados empíricos encontrados, e no Capítulo 7, apresentaremos nossas conclusões e trabalhos futuros.

2 FUNDAMENTAÇÃO TEÓRICA

Neste capítulo, iremos apresentar e discutir os conceitos básicos necessários para situar o leitor no contexto de como funcionam as principais técnicas para detecção de plágio atuais. Isso se faz necessário na medida em que vários desses conceitos serão mencionados ao apresentarmos semelhanças e diferenças de nosso método em relação a técnicas existentes, e também durante a construção dos argumentos técnicos que irão sustentar o modo de funcionamento de nosso método proposto. A seguir, iremos apresentar breve discussões de alto nível sobre *(i)* definições técnicas relevantes para este trabalho; *(ii)* o modo como características sintáticas e semânticas interagem no escopo de um código fonte e seus binários; *(ii)* o modo como o processo de otimização de códigos binários—no qual nosso método se baseia—funciona. É importante destacar que o escopo de nosso trabalho foca, em particular, no contexto do uso e fluxo de construção de códigos fonte na linguagem C. Desta forma, os assuntos abordados a seguir têm como viés a análise de códigos estáticos, com compilação anterior ao tempo de execução. Isso implica que não iremos, por exemplo, entrar em maiores detalhes técnicos acerca do funcionamento de tais técnicas no contexto de linguagens interpretadas.

2.1 Algoritmos de Análise Sintática e Semântica

A avaliação de um código fonte para identificação de possíveis plágios é uma tarefa substanciosa e complexa: suas características podem ser categorizadas do ponto de léxico, sintático, e semântico. Por consequência, se fazem necessários métodos que traduzam tais características para formatos alternativos ao código fonte C original, e que possibilitem sua análise de maneira organizada e detalhada para possibilitar comparações futuras durante processos de detecção de plágio. Tais métodos, conhecidos como *algoritmos de representação intermediária*, geram, ao longo do processo de compilação, diversas representações alternativas/intermediárias do código fonte—cada qual em um nível de abstração distinto. Apesar de existirem diferentes tipos de algoritmos de representação intermediária, iremos focar nos quatro tipos mais relevantes e com maior prevalência nos estudos relacionados ao tema de plágio: tokenização, algoritmos para construção de árvore de sintaxe abstrata (AST), algoritmos para construção de grafos de controle de fluxo (CFG), e técnicas para criação de grafos de controle de dependências (PDG). Cada uma dessas técnicas transforma a representação original de alto-nível de um programa (i.e., o

código fonte escrito na linguagem C) para representações alternativas que caracterizem diferentes características do programa, em diversos níveis de abstração. Acesso a tais representações alternativas de um programa—representando-o de forma que enfatize, por exemplo, sua estrutura sintática, são essenciais para o processo de compilação. No contexto deste trabalho, entretanto, elas também podem ser úteis para efetuar comparações mais diretas da estrutura e funcionalidade de um dado programa, de forma que facilite processos de detecção de plágio. A seguir, descrevemos brevemente os quatro tipos mais relevantes de algoritmos de representação intermediária.

- **Tokenização**: Se refere o método que divide o código fonte em tokens (palavras-chave, identificadores, literais). É de essencial importância para efetuar a análise sintática inicial de um determinado código fonte.
- **AST**: Se trata de um método que constrói uma árvore representando a estrutura sintática do código fonte, indicando a correspondente hierarquia gramatical e relações entre os diversos elementos do código. É usado durante a fase de análise sintática a fim de modelar a estrutura do programa.
- **CFG**: Método relativo ao fluxo de execução de um dado programa programa, responsável por representar/caracterizar os caminhos possíveis que a execução pode seguir. É utilizado durante a fase de análise semântica, e é especialmente relevante em processos de otimizações de código e análise de fluxo de controle.
- **PDG**: Técnica que combina informações de dependência de dados e de controle, representando as relações entre diferentes partes do código fonte. É utilizada durante a fase de análise semântica para fins de otimizações avançadas, paralelização, e análise de segurança.

Diversos analisadores anti-plágio de código fonte fazem uso primário de processos de tokenização e análise de AST. A tokenização de um programa permite a comparação básica de estruturas léxicas de diferentes códigos fonte, enquanto a AST proporciona uma análise mais aprofundada da estrutura sintática do código, facilitando assim a identificação de plágios, mesmo no caso de modificações superficiais no código. Ferramentas como a técnica `MOSS` (Measure of Software Similarity (SCHLEIMER; WILKERSON; AIKEN, 2003) e JPlag (PRECHELT; MALPOHL; PHILIPPSEN, 2002) utilizam tais técnicas para detectar similaridades significativas entre códigos, indicando assim potenciais casos de plágio.

2.2 Ferramenta Anti-Plágio `MOSS`

A técnica anti-plágio `MOSS` (Measure of Software Similarity (SCHLEIMER; WILKERSON; AIKEN, 2003)) é uma das mais amplamente utilizadas, atualmente, em diversos ambientes—em particular no meio acadêmico e científico. Essa técnica faz uso de um algoritmo baseado em *fingerprinting* para detectar semelhanças entre conjuntos de códigos fonte. Ele gera *fingerprints* ("impressões digitais") de cada documento/código fonte através da identificação de sequências de tokens significativos, ignorando detalhes triviais ou irrelevantes.

Os *fingerprints* de diferentes documentos são então comparados, na busca de correspondências, após todos documentos serem processados em uma etapa de normalização que retira comentários e espaços em branco. Ao fazer isso, o algoritmo `MOSS` é capaz de eficientemente identificar partes de código que são semelhantes ou idênticas, indicando, dessa forma, potenciais casos de plágio.

2.3 Técnicas de Transformação Sintáticas e Semânticas

Sobre análises para identificação de plágio e similaridade entre códigos fonte, ressaltamos um conjunto de técnicas conhecidas como métodos para *detecção de códigos clonados* (*Code Clone Detection*). Tais técnicas também investigam alterações semânticas e sintáticas nas estruturas do código fonte original para identificar potenciais cópias/plágios através da comparação de trechos específicos do código C. Técnicas desse tipo frequentemente são aplicadas, por exemplo, na identificação de roubo de informações intelectuais resultantes do uso indevido de códigos proprietários. Dado que o escopo de estudo do presente trabalho ter intersecções com tais métodos, iremos utilizar algumas técnicas definidas desse tipo para explicar o funcionamento de técnicas de evasão—mais especificamente, técnicas de manipulação de código fonte para evasão de algoritmos anti-plágio.

No contexto de técnicas baseadas em transformações sintáticas/semânticas, Zhang et al (ZHANG et al., 2021) apresentam um método de graduação de similaridades entre trechos de código fonte, analisados na busca por trechos equivalentes. A técnica proposta por Zhang et al. classifica os tipos de similaridade em 4 níveis:

- **Tipo I**: Os códigos analisados são praticamente idênticos quando ignoramos co-

mentários, indentação, espaços em branco, e layout.

- **Tipo II**: Além de mudanças do Tipo I, há também alterações na nomeação de variáveis, funções, tipos de dados, etc.
- **Tipo III**: Além dos Tipos I e II, podem ocorrer também alterações adicionais na estrutura do programa, tais como alterações, reordenamentos, adições, ou exclusões de declarações feitas no código C.
- **Tipo IV**: Corresponde a um tipo de similaridade baseado em análises no nível semântico. Isso significa que podem ocorrer alterações completas de diversos elementos sintáticos, desde que o código resultante mantenha a equivalência semântica em relação ao código original.

Zhang et al. também propõem uma série de 15 transformações atômicas de código, as quais evadem (em diferentes níveis de abstração) métodos anti-plágio baseados na análise de similaridades sintáticas e semânticas. Tais transformações são apresentadas na Figura 2.1.

Transformation Operator	**Description**	**Original**	**Changed**
Op1-ChRename	Function name and variable name renaming.	int i;	int i1;

Op2-ChFor	The for-loop is transformed into a while-loop.	for(i=0;i<10;i++){
		BodyA
		}
			i=0; while(i<10){
			BodyA
			i++; }

Op3-ChWhile	The while-loop is transformed into a for-loop.	while(i<10){ BodyA }	for(;i<10;){ BodyA}
Op4-ChDo	The do-loop is transformed into a while-loop.	do{ BodyA }while(i<10);	BodyA while(i<10){ BodyA}
Op5-ChIfElseIF	Transformation of if elseif to if else.	if(grad<60) BodyA else if(grad<80) BodyB else BodyC	if(grad<60) BodyA else{ if(grad<80) BodyB else BodyC }
Op6-ChIf	Transformation of if else to if elseif.	if(grad<60) BodyA else{ if(grad<80) BodyB else BodyC }	if(grad<60) BodyA else if(grad<80) BodyB else BodyC
Op7-ChSwitch	Transformation of the Switch statement to the if elseif statement.	switch(a){ case 60: BodyA case 70: BodyB default: BodyC }	if(a==60) BodyA else if(a==70) BodyB else BodyC
Op8-ChRelation	Transformation of relational expressions.	a<b	b>a
Op9-ChUnary	Modifications to unary operations.	i++;	i=i+1;
Op10-ChIncrement	Modifications to incremental operations.	i+=1;	i=i+1;
Op11-ChConstant	Modifying constants.	8	(a-b) //8=a-b
Op12-ChDefine	Modifications to variable definitions.	int b=0;	int b;b=0;
Op13-ChAddJunk	Adding junk codes.	if(a){ BodyA}	if(a) { BodyA if(0) return 0; }
Op14-ChExchange	Change the order of the statements in a block without data and control dependency.	a=b+10; c=d+10;	c=d+10; a=b+10;
Op15-ChDeleteComments	Deleting statements that print debugging hints and comment.	printf("test"); //comments	~~//printf("test");~~ ~~//comments~~

Tabela 2.1 – Descrição e exemplos dos 15 operadores de transformações atômicas propostos por Zhang et al. (2021).
Fonte: (ZHANG et al., 2021).

Em nosso trabalho, utilizaremos um subconjunto de 4 transformações (dentre as 15 propostas por Zhang et al.; vide a Tabela 2.1) como base para a estratégia de criação dos arquivos plagiados utilizados durante a análise de desempenho da técnica que iremos propor—mais detalhes podem ser encontrados no capítulo de Metodologia Experimental. As 4 operações/transformações em questão foram escolhidas por nós devido ao seu potencial de evasão semântica, detalhadas a seguir:

- **Op4-ChDo**: Corresponde à operação que transforma loops do tipo *do-while* em loops do tipo *while*.
- **Op7-ChSwitch**: Operação que mapeia uma cláusula do tipo *Switch* para uma seqüência de cláusulas *if/else if/else* encadeadas.
- **Op12-ChDefine**: Corresponde à manipulação de definições de variáveis no código fonte.
- **Op13-ChAddJunk**: Operação responsável pela adição de *deadcode*—códigos que não alteram o funcionamento do fluxo principal do programa, e que nunca são executados diretamente.

Modificar códigos fonte utilizando estas operações gera tanto um impacto sintático (por alterarmos, e.g., uma função aumentando as linhas do código fonte ou alterando a definição e inicialização de variáveis), a modificação também causa impactos nas estruturas de representação semântica intermediárias—especificamente CFG e PDGs. Segundo os resultados apresentados por Zhang et al. (ZHANG et al., 2021), essas técnicas para modificação de códigos fonte frequentemente conseguem evadir diversos tipos de analisadores sintáticos e semânticos, devido à sua capacidade de fundamentalmente alterar a forma que os fluxos de execução são definidos.

2.4 Técnicas de Ofuscação

No contexto da área de análise de *malwares*, existe uma subárea associada à pesquisa de técnicas para evasão de antivírus e sistemas anti-plágio baseados na análise de similaridade entre códigos binários: a ofuscação (POORNIMA; MAHALAKSHMI, 2023).

De forma geral, essa área se baseia em métodos de manipulação do código binário com a finalidade de gerar versões melhoradas do binário e que sejam o mais distante possível da assinatura do binário original; este último ponto, em particular, tem relações diretas com o tipo de abordagem que investigaremos no presente trabalho.

2.4.1 Técnicas de Otimização de Binários

Dentre uma das técnicas de ofuscação empregadas para evasão, inclui-se a técnica de otimização de binários. Esta técnica emprega o estudo de otimização de código durante o processo de compilação, de maneira que minimize a similaridade da assinatura do

código binário pós-ofuscação e o binário do código fonte original.

Esse processo de otimização é regulado através do uso de *flags* de otimização do compilador, as quais representam diversos níveis pré-determinados de otimização (cada qual associado a um conjunto de transformações efetuadas durante o processo de compilação), além de *flags* individuais. no caso de otimizações mais sofisticadas.

2.4.2 GCC — *Flags* de Otimização

O compilador que utilizaremos nesse trabalho é o GCC (Free Software Foundation, 2024). Esse compilador suporta o uso de *flags* individuais, e também disponibiliza três níveis de otimização—cada qual associado a respectivas *flags*: *-O0*, *-O1*, *-O2*, e *-O3*.

Com a exceção de *-O0*, todos demais níveis de otimização do GCC estão associados a um conjunto específico de *flags* que se mantêm ativas a cada nível. Ou seja, ao se utilizar o nível de otimização *-O3*, todas as *flags* presentes em *-O1* e *-O2* se mantêm ativas—além, claro, *das* flags específicas ao nível *-O3*. O nível *-O0* corresponde o valor default de otimização, que diminui o tempo de compilação.

2.4.3 Técnicas de Comparação de Binários

No contexto de técnicas para análise e manipulação dos binários, enfatizamos aqui um conjunto de métodos, em particular, que têm objetivos semelhantes àqueles estudados neste trabalho; em particular, métodos de comparação e pesquisa de similaridade entre códigos binários pós-compilação. Um objetivo central de tais métodos é traçar correlações entre dois códigos analisados, usualmente via técnicas de processamento textual tal como o método da distância de Levenshtein (LEVENSHTEIN, 1965).

Esse método é capaz de determinar o menor número de ações ou operações atômicas (e.g., inserção de um caractere; remoção de um caractere; substituição de um caractere por outro) que sejam capazes de transformar uma string em outra. O método que quantifica essa distância também é conhecido como *edit distance*. Quanto menor a *edit distance* entre duas strings (ou sequencia de bytes em um código binário), mais semelhantes eles são. Grande parte das aplicações de comparação de binários (i.e., *binary diffing*) empregam alguma variante deste tipo de algoritmo.

2.4.4 Ferramenta de Comparação de Similaridade de Binários — Radiff2

Ferramentas de diferenciação e comparação binária, popularmente chamadas de *bindiff*, são métodos que tem como propósito quantificar similaridades e diferenças entre dois códigos binários. A maior parte das ferramentas deste tipo são plugins de outras ferramentas, mais robustas, conhecidas como *disassemblers*, as quais têm a capacidade de fazer engenharia reversa em códigos, análise de baixo nível acerca, por exemplo, do escopo de endereçamento, dentre outras. Quase todas as ferramentas disponíveis no mercado são proprietárias ou apresentam limitações de uso em licenças abertas.

Dadas restrições e levando-se em conta o objetivo do presente trabalho, a ferramente *Radiff2* se mostra valiosa. *Radffit2* é uma ferramenta open source de fácil uso e que disponibiliza o mesmo conjunto básico de funções que outras ferramentas de análise de similaridade. Essa será a ferramenta base para comparação de binários (em particular, via a quantificação da distância de Levenshtein) em nosso trabalho; para mais detalhes, vide as seções 5.3 e 6.5.

3 TRABALHOS RELACIONADOS

Este capítulo apresenta e discute nosso conjunto de artigos relacionados. A seguir, faremos uma breve revisão de artigos pertinentes ao nosso trabalho, divididos nos seguintes grupos de assuntos: técnicas de análise de similaridade em código fonte aplicadas à detecção de plágio; técnicas de análise de similaridade em binários aplicadas à detecção de plágio; e técnicas de evasão relacionadas ao nosso objetivo de pesquisa.

Uma das primeiras transformações sintáticas aplicadas por estudantes na esperança de burlar análises anti-plágio consiste na troca da ordem de trechos independentes do código fonte. Há uma série de artigos (WISE, 1996; PRECHELT; MALPOHL; PHILIPPSEN, 2002; FAIDHI; ROBINSON, 1987) que apresentam algoritmos de análise sintática de similaridade que surge como resposta a esse tipo de técnica simples de evasão. De forma geral, eles são baseados em métodos de *greedy string tiling*, as quais quantificam a similaridade entre strings e são robustas, por exemplo, à transposição de trechos de código. Além disso, também são frequentemente baseados na análise de representações intermediárias do código fonte, na forma de *tokens* e hashing. Esses métodos são eficazes nos casos de plágio mais comuns. Dentre as técnicas que implementam esse tipo de abordagem anti-plágio, enfatizamos a ferramenta `MOSS` (SCHLEIMER; WILKERSON; AIKEN, 2003), amplamente utilizada no meio acadêmico e científico, e, por esse motivo, escolhida como ponto inicial de comparação em nossos experimentos (vide Seção 6.3). Além do `MOSS`, outra técnica de análise sintática para detecção de similaridade foi proposta por Nichols et al. (2019). Esta técnica baseia-se em comparações através do algoritmo Smith-Waterman para alinhamento de sequencias, com base na análise de representações intermediárias do programa—em particular, suas respectivas árvores de sintaxe abstrata (ASTs). Ao contrário da nossa abordagem, entretanto, que irá ser construída no contexto de comparações de códigos binários otimizados, este algoritmo identifica possíveis plágios analisando representações de mais alto nível derivadas dos códigos fonte em questão.

Foram propostos diversos métodos adversários para evadir o tipo de análise sintática utilizado pela maioria dos sistemas anti-plágio da época. Em particular, enfatizamos aqui o algoritmo Mossad (DEVORE-MCDONALD; BERGER, 2020). Mossad—que é potencialmente o trabalho relacionado mais relevante no contexto da nossa pesquisa—consiste em um framework para transformação de programas com o objetivo de evadir sistemas de avaliação anti-plágio automatizados baseados em tokenização e ASTs. Esse

algoritmo é baseado em transformações sintáticas relacionadas primariamente à inclusão de código morto ao longo do código fonte original. Tal abordagem faz com que tanto o resultado do processo de tokenização quanto do processo de construção de ASTs sejam afetados, e, portanto, que análises sintáticas não consigam reconhecer o plágio. No artigo em que o Mossad foi proposto Devore-McDonald and Berger (2020), os autores conduziram análises experimentais de desempenho envolvendo uma técnica para detecção de plágio baseada não na comparação de códigos binários, e sim de arquivos-texto contendo o código assembly associado ao programa sendo analisado (quando compilado com flags de otimização). Isso diferente de nossa abordagem na medida em que nós investigaremos a eficácia de comparações de arquivos binários (executáveis) produzidos ao final do processo de compilação completo; em outras palavras, após o arquivo C ser traduzido para um arquivo-texto contendo o código assembly correspondente; após o arquivo assembly ser mapeado para um arquivo objeto .o; e após todos arquivos .o relevantes serem processados em um processo de linkagem, para produção do arquivo binário executável propriamente dito.

Há também técnicas que se baseiam em representações de grafos, as quais são mais robustos do que os métodos baseados na análise de ASTs ou em tokenização. Liu et al. (2006), por exemplo, propuseram algoritmos que utilizam o grafo de dependências de um programa para determinar potenciais casos de plágio. No contexto de detecção de plágio via grafos de controle de fluxo, temos o artigo de Chae et al. (2013), o qual utiliza a valoração no grafo de um programa no momento da comparação, e posteriormente determina um valor número quantificando o grau de similaridade. Ambas as técnicas podem ser evadidas quando aplicamos transformações semânticas no código, de forma que a relação dos nodos desta representação intermediária seja alterada.

Algumas das técnicas mencionadas estão diretamente relacionadas à abordagem do artigo de Zhang et al. (2021), que discute de maneira detalhada as diferentes estratégias para construção de algoritmos de detecção de plágio, assim como os tipos de transformações sintáticas e semânticas que podem ser empregadas para evadir cada possível técnica e suas eficácias. Em especial, os capítulos 2, 4 e 7.2 do artigo de Zhang et al. são fundamentais para a compreensão de como os analisadores de similaridade entre códigos funcionam, suas características, e suas limitações.

Por fim, nossa revisão de literatura revelou também a existência de uma extensa coleção de trabalhos relacionados comparando técnicas de análise de similaridade entre códigos binários e as possíveis abordagens de ofuscação/evasão (LUO et al., 2017; DA-

MÁSIO et al., 2023; POORNIMA; MAHALAKSHMI, 2023). Dentre esses trabalhos, o artigo que melhor descreve as correlações entre técnicas de evasão e técnicas correspondentes de quantificação de similaridade é o de trabalho de Ren et al. (2021). Nesse artigo, os autores constroem um framework que utiliza flags de otimização durante o processo de compilação com objetivo de construir arquivos binários que garantidamente sejam substancialmente diferentes do arquivos binários iniciais advindos diretamente do processo de compilação. Os autores também discutem o efeito de diversas transformações sintáticas e semânticas quando aplicadas ao código fonte, correlacionando os tipos de alteração causadas por vários processos de otimização e seu impacto respectivo nas estruturas de representação intermediária do programa. Além disso, o trabalho de Ren et al. (2021) compara e discute os efeitos da aplicação de diferentes níveis de otimização nos códigos binários que são produzidos durante a compilação, demonstrando de maneira clara o comportamento e impacto do uso da flag *-O3*, comparado com o uso da flag *-O0*, por exemplo. O principal limitação desta abordagem—no contexto do objetivo abordado em nossa pesquisa—é o ponto de vista adotado pelos autores. Em nosso trabalho, temos como objetivo apresentar uma técnica que efetivamente "remova" do código binário (via otimizações durante a compilação) quaisquer efeitos que modificações sintáticas e semânticas feitas no código fonte poderiam causar, a fim de facilitar a detecção de plágios.

O trabalho de Ren et al. (2021) também explora o uso de técnicas de otimização, mas não para facilitar a detecção de potenciais plágios - ele o utiliza como método de ofuscação desenvolvido especificamente para distanciar o código binário final, ofuscado, ainda mais de seu estado inicial. Outra diferença chave é que o trabalho de Ren et al. (2021) assume que os códigos fonte originais não estão disponíveis, e portanto opera diretamente apenas sobre arquivos binários. Em nosso trabalho, por outro lado, no qual focamos na detecção de plágio em ambientes acadêmicos e científicos, é natural que os códigos C sendo analisados (produzidos por alunos, desenvolvedores, cientistas, etc.) estejam disponíveis e possam ser incorporados ao *pipeline* completo de comparação e análise para detecção de plágio.

Recapitulando, nesta seção apresentamos o corpo de artigos que serviram de base para esta pesquisa e uma análise crítica de aplicação e sustentação para pontos fundamentais deste trabalho. Comparativamente aos temas de similaridade entre códigos fonte utilizando os algoritmos de análise sintática e semântica, nosso método se diferencia utilizando a análise de binários, se posicionando em um escopo não explorado pelos típicos analisadores anti-plágio presentes no mercado. Quando vemos pelo contexto binário em

si, nosso método também se diferencia por não estar na mesma categoria de testes adversários sem a presença do código fonte original - a grande maioria dos trabalhos parte do pressuposto de um binário completamente independente. Nós além de utilizarmos o código fonte, também utilizamos o oposto da ideia normalmente sustentada para o uso de otimizações de binários, já que ao invés de utilizarmos esta função como método de ofuscação, em nossa proposição ela é utilizada como reversão de ofuscação de nível sintático.

4 SOLUÇÃO PROPOSTA

Neste capítulo, iremos descrever o método proposto neste trabalho: o algoritmo `ROBiT`. Todas as informações de embasamento teórico apresentadas até agora serão exploradas para possibilitar o desenvolvimento de nosso modelo, o qual utiliza otimizações durante o processo de compilação e, dessa forma, pode servir como um método complementar às técnicas existentes de estado-da-arte para detecção de plágios. Antes de apresentarmos o método, iremos justificar as escolhas de projeto feitas acerca de como especificar seus mais relevantes componentes, de forma que elas sejam alinhadas com as observações feitas durante nosso processo de revisão de trabalhos existentes. Além disso, iremos apresentar algumas das ferramentas e frameworks/softwares que iremos explorar para construir o osso método.

4.1 Escolha das Transformações Sintáticas e Semânticas

Como discutido na Seção 3.3 e na seção de Trabalhos Relacionados, estamos, nesse projeto, utilizando o trabalho desenvolvido por Zhang et al. (2021) como referência para escolhermos as transformações que serão utilizadas e aplicadas aos códigos fonte base, com objetivo de criar arquivos de plágio a serem utilizados durante a experimentação e avaliação empírica de desempenho de nosso método. Tais transformações foram escolhidas devido ao seu potencial de evasão semântica e são essenciais para garantirmos que o método proposto nesse capítulo possa ser, posteriormente, avaliado de forma apropriada durante a condução de nossos experimentos. Conforme mencionado anteriormente, Zhang et al. (2021) apresentam, em seu artigo, uma excelente discussão demonstrando os efeitos que diversas transformações atômicas, quando aplicadas a um dado código fonte, têm em termos de seu impacto sintático e também em termos das modificações que elas induzem nas estruturas de representação semântica intermediárias.

No que diz respeito à discussão apresentada no capítulo anterior acerca das transformações por nós escolhidas (**ChDo**, **ChSwitch**, **ChDefine**, **ChAddJunk)**, observamos que estas são as únicas transformações—dentre as 15 originalmente estudadas por Zhang et al.—que afetam todos os quatro tipos de algoritmos de representação parcial/intermediária de códigos. O motivo pelos quais eles afetam programas de forma tão substancial (e que faz com que consigam efetivamente evadir as principais estratégias anti-plágio) é que elas alteram significativamente os grafos de representação devido à sua capacidade de funda-

mentalmente alterar o fluxo de um dado programa; e esse é exatamente o nosso objetivo ao selecionarmos as transformações mais relevantes no contexto de nosso trabalho.

4.2 Considerações Sobre a Flag *-O3*

A flag de compilação GCC *-O3* foi escolhida como técnica de otimização base, no contexto deste projeto, devido às observações resultantes dos experimentos conduzidos por Ren et al. (2021). Naquele artigo, os autores mostram como a diferença entre o efeito e impacto do uso de flags *-O0* e *-O3* é substancial; e que isso é causado primariamente devido ao fato que a função *-O3* simultaneamente otimiza *vários* componentes na estrutura de representação do código. Há um contraponto a ser feito acerca do efeito do uso de determinadas flags de otimização em termos que quanto o binário correspondente resultante pode diferir do binário "original" (i.e., o obtido através de compilação sem otimização).

Algumas transformações escolhidas, como por exemplo **ChAddJunk** e **ChDefine**, tendem a produzir binários com estrutura semelhante à estrutura do binário original, i.e., do binário resultante de compilação sem otimizações. Por esse motivo, o método proposto neste capítulo não analisará apenas o código fonte, ou apenas o binário simples (não otimizado). O objetivo será comparar binários resultantes de processos de compilação com otimização, devido à observação de que otimizações em códigos binários podem efetivamente servir como um "filtro reverso", removendo da funcionalidade do código binário possíveis fontes de ofuscação implementadas pelo plagiador em suas modificações de um código fonte original.

4.3 Radiff2

A ferramente Radiff2 será a base que possibilitará a comparação de arquivos binários, por parte de nosso método. O resultado por ela produzido poderá auxiliar na determinação sobre se um binário é ou não um plágio. A função *-ss* dessa ferramenta faz com que ela utilize um algoritmo de comparação de binários que determina o menor número de ações ou operações atômicas (e.g., inserção de um caractere; remoção de um caractere; substituição de um caractere por outro) que sejam capazes de transformar um código binário em outro. Esse método para quantificação da distância (ou similaridade) entre arquivos binários também é conhecido como *edit distance*. Quanto menor a *edit*

distance entre duas sequencia de bytes em códigos binários, mais semelhantes eles são.

Com base na *edit distance* (ou distância de Levenshtein (LEVENSHTEIN, 1965)), é possível determinar um percentual de similaridade entre os dois códigos. Decidimos pelo uso exclusivo da função *-ss* pois, durante nossos testes preliminares, entendemos que comparações em outros níveis de abstração—por exemplo, através da comparação direta do código assembly correspondente a cada programa—já haviam sido exploradas na literatura ou fugiriam do escopo geral deste trabalho.

4.4 `ROBiT`: Um Método de Análise Anti-Plágio

`ROBiT` é o método anti-plágio que iremos apresentar neste trabalho. Ele visa identificar casos de similaridade de plágio ao analisar conjuntos de programas C, efetuando a comparação de seus respectivos códigos binários, quando produzidos através de processos de compilação GCC com otimizações. Dado um conjunto de códigos fonte submetidos à análise, nosso método irá, primeiramente, compilar os códigos C fornecidos utilizando a flag de otimização *-O3*. Após, irá quantificar a distância mínima de Levenshtein entre todos pares de binários correspondentes via a técnica Radiff2. Uma vez feitas todas as comparações de possíveis pares de programas, os correspondentes resultados serão armazenado em um arquivo texto para posterior consulta e análise.

Operator	AST	CFG	PDG	Token
Op1-ChRename	◓	○	◓	○
Op2-ChFor	●	○	○	●
Op3-ChWhile	◓	○	○	●
Op4-ChDo	●	●	●	●
Op5-ChIfElseIF	◓	○	○	●
Op6-ChIf	◓	○	○	●
Op7-ChSwitch	●	●	●	●
Op8-ChRelation	●	○	◓	●
Op9-ChUnary	●	○	◓	●
Op10-ChIncrement	●	○	◓	●
Op11-ChConstant	●	○	◓	●
Op12-ChDefine	●	●	●	●
Op13-ChAddJunk	●	●	●	●
Op14-ChExchange	○	●	○	◓
Op15-ChDeleteC	●	◓	◓	●

*Note that we use the symbol "●"to denote severe effects, "◓"to denote only minor effects, and "○"to denote no effects.

Tabela 4.1 – Operadores de transformação e seu impacto nas quatro transformações de código frequentemente utilizadas.
Fonte: (ZHANG et al., 2021).

Mais especificamente, os resultados do processo descrito anteriormente irão gerar dois números para cada par de arquivos: *(i)* o primeiro corresponde ao grau de similaridade entre os dois códigos em questão, em termos percentuais calculados de forma inversamente proporcional à distância de Levenshtein: quando menor a distância, maior o percentual de similaridade; *(ii)* o segundo corresponde à distância de Levenstein propriamente dita. Esses dados estão correlacionadas na medida em que quanto maior a distância de Levenstein (ou *edit distance*) entre dois códigos binários, menor será a probabilidade de ter ocorrido um plágio.

É importante notar o uso de otimizações GCC *-O3* efetivamente resulta em um tipo de "normalização semântica" em ambos os códigos binários. Se dois programas C implementam uma mesma funcionalidade, mas utilizando diferentes comandos e tipos de instruções C, o processo de otimização tende a mapear os correspondentes códigos fonte para binários semelhantes. Intuitivamente, as representações de baixo nível, induzidas pelo processo de otimização avançado, tendem a ser semelhantes na medida em que a otimização preserva a *funcionalidade* implementada por cada código C, independentemente de *como* tal funcionalidade foi implementada na linguagem de alto nível. Isso possibilidade que as comparações dos binários resultantes identifique casos de plágio de forma mais consistente na média em que é capaz de identificar similaridades funcionais mesmo quando os códigos fontes correspondentes tenham sofrido alterações sintáticas complexas.

5 METODOLOGIA EXPERIMENTAL

Neste capítulo discutiremos a metodologia experimental utilizada na condução dos experimentos que acompanham e corroboram nossos argumentos de pesquisa. Os experimentos propostos neste capítulo consistem na comparação do desempenho do nosso método (apresentado no capítulo anterior) em relação a técnicas alternativas—incluindo uma variante baseada na comparação direta de binários, sem otimização, e do `MOSS` (um dos sistemas anti-plágio mais amplamente utilizados atualmente). O objetivo dos experimentos discutidos a seguir é investigar e demonstrar diversas características do nosso método, tais como:

1. Demonstrar empiricamente que algoritmos de análise sintática, amplamente utilizados para detecção de plágio, são suscetíveis a estratégias de alteração de código baseadas em transformações sintáticas e semânticas tais como **Op4-ChDo**, **Op7-ChSwitch**, **Op12-ChDefine**, e **Op13-ChAddJunk**, descritas anteriormente.
2. Analisar a eficácia do `ROBiT` quando comparado com o analisador sintático `MOSS`, apresentando evidências experimentais de que ele, de forma geral, apresenta desempenho superior na análise e reconhecimento de plágios.
3. Demonstrar que a aplicação de técnicas de otimização binária através de flags do compilador, o `ROBiT` consegue efetivamente "reverter" as alterações semânticas e sintáticas feitas por transformações visando a construção de plágios, o que resulta em uma taxa de detecção superior, em vários casos desafiadores, quando comparada às taxas identificadas por métodos baseados na comparação de binários não otimizados, e também em relação a métodos como o `MOSS`.

A seguir detalharemos o passo a passo das escolhas das estruturas utilizadas para a prova de conceito, além de aprofundarmos o método de utilização de ferramentas já citadas no capítulo 2, durante nossos embasamentos teóricos.

5.1 Escolha dos Problemas-Base

Para os experimentos, focamos em problemas próximos da realidade dos estudantes, tipicamente relacionados à produção de código na disciplina de Algoritmos e Programação, ligada aos cursos de Ciência e Engenharia de Computação, na Universidade Federal do Rio Grande do Sul (UFRGS). Para conduzirmos a análise experimental

de nosso método, utilizaremos dois enunciados de problemas de programação disponíveis nos últimos semestres, os quais foram fornecidos por alunos do curso de Ciência da Computação:

- **Programa 1**: *"Crie um programa que leia a massa inicial de um material (em gramas) e calcule quanto tempo leva para que essa massa se reduza a menos de 0.5 gramas, assumindo que a massa é reduzida pela metade a cada 50 segundos. O programa deve verificar se a massa inicial é válida (maior ou igual a 0.5 gramas) e imprimir a massa final, juntamente com o tempo total decorrido em horas, minutos e segundos"*.
- **Programa 2**: "*Escreva um programa em C que calcule o valor aproximado em radianos e graus do arctan(x), onde o usuário insere o valor de x. O programa deve verificar se o valor de x está dentro do intervalo válido (neste caso -1 < x < 1). Se estiver dentro do intervalo válido, o usuário deve inserir o número de termos para a soma. O programa então deve calcular o arctan(x) utilizando a série de Taylor até o número de termos especificado e imprimir o resultado em radianos e graus*".

Tipicamente, exercícios em disciplinas de ensino de programação envolvem enunciados com problemas simples e com escopo bem definido, o que implica uma baixa variabilidade na forma como podem ser resolvidos—principalmente no que diz respeito a questões como escolha de nomes de variáveis, cálculos necessários para resolução do problema, definição de limites, etc. Essas limitações são relevantes no contexto deste projeto pois permitem isolar os resultados de transformações sintáticas e semânticas aplicadas a um código fonte base para produção de plágios, dessa forma facilitando a investigação sobre seu impacto no código fonte resultante.

5.2 Geração de Códigos Utilizados na Análise Experimental

Para a geração dos códigos fonte utilizados nos experimentos a serem conduzidos, empregamos o ChatGPT ((YENDURI et al., 2023)), motivados pela grande adoção dessa ferramenta no meio acadêmico, tanto com objetivo de auxiliar alunos na resolução de tarefas de programação quanto no contexto de produção plágios. O ChatGPT é capaz de fornecer respostas únicas e diferentes a cada pergunta/prompt feita pelo usuário.

É importante, durante a geração de programas base com o ChatGPT, que possamos

informar quais características o correspondente código fonte que resolve um determinado problema deve possuir—por exemplo, que ele utiliza cláusulas *switch-cases* e *do-whiles*, visto que duas transformações sintáticas posteriores, para geração de códigos de plágio, podem ser dependentes da presença destas condições.

```
ChatGPT, como você criaria um programa na linguagem C baseado no
enunciado abaixo, utilizando switch-cases e também cláusulas do tipo
do-while?
                          <Enunciado do problema>
```

Em nossos experimentos, este enunciado criou os códigos utilizados como base para cada um dos problemas.

5.2.1 Prompts Utilizados para Criação de Casos de Plágio

O próximo passo para condução dos experimentos consiste na criação de códigos de plágio baseados no código base, através da aplicação de diversas técnicas de transformação. Em particular, para cada código base correspondendo a um dos problemas de programação, queremos obter uma versão nova resultante da aplicação de cada uma das técnicas de transformação, gerando 4 novas versões de códigos a partir do código original. Para motivos de comparação durante os experimentos, também desejamos gerar uma quinta versão modificada do código original, com todas as 4 transformações aplicadas simultaneamente. O prompt ChatGPT utilizado para produção desses códigos fonte é o seguinte:

Considere o código C fornecido a seguir: ***<Código base do problema>***.

Aplique as seguintes transformações, gerando uma versão diferente do código para cada transformação a seguir:

(1) Loops *do-while* podem ser transformados em loops *while*.
(2) Mudança das definições das variáveis no código para serem diferentes do que o original. Por exemplo,

```
int b = 0;
```

poderia ser transformado em

```
int b; b = 0;
```

Além disso, decomponha valores numéricos de variáveis em composições de soma que não alterem a corretude da solução, nem a semântica do programa.
(3) Adicione *junk codes* espalhados pelo código, e que não alterem semanticamente o programa.
(4) Altere a cláusula *switch-case* para *if-else-if* encadeados.

Ao final, crie uma quinta versão do código, aplicando simultaneamente todas as transformações acima.

Este prompt obteve sucesso na criação dos 5 códigos que iremos considerar como diferentes versões de plágio para cada código fonte original, totalizando assim 6 códigos por experimento, com 2 experimentos planejados. A nomenclatura dos arquivos produzidos por esse processo seguirá o seguinte padrão: **p<numero do enunciado>-<transformação aplicada>**. Por exemplo, um arquivo nomeado **p2-if** diz respeito ao programa base 2, transformado através de uma operação que modifica cláusulas de *switch-case* em comandos *if-else* encadeados. De forma semelhante, um arquivo nomeado **p1-all** refere-se ao programa 1, com todas as transformações aplicadas.

5.3 Análise do MOSS

Tendo obtido todos os códigos originais e plágios necessários para condução dos experimentos, o próximo passo consiste em analisar tais programas através do sistema anti-plágio MOSS, a fim de encontrar similaridade entre os códigos. Seguindo as instruções de utilização do MOSS (AIKEN, 2004), recebemos via e-mail um script Perl com um ID único para uso da plataforma.

Este script pode ser então executado utilizando-se como argumento a linguagem de programação utilizada e os nomes de todos arquivos que devem ser comparados. O script então retorna como resposta um endereço web no qual os resultados comparativos acerca de suspeitas de plágio podem ser analisados, informando a probabilidade de plágio para cada par de arquivos comparados, assim como uma comparação lado a lado dos códigos fontes correspondentes, com áreas problemáticas destacadas utilizando diferentes cores. O script mencionado pode ser encontrado no site do projeto do `MOSS`.

5.4 Compilação e Otimização dos Códigos Binários

O próximo passo na condução dos experimentos é a criação das duas versões de binário que serão analisadas e comparadas pelos métodos `ROBiT` e `ROBiT-Simple`. Um dos binários é gerado através de compilação clássica via GCC, e outra versão, otimizada, é gerada através da inclusão da flag de compilação *-O3*.

Para a criação dos arquivos binários nós simplesmente utilizamos a versão original do script do `ROBiT` (Figura B.1), o qual recebe como entrada os arquivos de código fonte de interesse e cria os binários correspondentes, já otimizados. Para o propósito de nossos experimentos criamos também um segundo script, variante do `ROBiT`, com a única alteração—a retirada da flag de otimização. Esse script é o responsável pela execução do método `ROBiT-Simple`.

5.5 Comparação entre Binários

Por fim, chegamos no último passo necessário para execução dos experimentos, que é a comparação entre os binários do conjunto gerado anteriormente. Como resultado da execução do código principal do método `ROBiT`, obtivemos todos os arquivos compilados e também o resultado das comparações de todos códigos binários correspondentes. O resultado de tais comparações (efetuadas através da ferramente Radiff2) é armazenado em um arquivo texto.

Recapitulando, neste capítulo apresentamos todos os passos necessários para a preparação de dados para a fase de experimentação e avaliação de nosso método. Primeiramente, elencamos os enunciados de problemas que servirão de base para nossos experimentos, utilizando para isso o ChatGPT. Descrevemos também as transformações sintá-

ticas e semânticas que serão utilizadas para criarmos as 6 versões modificadas de código fonte inicial. Por fim, descrevemos o processo pelo qual enviaremos os resultados (códigos fonte originais e plágios) para o `MOSS`. Em seguida, apresentamos a maneira como o nosso método, `ROBiT`, e uma versão levemente alterada do mesmo, o `ROBiT-Simple`, podemos ser utilizados para análise dos códigos fonte originais a fim de efetuarmos dois tipos de análise: uma quantificando o percentual de similaridade entre todos os pares de arquivos binários gerados com otimização (`ROBiT`), e outra quantificando o percentual de similaridade entre todos os pares de arquivos binários gerados sem otimização (`ROBiT-Simple`).[1]

[1]Os códigos descritos neste capítulo podem ser encontrados no github da autora: https://github.com/dujour/ROBiT

6 RESULTADOS

Este capítulo apresenta os resultados obtidos empiricamente após a execução dos experimentos com nosso método, ROBiT. Iremos apresentar, a seguir, análises qualitativas e quantitativas acerca dos resultados empíricos obtidos, a fim de validar a eficácia de nosso método frente a duas técnicas alternativas: o método ROBiT-Simple, que também se baseia na comparação de binários via *edit distance*, mas que não explora métodos de otimização durante o processo de compilação; e o algoritmo MOSS, atualmente um dos analisadores anti-plágio mais amplamente utilizados no meio acadêmico e científico.

Os resultados experimentais a seguir se dividem em três categorias, incluindo avaliações empíricas de desempenho de cada método e correspondentes discussões, interpretações, e análises qualitativas. Em particular, iremos apresentar, a seguir:

1. Comparação de nosso método, ROBiT, com o MOSS (*Seção 7.1*);
2. Comparação de nosso método com a técnica ROBiT-Simple, a qual não explora conceitos de otimização para "uniformizar" a estrutura dos respectivos códigos de baixo nível resultantes da compilação—independentemente de como a funcionalidade de tais programas foi implementada/especificada na linguagem de alto nível (*Seção 7.2*);
3. Interpretação e discussão sobre o desempenho de diferentes métodos em desafiadores casos de detecção de plágio, cada qual correspondendo a um *setting* no qual os códigos fonte originais foram modificados via um amplo leque de possíveis transformações sintáticas e semânticas (*Seção 7.3*).

6.1 ROBiT vs. MOSS

Primeiramente, iremos comparar o método ROBiT com a técnica clássica, e amplamente adotada, MOSS. Para isso, quantificamos o percentual de similaridade identificado por cada técnica ao analisar cinco variantes (tipos de plágio) produzidas com base em dois programas de base, ambos escritos na linguagem C. Quanto maior o percentual de similaridade produzido por uma técnica, mas precisa ela foi em corretamente identificar a ocorrência de plágio.

A seguir, iremos nos referir aos resultados associados a análise de cada um dos dois programas de base mencionados como *Experimento 1* (Figura 7.1) e *Experimento 2*

(Figura 7.2). Em todos gráficos apresentados nessa seção e em seções subsequentes, as probabilidades reportadas correspondem à média das probabilidades/similaridades reportadas por cada método ao comparar em um total de 15 pares de códigos fontes—incluindo o código base e seus respectivos plágios, assim como pares de códigos plagiados via diferentes tipos de modificações sintáticas e semânticas.

Em ambos os experimentos, nota-se que o `ROBiT` tem um desempenho significativamente superior ao do `MOSS`. A diferença entre os valores de percentuais de similaridade identificadas por cada métodos é significativa, como podemos observar nas Figuras 6.1 e 6.2. O motivo principal para tal diferença de desempenho parece ser o efeito das técnicas de transformação escolhidas e utilizadas no momento de criação de códigos plagiados. O impacto do uso destas técnicas na capacidade do `MOSS` de reconhecer os plágios é notável: nos (poucos) casos em que o `MOSS` consegue reconhecer a ocorrência de plágio, a probabilidade por ele reportada—em ambos os experimentos—é extremamente baixa; ou seja, não apenas o `MOSS` falha na detecção de plágios, como também, quando suspeita de um potencial plágio, reporta tal fato com um baixo nível de certeza.

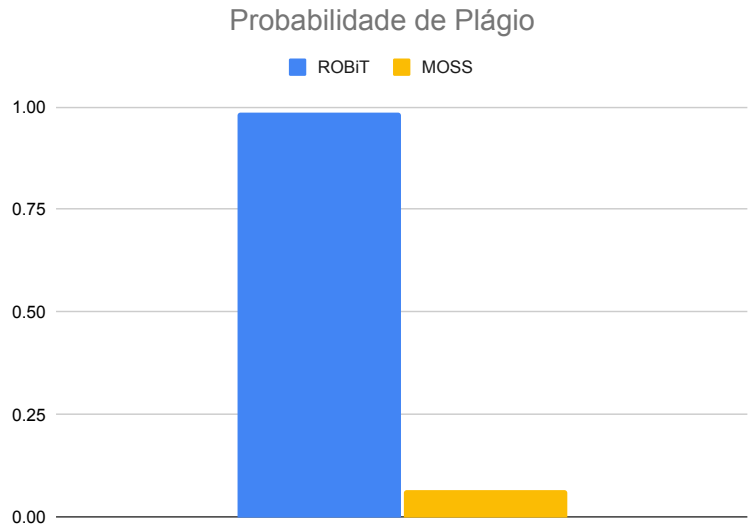

Figura 6.1 – Comparação das probabilidades de plágio identificadas pelas técnicas `ROBiT` e `MOSS` (Experimento 1).
Fonte: A Autora.

É importante reiterar que as técnicas utilizadas para modificar códigos fonte base e gerar arquivos de plágio, em ambos os experimentos, correspondem a técnicas que, embora relativamente simples (e.g., substituição de uma estrutura de laço por outra), afetam de forma significativa diversos métodos anti-plágio existentes. *Por essa razão, as melhorias observadas no nível de detecção plágios identificados por nosso método, quando comparado com o algoritmo `MOSS` (o qual é amplamente utilizado em diversos meios)*

reforça a importância de nossa contribuição para a área.

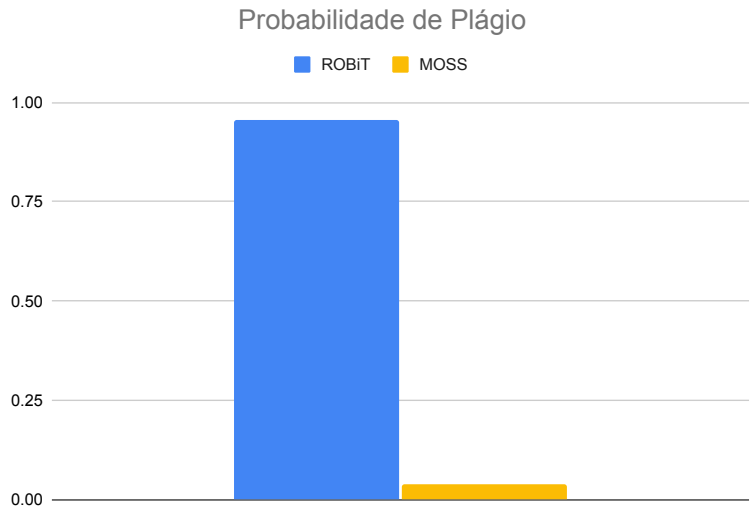

Figura 6.2 – Comparação das probabilidades de plágio identificadas pelas técnicas `ROBiT` e `MOSS` (Experimento 2).
Fonte: A Autora.

6.2 `ROBiT` vs. `ROBiT-Simple`

O segundo resultado empírico que discutimos nesse capítulo corresponde à comparação de pares de binários (em ambos experimentos) para o caso dos algoritmos `ROBiT` e `ROBiT-Simple`. Ambas técnicas são baseadas na comparação de códigos binários: `ROBiT` (nosso método) opera sobre binários gerados através de processos de compilação com otimização, enquanto `ROBiT-Simple` analisa apenas binários sem otimização.

Observamos, nas Figuras 6.3 e 6.3, que em um dos experimentos (Experimento 1) o nosso método—`ROBiT`—obteve uma leve vantagem em relação ao `ROBiT-Simple`. Por outro lado, no Experimento 2, o algoritmo `ROBiT-Simple` apresentou um desempenho levemente superior. Isso se deve à escolha dos códigos fonte base utilizados nos experimentos. Em algumas situações, por exemplo, um dos códigos fonte base não possuía cláusulas do tipo *switch-case*, e portanto nenhuma das transformações sintáticas e semânticas capazes de modificar tais cláusulas em sequências de comandos *if-else* encadeados poderia ser aplicada. Consequentemente, os diferentes códigos fonte base analisados em nossos experimentos, durante a geração de plágios, podem ser impactados de formas distintas por apenas algumas—mas não todas—as possíveis transformações sintáticas e semânticas. Nossa hipótese é que as modificações e transformações em particular

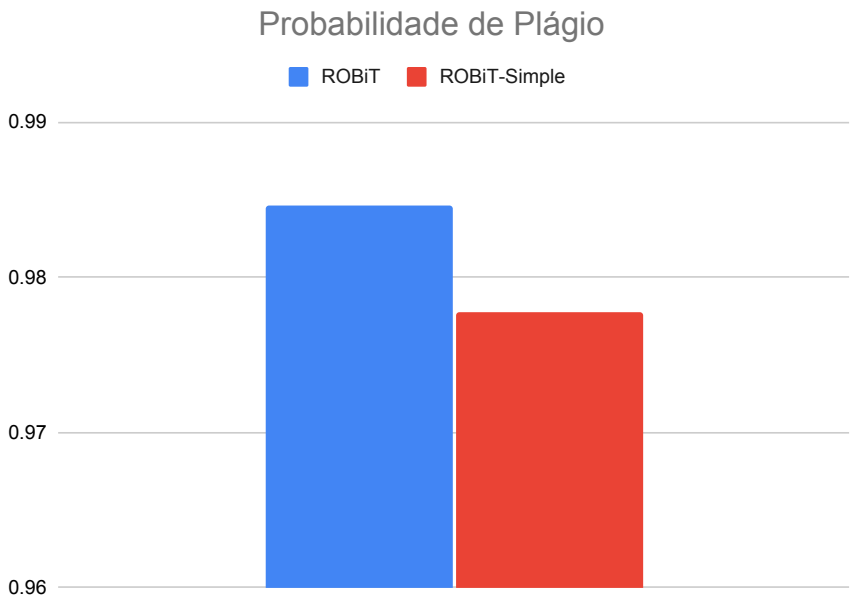

Figura 6.3 – Comparação das probabilidades de plágio identificadas pelas técnicas `ROBiT` e `ROBiT-Simple` (Experimento 1).
Fonte: A Autora.

passíveis de serem aplicadas a cada programa base determinam qual técnica de detecção de plágio (baseada em comparações de arquivos binários) terá desempenho levemente superior—mesmo que tais diferenças de desenpenho não sejam necessariamente estatisticamente significativas.

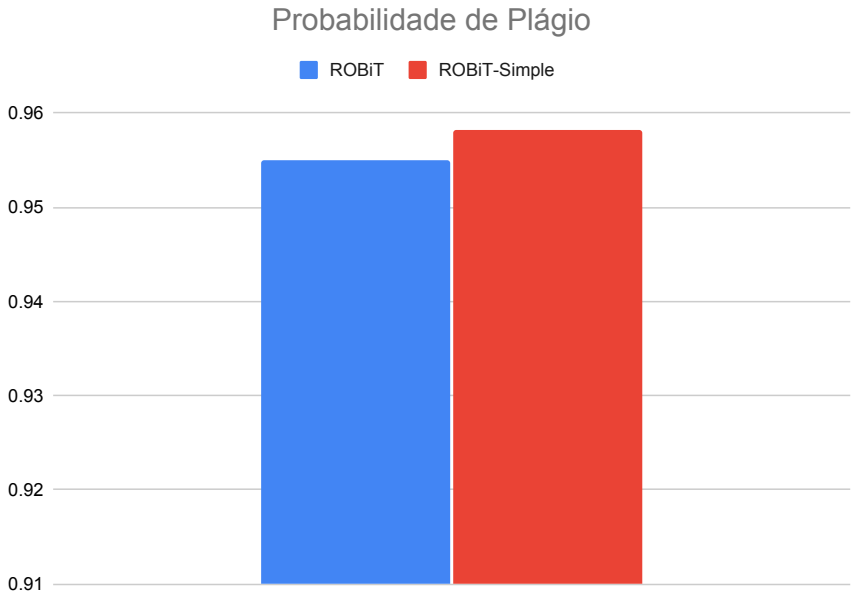

Figura 6.4 – Comparação das probabilidades de plágio identificadas pelas técnicas `ROBiT` e `ROBiT-Simple` (Experimento 2).
Fonte: A Autora.

A fim de expandir o escopo do Experimento 1, o qual já inclui um arquivo de plágio gerado através da aplicação de *todas* possíveis transformações sintáticas e semânticas, adicionamos um segundo arquivo de plágio com essa mesma característica. Esses programas são chamados, respectivamente, de *p1-all* e *p1-all-2*. Embora esses programas incluam os mesmos tipos de construções sintáticas e comandos da linguagem C, seus códigos fonte naturalmente diferem levemente. Ao serem analisados pelo `ROBiT`, entretanto, devido ao uso de otimizações, os códigos binários resultantes tendem a ser *semelhantes*. Por outro lado, essas pequenas diferenças em seus códigos fontes podem gerar arquivos binários substancialmente diferentes caso compilados sem otimizações. Nesse caso, é natural que o método `ROBiT` obtenha desempenho levemente superior ao `ROBiT-Simple`, no que diz respeito a sua capacidade de detectar similaridades. Tal diferença de desempenho entre os dois métodos pode ser observada na primeira linha da Figura 6.5.

Esse tipo de padrão de desempenho, em que o `ROBiT` supera o `ROBiT-Simple` apenas em algumas situações em particular, mas não em todas, pode ser observado mais claramente ao analisarmos os resultados do Experimento 2 (Figura 6.6). Nesse caso, é evidente que o método em particular que apresenta melhor desempenho (ao comparar a similaridade entre um dado par de programas) varia levemente. De forma geral, entretanto, existe um equilíbrio de casos nos quais um dos dois métodos é ligeiramente melhor que o outro. É importante observar, entretanto, a escala de percentuais de similaridade relatados na Figura 6.6: essa escala apresentada do eixo x da figura varia de 92% de similaridade até 100% de similaridade. Eventuais diferenças na capacidade de detecção de similaridade de cada método são pequenas, sempre variando em menos de 10 pontos percentuais.

Considerando as observações apresentadas e as possíveis transformações que podem ser aplicadas a códigos fonte, é possível concluir que o método `ROBiT` tende a ser levemente superior ao método `ROBiT-Simple`; e que ambos são significativamente mais precisos que o método de estado-da-arte, `MOSS`.

6.3 Análise do Impacto de Transformações Sintáticas e Semânticas

Ao observarmos a análise de desempenho do `MOSS` em ambos os experimentos, é notável a forma como a aplicação de transformações sintáticas e semânticas nos códigos fonte originais substancialmente afeta sua eficácia de forma negativa. O `MOSS`, de

fato, tem o pior desempenho dentre os três métodos avaliados em nosso trabalho, mas por um bom motivo: o MOSS é baseado na análise de tokens, o que o torna particularmente suscetível a transformações capazes de afetar tal característica de programas. Nossos resultados empíricos corroboram essa observação, na medida em que o MOSS se mostra ineficaz na quantificação de percentuais de similaridade em 13 dos 15 casos de transformações apresentadas no trabalho de Zhang et al. (2021)—vide Figura 4.1.

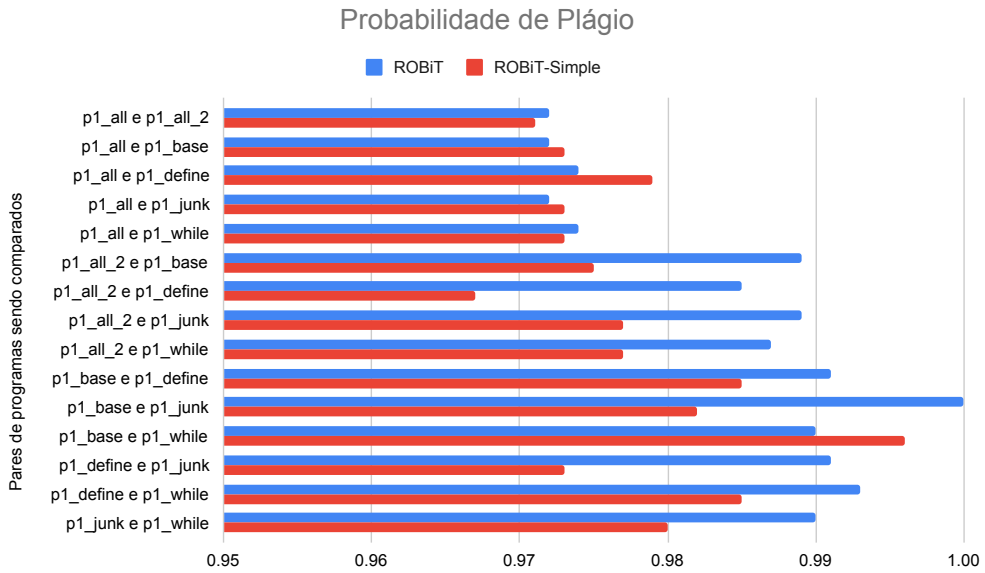

Figura 6.5 – Comparação da probabilidades de plágio identificadas pelas técnicas ROBiT e ROBiT-Simple, para cada par de programas analisados no Experimento 1.
Fonte: A Autora.

Em relação ao Experimento 1 (Figura 6.5), observamos que o MOSS foi capaz de identificar possíveis casos plágio em apenas dois casos: ao comparar os arquivos *p1-base* e *p1-define* (52% de chance de plágio); e os arquivos *p1-base* e *p1-while* (39% de chance de plágio). Note que esses percentuais não são visíveis na Figura 6.5, visto que eles são significativamente inferiores aos níveis de similaridade identificados pelos métodos ROBiT e ROBiT-Simple, os quais são muito mais eficazes. O desempenho relativamente positivo do MOSS nos dois casos mencionados pode ser compreendido a análise manual dos códigos em questão, resultantes da aplicação de transformações (geradas pelo ChatGPT) do código fonte original. Em particular, pudemos observar que nesses casos específicos, vários trechos dos códigos, embora estruturalmente equivalentes, permaneceram sintaticamente semelhantes, e portanto foram (corretamente) identificados como plágios pelo MOSS. Isso sugere que o prompt ChatGPT utilizado em nossos experimentos, durante o processo de geração de arquivos de plágio, pode não ter sido detalhado o suficiente a ponto de possibilitar a criação automatizada de plágios capazes de evadir téc-

nicas de detecção de similaridade. Esse é um ponto de possível melhoria futura em nosso trabalho; em particular, no que diz respeito à geração de diferentes tipos plágio tendo em vista a criação de arquivos de teste para avaliação de desempenho de diferentes técnicas anti-plágio.

O impacto dos diferentes tipos de transformação na capacidade de detecção de plágios também pode ser observado na Figura 6.5—a qual apresenta comparações entre todos arquivos binários associados ao Experimento 1. É possível notar alguns casos de interesse, como as comparações dos arquivos *p1-all* e *p1-define*, e dos arquivos *p1-base* e *p1-while*. Nesses casos, o método `ROBiT-Simple` apresenta melhor desempenho que o `ROBiT`. Isso provavelmente se deve ao fato de que o processo de otimização pode não ter sido capaz de "reverter" algumas transformações sintáticas e semânticas (aplicadas ao código fonte original) de forma a produzir um código binário suficientemente semelhante àquele produzido como representação intermediária durante a compilação sem otimizações. Por fim, considere agora a comparação dos arquivos *p1-base* e *p1-while*. Nesse caso, o código fonte original (*p1-base*) utiliza um laço *do-while* (Figura A.1), o qual é transformado, durante a criação do arquivo de plágio *p1-while* (Figure A.2), em um laço do tipo *while*. Neste caso, o processo de otimização implementado pelo `ROBiT` inadvertidamente mapeia tais construções para funções com grafos de representação levemente diferentes, enquanto que o `ROBiT-Simple` mantém uma similaridade maior em relação a tal estrutura de dados; isso poderia explicar por que, nesse cenário específico, o `ROBiT-Simple` produz uma estimativa da probabilidade de plágio mais acurada em relação ao `ROBiT`.

O desempenho do `MOSS` ao analisar códigos fonte do Experimento 2 é ainda menos conclusivo e preciso do que em relação ao Experimento 1. Ao comparar os arquivos *p2-def* e *p2-if*, por exemplo (os quais consistem, de fato, em plágios), o escore de similaridade produzido pelo `MOSS` ficou em torno de 26%. Algo semelhante ocorre durante a comparação dos arquivos *p2-all* e *p2-while*, a qual produziu em um escore de similaridade de apenas 27%. Note que, semelhantemente ao experimento anterior, tais percentuais não são visíveis na Figura 6.6, visto que eles são significativamente inferiores aos níveis de similaridade identificados pelos métodos `ROBiT` e `ROBiT-Simple`. As quatro comparações anteriores sofrem do mesmo mal discutido no contexto do Experimento 1: a concentração de transformações em pontos específicos do código fonte faz com que grandes trechos de código ainda permaneçam iguais, mesmo após a aplicação de modificações sintáticas e semânticas.

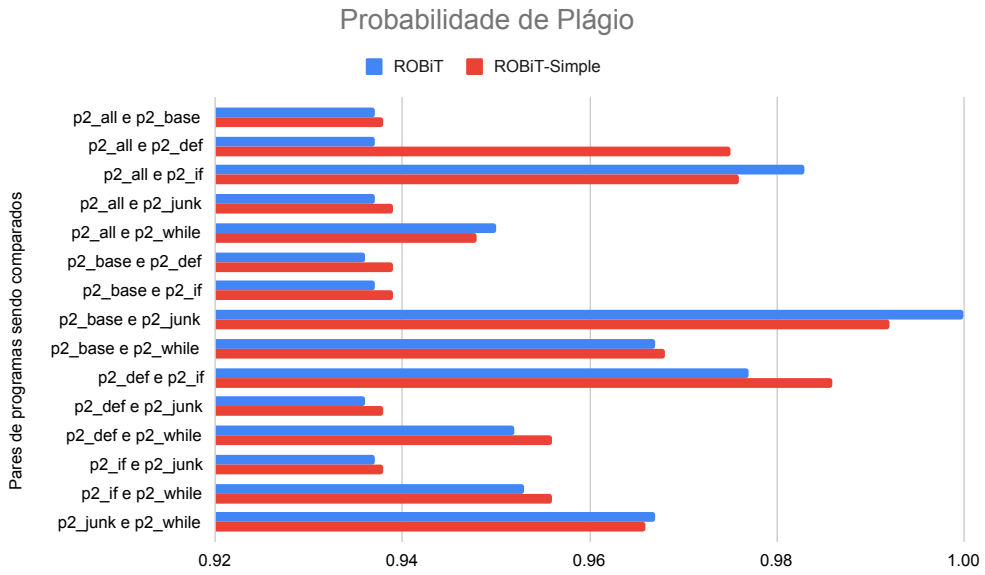

Figura 6.6 – Comparação da probabilidades de plágio identificadas pelas técnicas `ROBiT` e `ROBiT-Simple`, para cada par de programas analisados no Experimento 2.
Fonte: A Autora.

Ao analisarmos as comparações detalhadas apresentadas na Figura 6.6, podemos perceber que os métodos `ROBiT` e `ROBiT-Simple`, de forma geral, obtêm níveis de desempenho semelhantes, mas com algumas exceções.

Considere o caso de comparação dos arquivos *p2-all* (plágio gerado através da aplicação de *todas* as possíveis transformações) e *p2-def* (plágio em que apenas transformações afetando a definição e inicialização de variáveis). Nesse caso, a desempenho dos algoritmos `ROBiT` e `ROBiT-Simple` é substancial. Uma hipótese para isso é que método `ROBiT` executa um processo de otimização que tenta "reverter", no caso do arquivo *p2-def*, apenas um tipo de transformação, o que provavelmente permite a construção de um arquivo binário semelhante ao original. Por outro lado, o `ROBiT`, ao processar o arquivo *p2-all*, tenta "reverter" um número significativamente maior de transformações, o que torna menos provável a obtenção de um binário significativamente parecido com o original. Por esse motivo, é menos provável que a comparação entre esses binários indique um nível de semelhança alto o suficiente para indicar um potencial plágio; e, por esse motivo, o método `ROBiT` acaba tendo desempenho inferior ao `ROBiT-Simple`.

É importante notar que o método `ROBiT` reconhece com 100% de certeza a ocorrência de plágio em um caso emblemático, em ambos os experimentos: quando utilizamos funções de código morto e de desvios nunca aplicados (isto é, a transformação **Op13-ChAddJunk**), nosso método de otimização sempre reverte completamente tal transformação, produzindo um código binário equivalente ao código base. Isso é relevante pois

esse tipo de transformação é uma técnica conhecida por afetar todos tipos de representação intermediárias de um programa, e também por ser comumente utilizada em evasão de analisadores anti-plágio. Nesse contexto, podemos confirmar que `ROBiT` é eficaz em confirmar casos de plágio contra códigos produzidos através desse tipo de transformação.

Considerando os experimentos descritos anteriormente, podemos concluir que:

- Transformações sintáticas e semânticas podem afetar negativamente a eficácia do `MOSS` no que diz respeito a sua capacidade de prever casos de plágio, resultando em percentuais de similaridade baixos e, frequentemente, em casos de evasão completa durante a análise de plágios conhecidos.
- O algoritmo apresentado nesse trabalho, `ROBiT`, se mostra significativamente mais eficaz quando comparado com `MOSS`, tendo um nível maior de reconhecimento de plágios, assim como maior sensibilidade.
- `ROBiT` também se mostra equivalente ou levemente superior, nos casos mais desafiadores de detecção de plágio, quando comparado com o algoritmo `ROBiT-Simple`.
- Por fim, o `ROBiT` se mostra especialmente eficaz em casos de plágio baseados no uso de transformações do tipo **Op13-ChAddJunk**, as quais são largamente utilizada para a confecção de plágios.

7 CONCLUSÃO

O desenvolvimento de técnicas eficazes para detecção de plágio é uma tarefa complexa, tendo em vista que as instituições afetadas frequentemente não possuem os recursos necessários para lidar com o grande número de infrações em potencial. Nesse trabalho, Neste trabalho investigamos métodos baseados na análise de similaridade de códigos binários e propusemos um novo algoritmo, `ROBiT`, que servirá como complemento às técnicas existentes anti-plágio.

Conforme discutido nos Capítulos 2 e 3, embora vários métodos existentes sejam efetivos em casos diversos, eles—de forma geral—não exploram análises e comparações de arquivos binários, o que frequentemente implica que eles não são robustos frente a técnicas de evasão baseadas em sofisticas transformações semânticas e sintáticas. Nesse trabalho, propusemos uma nova técnica anti-plágio, a qual explora tal tipo de análise, e avaliamos empiricamente diversas hipóteses relacionadas à sua eficácia em diferentes cenários—com diversos graus de sofisticação—envolvendo análise de códigos fonte C adulterados com objetivo de burlar sistemas de detecção de plágio existentes.

Através de nossas análises experimentais, pudemos observar o comportamento de nosso método, `ROBiT`, observando suas capacidades e limitações. Em particular, pudemos comprovar experimentalmente que, comparado com métodos bem estabelecidos e amplamente utilizados, como o `MOSS`, nosso método obteve resultados consistentemente superiores tanto no que se diz respeito à quantidade de plágios descobertos, quanto ao grau de certeza informado pela técnica sobre a existência do plágio.

Além disso, ao compararmos a eficácia do método `ROBiT` em relação a uma técnica semelhante (`ROBiT-Simple`), que não explora processos de otimização durante a compilação, pudemos observar níveis de desempenho semelhantes. Em alguns casos de teste, o `ROBiT` demonstra desempenho claramente superior, mas, de forma geral, não é possível concluir com alta confiabilidade estatística que um método é sempre superior ao outro. Uma exceção diz respeito a casos emblemáticos tais como aqueles em que plágios são produzidos via transformações sintáticas do tipo **Op13-ChAddJunk**. Nesses casos, nosso método consistentemente apresentou desempenho superior tanto ao `MOSS` quanto ao `ROBiT-Simple`, sendo sempre capaz de reconhecer os plágios.

Como um possível ponto de evolução deste trabalho, consideramos a inclusão de um número maior de casos de testes a fim de mais precisamente avaliar o desempenho, limitações, e capacidades do algoritmo `ROBiT`, por exemplo, no contexto da análise de

códigos fonte longos. Outro possível trabalho futuro diz respeito à adaptação e integração do `ROBiT` com outras ferramentas anti-plágio, de forma a construir um método capaz de combinar diversos métodos complementares de análise de códigos fonte, com objetivo de mais precisamente identificar níveis de similaridade.

REFERÊNCIAS

AIKEN, A. Moss: A system for detecting software plagiarism. **<https://theory.stanford.edu/~aiken/moss/>**, 2004.

BIDERMAN, S.; RAFF, E. Fooling MOSS detection with pretrained language models. In: **Proceedings of the 31st ACM International Conference on Information & Knowledge Management**. New York, NY, USA: Association for Computing Machinery, 2022. (CIKM '22), p. 2933–2943. ISBN 9781450392365.

CHAE, D.-K. et al. Software plagiarism detection: a graph-based approach. In: **Proceedings of the 22nd ACM International Conference on Information & Knowledge Management**. New York, NY, USA: Association for Computing Machinery, 2013. (CIKM '13), p. 1577–1580.

DAMÁSIO, T. et al. A game-based framework to compare program classifiers and evaders. In: **Proceedings of the 21st ACM/IEEE International Symposium on Code Generation and Optimization**. New York, NY, USA: Association for Computing Machinery, 2023. (CGO 2023), p. 108–121. ISBN 9798400701016.

DEVORE-MCDONALD, B.; BERGER, E. D. Mossad: defeating software plagiarism detection. **Proc. ACM Program. Lang.**, Association for Computing Machinery, New York, NY, USA, v. 4, n. OOPSLA, November 2020.

DURACIK, M. et al. Abstract syntax tree based source code antiplagiarism system for large projects set. **IEEE Access**, v. 8, p. 175347–175359, 2020.

FAIDHI, J.; ROBINSON, S. An empirical approach for detecting program similarity and plagiarism within a university programming environment. **Computers & Education**, v. 11, n. 1, p. 11–19, 1987. ISSN 0360-1315.

Free Software Foundation. GCC Command Options: Options That Control Optimization. **GCC Online Documentation**, 2024. Available from Internet: <https://gcc.gnu.org/onlinedocs/gcc/Optimize-Options.html>.

GENG, C. et al. Can chatgpt pass an introductory level functional language programming course? **arXiv preprint arXiv:2305.02230**, 2023.

LEVENSHTEIN, V. I. Binary codes capable of correcting deletions, insertions, and reversals. **Soviet physics Doklady**, v. 10, p. 707–710, 1965.

LIU, C. et al. GPLAG: detection of software plagiarism by program dependence graph analysis. In: **Proceedings of the 12th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining**. New York, NY, USA: Association for Computing Machinery, 2006. (KDD '06), p. 872–881. ISBN 1595933395.

LUO, L. et al. Semantics-based obfuscation-resilient binary code similarity comparison with applications to software and algorithm plagiarism detection. **IEEE Transactions on Software Engineering**, v. 43, n. 12, p. 1157–1177, 2017.

NICHOLS, L. et al. Syntax-based improvements to plagiarism detectors and their evaluations. In: **Proceedings of the 2019 ACM Conference on Innovation and Technology in Computer Science Education**. New York, NY, USA: Association for Computing Machinery, 2019. (ITiCSE '19), p. 555–561.

POORNIMA, S.; MAHALAKSHMI, R. An inclusive report on robust malware detection and analysis for cross-version binary code optimizations. **International Journal on Recent and Innovation Trends in Computing and Communication**, v. 11, n. 9, p. 927–937, 2023.

PRECHELT, L.; MALPOHL, G.; PHILIPPSEN, M. Finding plagiarisms among a set of programs with jplag. **JUCS - Journal of Universal Computer Science**, Journal of Universal Computer Science, v. 8, n. 11, p. 1016–1038, 2002.

REN, X. et al. Unleashing the hidden power of compiler optimization on binary code difference: an empirical study. In: **Proceedings of the 42nd ACM SIGPLAN International Conference on Programming Language Design and Implementation**. New York, NY, USA: Association for Computing Machinery, 2021. (PLDI 2021), p. 142–157. ISBN 9781450383912.

SCHLEIMER, S.; WILKERSON, D. S.; AIKEN, A. Winnowing: local algorithms for document fingerprinting. In: **Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data**. New York, NY, USA: Association for Computing Machinery, 2003. (SIGMOD '03), p. 76–85. ISBN 158113634X.

WISE, M. J. YAP3: Improved Detection of Similarities in Computer Program and Other Texts. In: **Proceedings of the Twenty-Seventh SIGCSE Technical Symposium on Computer Science Education**. New York, NY, USA: Association for Computing Machinery, 1996. (SIGCSE '96), p. 130–134.

YENDURI, G. et al. Generative Pre-Trained Transformer: A Comprehensive Review on Enabling Technologies, Potential Applications, Emerging Challenges, and Future Directions. **ArXiv preprint ArXiv:2305.10435v2**, 2023.

ZHANG, W. et al. Challenging machine learning-based clone detectors via semantic-preserving code transformations. **IEEE Transactions on Software Engineering**, v. 49, p. 3052–3070, 2021.

ZHANG, W. et al. Challenging machine learning-based clone detectors via semantic-preserving code transformations. **IEEE Trans. Softw. Eng.**, IEEE Press, v. 49, n. 5, p. 3052–3070, may 2023. ISSN 0098-5589.

APÊNDICE A — EXEMPLO DE CÓDIGOS FONTE ANALISADOS

Apresentamos, a seguir, exemplos de dois códigos fonte analisados em nossos experimentos. O primeiro (Figura A.1) corresponde a um programa em sua forma original, enquanto que o segundo (Figura A.2) corresponde a um possível arquivo de plágio, construído com base no programa original, após a aplicação de uma possível operação de transformação.

```c
// Programa 1 - Original


#include <stdio.h>

int main(void) {
    float massa;
    int segundos = 0;
    printf("Insira a massa inicial (em gramas): ");
    scanf("%f", &massa);
    if (massa < 0.5) {
        printf("Valor inválido! Massa menor que 0.5 gramas\n");
        return 0;
    }
    do {
        massa /= 2;
        segundos += 50;
    } while (massa >= 0.5);

    printf("Massa Final: %f\n", massa);
    printf("Tempo: %d horas, %d minutos, %d segundos\n", segundos / 3600, (segundos % 3600) / 60, segundos % 60);
    return 0;
}
```

Figura A.1 – Código base do Experimento 1.
Fonte: A Autora.

```c
// Programa 1 - Plagio While


#include <stdio.h>

int main(void) {
    float massa;
    int segundos = 0;
    printf("Insira a massa inicial (em gramas): ");
    scanf("%f", &massa);
    if (massa < 0.5) {
        printf("Valor inválido! Massa menor que 0.5 gramas\n");
        return 0;
    }
    while (massa >= 0.5) {
        massa /= 2;
        segundos += 50;
    }

    printf("Massa Final: %f\n", massa);
    printf("Tempo: %d horas, %d minutos, %d segundos\n", segundos / 3600, (segundos % 3600) / 60, segundos % 60);
    return 0;
}
```

Figura A.2 – Código plagiado do Experimento 1 - Transformação do *while*
Fonte: A Autora.

APÊNDICE B — SCRIPT PARA EXECUÇÃO DO ALGORITMO ROBIT

Apresentamos, a seguir, na Figura B.1, o script Bash responsável pela execução do método ROBiT, introduzido no presente projeto de pesquisa.

```bash
#!/bin/bash

# Verifica se pelo menos um arquivo .c foi fornecido
if [ "$#" -lt 1 ]; then
  echo "Usage: $0 <file1.c> <file2.c> [... <fileN.c>]"
  exit 1
fi

# Nome do arquivo de saída para os resultados das comparações
output_file="radiff_results.txt"
# Limpa o arquivo de saída ou cria se não existir
> "$output_file"

# Compila cada arquivo .c fornecido
for file in "$@"; do
  if [[ $file == *.c ]]; then
    base_name=$(basename "$file" .c)
    gcc -O3 "$file" -o "$base_name" -lm
    if [ $? -ne 0 ]; then
      echo "Erro ao compilar $file"
      exit 1
    fi
  else
    echo "O arquivo $file não é um arquivo .c válido."
    exit 1
  fi
done

# Coleta os nomes dos executáveis compilados
compiled_files=()
for file in "$@"; do
  if [[ $file == *.c ]]; then
    compiled_files+=($(basename "$file" .c))
  fi
done

# Executa radiff2 em todos os pares possíveis e salva a saída no arquivo de resultados
for ((i = 0; i < ${#compiled_files[@]}; i++)); do
  for ((j = i + 1; j < ${#compiled_files[@]}; j++)); do
    file1="${compiled_files[i]}"
    file2="${compiled_files[j]}"
    # Captura a saída do radiff2
    radiff_output=$(radiff2 -ss "$file1" "$file2")
    # Extrai as linhas relevantes (pode haver uma linha de tamanho de arquivo, além de similarity e distance)
    relevant_lines=$(echo "$radiff_output" | grep -E 'similarity:|distance:')
    # Escreve no arquivo de resultados
    echo "Comparando $file1 e $file2:" >> "$output_file"
    echo "$relevant_lines" >> "$output_file"
    echo "" >> "$output_file" # Linha em branco para separar as entradas
  done
done

echo "Resultado das comparações salvo em $output_file"
```

Figura B.1 – Método ROBiT— Código do script Bash responsável pela execução da técnica proposta nesse trabalho.
Fonte: A Autora.