Impacto de Ofuscadores e Otimizadores de Código na Acurácia de Classificadores de Programas

Thaís Damásio
UFMG
Minas Gerais, Brazil
thais.damasio@dcc.ufmg.br

Michael Canesche
UFMG
Minas Gerais, Brazil
michaelcanesche@dcc.ufmg.br

Vinícius Pacheco
UFMG
Minas Gerais, Brazil
vinicius.pacheco@dcc.ufmg.br

Anderson Faustino da Silva
UEM
Paraná, Brazil
anderson@din.uem.br

Fernando M. Quintão Pereira
UFMG
Minas Gerais, Brazil
fernando@dcc.ufmg.br

ABSTRACT

O problema de detecção de clones consiste em determinar, dado um programa inicial e classes de problemas, qual problema o programa resolve. A detecção de clones é útil em vários contextos diferentes, incluindo a identificação de código malicioso e a identificação de plágio. Uma vez que a solução exata para detecção de clones é uma impossibilidade teórica, esse problema é resolvido via heurísticas; a maior parte delas de natureza estatística. Este trabalho analisa o impacto de técnicas de transformação de código sobre essas heurísticas. Mostramos que otimizações de código padrão são tão efetivas quanto ofuscadores para enganar as heurísticas mais populares para detecção de clones, gerando, contudo, código ordens de magnitude mais eficientes. Mostramos também que histogramas de instruções são tão efetivos quanto representações mais complexas ao executar as heurísticas. E por fim mostramos que otimizações de código podem funcionar como um normalizador de programas, em alguns casos, revertendo o efeito da ofuscação, e recuperando assim a capacidade de um detector estatístico de encontrar clones.

KEYWORDS

obfuscation, neural network, compiler optimizations

ACM Reference Format:
Thaís Damásio, Michael Canesche, Vinícius Pacheco, Anderson Faustino da Silva, and Fernando M. Quintão Pereira. 2022. Impacto de Ofuscadores e Otimizadores de Código na Acurácia de Classificadores de Programas. In *XXVI Brazilian Symposium on Programming Languages (SBLP 2022), October 6–7, 2022, Virtual Event, Brazil.* ACM, New York, NY, USA, 8 pages. https://doi.org/10.1145/3561320.3561322

1 INTRODUÇÃO

O problema da *detecção de clones* [21] consiste em um programa inicial P_1 e determinar se outro programa P_2 é um clone de P_1 de modo que ambos resolvam o mesmo problema. A detecção de clones é útil por várias razões. Duas dentre as principais razões são a detecção de plágio [4] e a detecção de código malicioso (*malware*) [28]. Esse problema é indecidível—a prova de indecidibilidade é um clássico corolário do Teorema de Rice [19]. Assim, a detecção de programas equivalentes é resolvida via abordagens muito conservadoras [25], ou via métodos estatísticos [27].

Entretanto, a inexistência de uma solução exata para a detecção de clones dá margem a diversos métodos de evasão, alguns dos quais bastante efetivos [18]. Em particular, existem hoje diversas técnicas de ofuscação de código que já foram usadas para derrotar classificadores bem conhecidos de programas [9, 31]. Este artigo revisita tais técnicas, e as compara contra uma forma simples de transformação de programas: a otimização de código.

Jogos Adversariais. Este artigo analisa a eficácia de técnicas de ofuscação e otimização de código—daqui por diante chamadas de técnicas de *transformação*—para derrotar classificadores estatísticos de programa. Tal análise é feita mediante a investigação de quatro perguntas de pesquisa:

(1) Quais formatos de representação de código são eficazes para detectar clones?
(2) Qual é o impacto de técnicas típicas de ofuscação de código no tempo de execução de programas?
(3) Qual é a efetividade relativa de diferentes técnicas de transformação de código para confundir classificadores de programas?
(4) Qual é o impacto de otimizações padrão de código como uma metodologia para normalizar programas a fim de recuperar a eficácia de classificadores?

Essas perguntas são analisadas em um arcabouço de *jogos adversariais*, definido na Seção 2 do presente trabalho. Um jogo consiste em um *classificador* e em um *plagiador*. O classificador é vitorioso caso ele consiga identificar programas equivalentes com acurácia acima de um nível pré-estabelecido. O plagiador é vitorioso caso sua estratégia consiga manter a acurácia abaixo daquele nível. Este artigo estuda quatro variações deste jogo: cada arranjo varia a quantidade de informação que é dada ao classificador, e as transformações de código às quais o plagiador tem acesso. A Seção 3 explicita os modelos de classificação e os transformadores de código avaliados neste artigo.

Resumo de Descobertas. Nós construímos um arcabouço de classificação de programas a partir de técnicas previamente descritas na literatura [13]. Tal arcabouço classifica programas usando para

SBLP 2022, October 6–7, 2022, Virtual Event, Brazil

ACM ISBN 978-1-4503-9744-5/22/10…$15.00
https://doi.org/10.1145/3561320.3561322

tanto a representação intermediária que LLVM [11] produz para o código que compila. A Seção 4 descreve uma série de experimentos realizados sobre tal arcabouço. A partir desses experimentos, pudemos derivar uma série de observações sobre a eficácia de diferentes técnicas de transformação de código para enganar classificadores estatísticos, as quais resumimos abaixo:

(1) O *histograma de instruções* de um programa é efetivo para garantir a eficácia de classificadores contra técnicas de ofuscação tão agressivas quanto aquelas descritas por Zhang et al. [31].
(2) Níveis de otimização de código padrão (`clang -O3`, por exemplo) são tão eficazes para derrotar classificadores estatístico quanto técnicas de ofuscação de código construídas especificamente para tal fim.
(3) A otimização de código funciona como um *normalizador de programas*, sendo capaz, em muitos casos, de recuperar—em sua totalidade—a eficiência de classificadores estatísticos contra diferentes ofuscadores.
(4) O conhecimento prévio do ofuscador torna o classificador resistente contra várias transformações de código, sem que nenhuma adaptação sobre a técnica de classificação seja necessária.

2 QUATRO JOGOS ADVERSARIAIS

A fim de averiguar como métodos de transformação de código (definidos na Seção 2.1) impactam a acurácia de técnicas de classificação de programas (definidas na Seção 2.2), criaremos um sistema de *jogos*, o qual é definido na Seção 2.3.

2.1 Transformadores de Código

Um programa de computador pode ser entendido como uma descrição de passos para controlar uma máquina de estados, no caso, o computador. Algumas mudanças de estado têm efeitos *observáveis*, isso é, podem ser percebidas por usuários do programa. Exemplos de efeitos observáveis incluem escrita de dados em dispositivos de saída, escrita em memória e leitura de dados de entrada (cujo efeito é consumir valores na fila de entrada do programa). A partir deste conceito de semântica observável, a Definição 2.1 define o conceito de transformação de código—conceito central para a apresentação deste artigo.

Definition 2.1 (Transformação de Código). Uma transformação de código é qualquer função de programas para programas. A transformação é dita *preservadora de semântica* se o programa transformado sempre produz a mesma semântica observável que o programa original. Caso a linguagem de programação que representa programas originais e transformados seja a mesma, a transformação é dita *monoglótica*. A implementação de uma transformação de código é chamada um *transformador de programas*.

Neste artigo, estamos interessados somente em transformadores de código monoglóticos. Exemplos típicos de tais transformadores são os *otimizadores de código*—ferramentas presentes na maior parte dos compiladores de uso industrial. Existe, entretanto, uma outra família de transformações de código monoglóticas bem conhecida: os *ofuscadores*. Um ofuscador é uma ferramenta que altera o código de um programa a fim de dificultar seu entendimento. Exemplo 2.2 ilustra uma dessas transformações.

Example 2.2. O ofuscador de código `O-LLVM` contém três transformações monoglóticas[1]. Uma dessas transformações é chamada *substituição de instruções*. Uma substituição altera uma única instrução por vez. Substituições são definidas para as seguintes instruções: `xor`, `or`, `and`, `sub` e `add`. Com relação a este último mnemônico, uma operação como `a = b + c` gera a seguinte sequência: `r = rand(); a = b + r; a = a + c; a = a - r`.

Uma das observações em torno da qual este artigo é construído é o fato de otimizações de código muitas vezes serem capazes de reverter o efeito de ofuscações. Em outras palavras, as versões otimizadas de um programa, e sua contra-parte ofuscada podem ser idênticas. O Exemplo 2.3 utiliza o ofuscador mencionado no Exemplo 2.2 para ilustrar essa observação.

Example 2.3. A Figura 1 mostra o código LLVM que resulta da compilação de dois programas com o comando `clang -O1 -emit-llvm -c -S`. O programa visto na A Figura 1 (c) resulta da substituição de instruções (vide Exemplo 2.2) aplicada ao programa da Figura 1 (a). Exceto pela chamada de `rand()` na Linha 01 da Figura 1 (d), o programa ofuscado e o programa original geram o mesmo código.

(a)
```
01 int foo(int b, int c) {
02   int a = b + c;
03   return a;
04 }
```

(b)
```
01 %add = add nsw i32 %c, %b
02 ret i32 %add
```

(c)
```
01 int foo(int b, int c) {
02   int r = rand();
03   int a = b + r;
04   a = a + c;
05   a = a - r;
06   return a;
07 }
```

(d)
```
01 %call = call i32 @rand()
02 %add = add i32 %c, %b
03 ret i32 %add
```

Figure 1: (a-b) Programa original e código otimizado por `clang -O1`. (c-d) Programa ofuscado e código otimizado por `clang -O1`.

2.2 Classificadores de Programas

Dois programas são ditos equivalentes caso eles produzam a mesma saída, dadas as mesmas entradas. O problema de provar a equivalência entre dois programas é indecidível, conforme já demonstrado via diferentes modelos de computação [22]. Ainda assim, é possível utilizar heurísticas para determinar se dois problemas são equivalentes. No contexto deste trabalho, estamos interessados em classificar programas de acordo com o problema que eles resolvem. Esse problema é formalizado na Definição 2.4.

Definition 2.4 (Classificação de Código). Um *problema computacional* é dado por uma especificação, um conjunto de n entradas válidas e um conjunto de n saídas esperadas. Um programa *resolve* um problema quando ele sempre produz a saída esperada dada uma entrada. Seja f um programa, e seja $P = \{p_1, p_2, \ldots, p_k\}$ uma lista

[1] Para saber mais: https://github.com/obfuscator-llvm

de k problemas, tal que f resolva um deles. O problema de classificação de programas consiste em descobrir qual problema f resolve. Uma solução para este problema é chamada um *Classificador de Programas*.

2.3 Descrição dos Quatro Jogos

Neste artigo, a acurácia de classificadores de programas será medida via um arcabouço de jogos adversariais. A Definição 2.5 formalize esses jogos.

Definition 2.5 (Jogo de Classificação). Um jogo de classificação é formado por um *classificador* de programas; por um *adversário*, também chamado um *plagiador*; por um limiar de acurácia K; por um conjunto de problemas P e por uma base de programas $F = F_c \ [\ F_p$. Cada programa $f \ 2 \ F$ resolve algum problema $p \ 2 \ P$. O classificador conhece a relação de programas e problemas em F_c; o plagiador conhece essa relação em F_p. O jogo acontece em duas fases:

(1) O plagiador escolhe um programa $f \ 2 \ F_p$ e o entrega para o classificador, possivelmente transformado.
(2) O classificador indica um problema p que ele acredita que f resolva.

O classificador ganha o jogo caso ele identifique p com probabilidade acima de K; doutro modo, ele perde o jogo.

Neste artigo são discutidos quatro jogos adversariais. Cada jogo varia de acordo com o acesso que classificador e plagiador têm a transformadores de código (veja Definição 2.1). A Figura 2 resume esse acesso. Dois jogos são simétricos: classificador e plagiador têm acesso a recursos do mesmo tipo. Ainda assim, note que os recursos não são os mesmos. Por exemplo, plagiador e classificador podem ter acesso a partes distintas de uma mesma base de programas e problemas em um jogo simétrico.

Jogo 0 (simétrico)	
Classificador	Dataset
Plagiador	Dataset

Jogo 1 (assimétrico)	
Classificador	Dataset
Plagiador	Dataset + Ofuscador

Jogo 2 (simétrico)	
Classificador	Dataset + Ofuscador
Plagiador	Dataset + Ofuscador

Jogo 3 (assimétrico)	
Classificador	Dataset + Otimizador
Plagiador	Dataset + Ofuscador

Figure 2: Quatro jogos adversariais.

2.4 Jogo 0

No Jogo-0, classificador e plagiador têm somente acesso à base de dados. Esse acesso é *balanceado*. Em outras palavras, para cada problema $p \ 2 \ P$, a base do classificador (F_c) contém uma quantidade n_c; $n_c > 1$ de programas que resolvem p. De forma similar, a base do plagiador (F_p) contém uma quantidade n_p; $n_p > 1$ de programas que resolvem p. O balanceamento garante que cada problema seja conhecido tanto pelo classificador quanto pelo plagiador. O acesso é também *disjunto*: $F_c \ \setminus \ F_p = ;$.

2.5 Jogo 1

No Jogo-1, o acesso à base de dados segue o Jogo-0 (sendo garantido o balanceamento e a disjunção dos dados). Contudo, o plagiado tem acesso a um ofuscador de código. O plagiador pode, assim, transformar um programa antes de desafiar o classificador. Em outras palavras, seja T um ofuscador qualquer de programas. Dado $f \ 2 \ F_p$, seja $f^0 = T^1 f^o$ a versão de f produzida por T. O plagiador entrega f^0 ao classificador. Este não tem acesso ao programa original f. Uma vez que o plagiador tem acesso a uma quantidade maior de recursos que o classificador, esse jogo é assimétrico.

2.6 Jogo 2

O Jogo-2 é uma versão simétrica do Jogo-1. Neste caso, ao classificador é dado acesso ao mesmo transformador de código usado pelo plagiador. Desse modo, o classificador pode treinar um modelo de classificação que usa os programas originalmente presentes em F_c transformados por T. Note que na fase de desafio é sempre usado um programa transformado, assim como é feito no Jogo-1. Isso quer dizer que o plagiador sempre entrega ao classificador a versão transformada de um programa de F_p.

2.7 Jogo 3

No Jogo-3, o classificador e o plagiador têm, ambos, acesso a um transformador de código. Entretanto, esses transformadores são distintos. Neste artigo, assume-se que o plagiador usa um ofuscador de código T, e que o classificador usa um otimizador de código N; $N \ , \ T$. O classificador pode então usar N como um *normalizador de programas*. O classificador realiza a calibragem de um modelo de predição sobre programas de F_c transformados, isto é, normalizados, por N. Ao receber o programa de desafio f^0; $f^0 = T^1 f^o$; $f \ 2 \ F_p$, o classificador o transforma em um programa $f" = N^1 f^{0o}$, antes de realizar a predição. O Jogo-3 tem por objetivo avaliar quais tipos de transformadores de código são bons normalizadores de programas, e quais tipos de transformadores de código são resistentes à normalização.

3 ARENA DE CLASSIFICAÇÃO

A Seção 4 deste trabalho irá analisar a acurácia de diferentes classificadores e plagiadores de código em cada um dos quatro jogos que a Figura 2 resume. Tais jogos usam diferentes implementações de transformadores de código e diferentes implementações de modelos de classificação. A Seção 3.1 descreve os transformadores considerados neste trabalho, ao passo que a Seção 3.2 descreve os classificadores utilizados.

3.1 Transformadores Utilizados no Trabalho

Tanto otimizadores quanto ofuscadores de código são transformadores. Neste artigo, consideramos um otimizador, e dois ofuscadores, os quais são descritos a seguir.

O-LLVM: O Obfuscator-LLVM [9] é uma ferramenta de código-aberto que ofusca a representação intermediária do compilador LLVM. Essa

ferramenta provê aos seus usuários três transformações monoglóticas. Uma dessas transformações, a substituição de instruções (SI), já foi explicada no Exemplo 2.2. As demais são descritas abaixo:

- **CFF:** *Control-Flow Flattening* arranja o fluxo de controle do programa em um comando *switch* dentro de um laço. A cada iteração do laço, o *switch* determina o próximo bloco básico a executar. Cada desvio do programa original é substituído por um comando que determina o próximo caso do comando *switch*.
- **BCF:** *Bogus-Control Flow* adiciona ao programa sequências de laços e desvios condicionais que não executam ou que executam uma vez. Essas novas estruturas são controladas por condições que não podem ser facilmente resolvidas em tempo de compilação. Assim, elas tendem a serem preservadas pelo compilador, mesmo em face de otimizações de código.

`CloneGen`: *Clone Generator* [31] é um arcabouço que suporta o uso de diferentes heurísticas para a geração de clones de código com preservação de semântica. Cada heurística determina uma forma diferente de combinar 15 transformações de código. Essas transformações—monoglóticas—são aplicadas sobre código fonte escrito em C ou C++. O Exemplo 3.1 ilustra uma dessas transformações.

Example 3.1. A figura 3 mostra uma, dentre as 15 transformações de código, descritas por Zhang et al. [31]. Essa transformação, que Zhang et al. chama *Op4-ChDo*, consiste em substituir um laço *do-while* por um laço *while-do*.

```
(a) 01 do {                    (b) 01 [Código X]
    02   [Código X]                02 while(condição){
    03 }while (condição)            03   [Código X]
                                    04 }
```

Figure 3: (a) Programa original. (b) Programa transformado com a técnica *Op4-ChDo*.

Zhang et al. define quatro heurísticas para escolher a ordem em que transformações de código são aplicadas sobre programas. Porém, nós conseguimos utilizar somente duas delas, as quais serão descritas a seguir. As outras duas geraram código incorreto em nossos experimentos. As heurísticas que utilizamos são:

- **MCMC:** *Markov Chain Monte Carlo* usa um algoritmo de *n-grama* para decidir quais as modificações devem ser aplicadas em um programa
- **DRLSG:** *Deep Reinforcement Learning Sequence Generation*, uma técnica de aprendizado por reforço cujo modelo de recompensa é customizado para confundir os classificadores.

`-O3`: É uma sequência de passes de otimização aplicadas no programa pelo compilador `clang`. O `-O3` prioriza a performance do binário gerado, mesmo que isso resulte no aumento do tamanho do arquivo.

3.2 Classificadores Utilizados

Neste artigo foram considerados cinco modelos de aprendizado para avaliar os jogos descritos na Seção 2. Tais modelos são descritos a seguir.

K-Nearest Neighbors (KNN) [8]: Utiliza a distância Euclidiana entre histogramas de instruções para calcular a distância entre programas. Um programa f é classificado como resolvedor de um problema p caso p seja o problema mais comum entre os problemas resolvidos pelos K programas mais próximos de f. Considera-se $K = 10$ neste trabalho.

Random Forest [3]: Este classificador utiliza n árvores de decisões treinadas sobre o histograma de instruções de programas selecionados aleatoriamente do conjunto de dados. Neste trabalho, considera-se $n = 100$.

Support Vector Machine (SVM) [5]: Este modelo encontra um *hiperplano ótimo* que separe os histogramas por classe. Neste trabalho, SVM foi treinado com a função de perda *squared hinge*.

Regressão Logística: Prediz o problema que um programa resolve a partir da aplicação de uma função logística (a sigmóide) ao resultado de uma regressão linear sobre o histograma de instruções do programa.

Multi-Layer Perceptron (MLP): Usa uma rede neural [20] com 50 camadas escondidas, alimentada com o histograma de instruções, para classificar programas.

4 AVALIAÇÃO

Esta seção avalia as perguntas de pesquisa enumeradas na Seção 1. A primeira dessas questões analisa a eficácia de diferentes técnicas de representação de programas para detectar clones. A segunda questão estuda o impacto relativo de ofuscação e otimização de código no desempenho de programas. Às duas questões restantes avaliam os quatro jogos descritos na Seção 2 do presente trabalho.

4.1 Representações de Código

A detecção de clones requer que programas sejam comparados. A comparação de programas, por sua vez, exige que tais programas sejam convertidos em um formato adequado. Tal formato é normalmente conhecido como uma *representação de código*. A literatura contém várias funções que mapeiam programas para representações de código diferentes, tais como `CFG` [2], `CDFG` [2], `ProgramL` [6], `ir2vec` [24] e `Milepost` [14]. Além dessas funções, uma representação comumente utilizada é o histograma de opcodes do programa. Esta seção compara a eficácia relativa das seis representações de código mencionadas acima, mais formas compactas de `CFG` e `CDFG`, extraídas do compilador `YaCoS` [29]. Uma descrição detalhada de cada uma dessas representações não é necessária para o entendimento de nossas conclusões. Basta saber que o objetivo de cada uma dessas funções é transformar um programa em um vetor multidimensional que possa ser usado, por exemplo, como entrada para uma rede neural, ou para um algoritmo de busca por similitude, como o k-NN. A seguir é descrito o arcabouço experimental usado para comparar as diferentes representações.

Hardware: Intel(R) Core(TM) i7-3770 usando uma CPU at 3.40GHz com 32GB de memória RAM.

Software: Sistema operacional Linux Ubuntu 20.04. As representações de código são usadas como dados de entrada em um modelo de classificação baseado nas redes neurais de Zhang et al. [30]. Essas redes neurais profundas foram idealizadas para classificar programas. Cada programa é convertido em uma representação de código,

e então fornecido como dado de treinamento ou validação para a rede.

Benchmarks: Utilizamos o conjunto POJ-104 [13] que é uma coleção de submissões de programas em C/C++ para resolver classes de problemas apresentados por um sistema pedagógico de juízes online[2]. Existem 104 classes de problemas nesta coleção e cada classe contém 500 submissões. Não foi possível usar todas as 104 classes, devido à quantidade de memória exigida por representações de código como `PrograML`, por isso utilizamos apenas 32 delas.

Metodologia: A rede de Zhang et al. foi treinada e validada com 80% dos benchmarks. Assim, a base de treinamento/validação contém 32 classes de problemas, cada uma com 300/100 programas. Os dados restantes, 32 classes, cada uma com 100 programas, são usados como dados de teste.

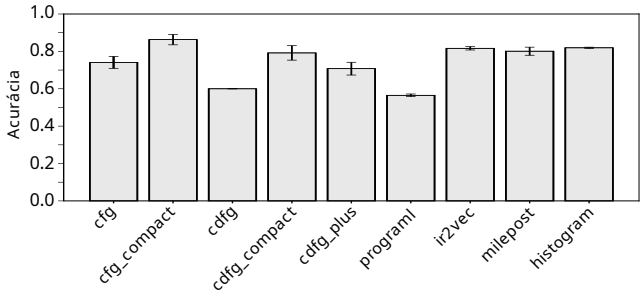

Figure 4: Acurácia do uso de histograma, na classificação, frente a outras representações.

Discussão. A Figura 4 mostra os resultados da comparação das diferentes representações de programa. Uma conclusão imediata deste experimento é que histogramas de opcodes são representações efetivas para o treinamento de modelos de classificação. Em nossa opinião, esse é um resultado surpreendente, pois os histogramas são, junto com as funções `Milepost`, a representação mais econômica que usamos. Essas duas representações usem menos que 2/5 da memória requerida, por exemplo, por `CFG`, `CDFG` e `PrograML`.

4.2 Ofuscação e Desempenho

Uma das conclusões deste trabalho é que técnicas de otimização de código são preferíveis a técnicas de ofuscação. Os dois transformadores possuem eficiência similar no que tange a capacidade de enganar o classificador de programas. Contudo, métodos de ofuscação têm uma grande desvantagem: eles causam uma perda de desempenho notável sobre o código transformado. Esta seção busca medir essa perda.

Hardware: Intel(R) Core(TM) i5-1035G1 de 4 núcleos, com clock de 3,6 GHz e 8 GB de memória RAM disponível. Nível 1 de caches de dados e de instruções com 128 KB.

Software: Linux Manjaro 21.2.6 (5.17.1-3-generic x86_64). Programas foram compilados com `clang` versão 13.0.1.

Benchmarks: Foram utilizados os 20 programas disponíveis no projeto *The Benchmark Game*[3]. Essa página web compara diferentes linguagens de programação utilizando *crowdsourcing*. Nós utilizamos, neste experimento, a versão mais rápida da solução em C para cada um dos 20 problemas de programação.

Metodologia: tempos de execução são médias de 16 execuções. Tempo é medido via o utilitário `time`, disponível no sistema Linux por padrão. Cada programa é alimentado com a sua entrada padrão (*standard input*) disponível no website *The Benchmark Game*.

Discussão. A Figura 5 compara o tempo de execução de diferentes versões de cada benchmark. Resultados são normalizados em função do tempo de execução do programa compilado com `clang -O0`, isso é, sem qualquer otimização. Esses programas são chamados *executáveis padrão*. Barras negativas indicam regressão de desempenho. Barras positivas indicam aceleração de desempenho. Os benchmarks *mandelbrot* e *n-body* não puderam ser executados após ofuscação. OLLVM gera código que não termina para esses programas.

Em média (média geométrica), programas executáveis otimizados com `clang -O3` são 7.46x mais rápidos que os executáveis padrão. Esses programas, quando ofuscados por `O-LLVM` ficam 8.33x mais lentos. Otimizações de código conseguem recuperar um pouco dessa perda de desempenho, porém não totalmente. Executáveis ofuscados e em sequência otimizados com `clang -O3` são 3.03x mais lentos. Em termos absolutos, os programas originais levam 7.93 segundos para executar uma vez. Esses programas, otimizados por `clang -O3` executam em 1.06 segundos. Ofuscados por `O-LLVM` os programas executam em 53.15 segundos. Se os programas ofuscados são otimizados por `clang -O3`, então eles executam em 21.05 segundos.

4.3 Avaliação de Acurácia

Esta seção reporta a acurácia dos modelos descritos na Seção 3.2 para avaliar os métodos de transformação de código descritos na Seção 3.1 em cada um dos quatro jogos vistos na Seção 2.3. O hardware usado nesta seção não influi nos resultados obtidos, então será omitido.

Benchmarks: Assim como na Seção 4.1, foram utilizados todos os programas disponíveis na base POJ-104. Porém, usamos todas as 104 classes de problemas disponíveis, uma vez que estaremos nos atendo unicamente aos histogramas de instruções – uma representação de código cujo consumo de memória não é alto.

Metodologia: resultados são reportados em termos de *acurácia*. A acurácia de um classificador é definida pela fórmula: $^1VP + VN^0 \bullet {}^1VP + VN + FP + FN^0$. A Definição 4.1 formaliza esses quatro termos.

Definition 4.1. Seja p um problema, f um programa e F_p o conjunto de programas que resolve o problema p. Dado um classificador, os seguintes resultados são possíveis:

- **Verdadeiro-Positivo (VP):** f resolve p e o classificador prevê que $f \in F_p$;
- **Verdadeiro-Negativo (VN):** f não resolve p e o classificador prevê que $f \notin F_p$;
- **Falso-Positivo (FP):** f não resolve p e o classificador prevê que $f \in F_p$;
- **Falso-Negativo (FN):** f resolve p e o classificador prevê que $f \notin F_p$.

4.3.1 Avaliação do Jogo 0. A Figura 6 apresenta os resultados referente ao jogo 0. O *Random Forest* atingiu cerca de 81% de acurácia

[2] http://programming.grids.cn/

[3] https://benchmarksgame-team.pages.debian.net/benchmarksgame/

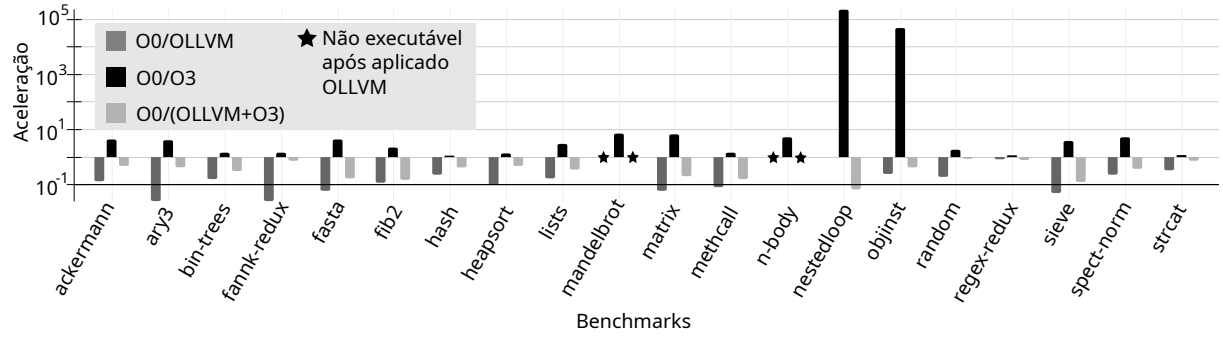

Figure 5: Impacto da ofuscação e das otimizações sobre os programas (maior melhor).

e foi o melhor entre os modelos avaliados. Note que a chance de um classificador aleatório acertar um problema é de 0.9%; logo, esse resultado demonstra que os classificadores conseguem reconhecer classes de programa de forma consistente. Tal resultado não é surpreendente, pois já havia sido observado anteriormente na literatura [13]. SVM foi o modelo com pior resultado—consequência da forma como ele resolve problemas multi-classes. SVM é um modelo para classificação binária. Ele é adaptado para problemas multi-classe via uma técnica chamada "um-vs-resto". Essa adaptação tende a ser imprecisa quando comparada com modelos projetados para multi-classificação [7]. Um-vs-resto também é ineficiente em termos de tempo, pois para cada classe c na base de dados ele cria um modelo m_c responsável por reconhecer se uma nova entrada é da classe c ou não. Assim, dentre todos os modelos, SVM gastou mais tempo na fase de treinamento como pode ser visto na tabela 2.

Modelo	KNN	Random Forest	SVM	MLP	Regressão Logística
Tempo (segundos)	0.07	3.12	576.75	209.28	32.47

Table 2: Tempo de execução dos modelos no Jogo 0.

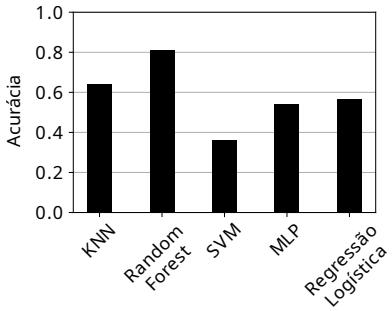

Figure 6: Resultados do Jogo 0. Os modelos deste experimento foram treinados e testados com a base de dados original.

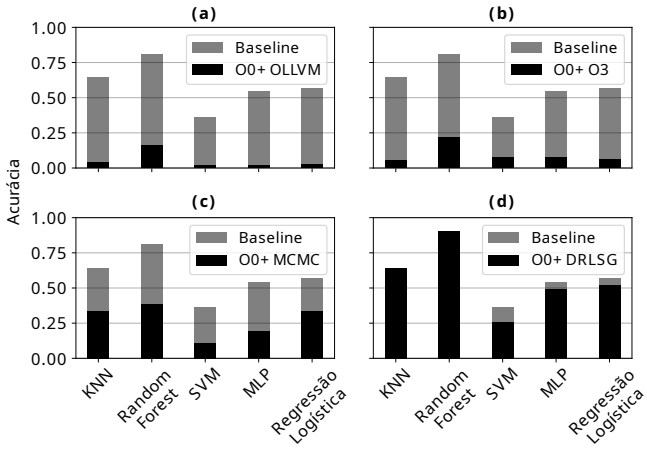

Figure 7: Resultados do Jogo 1, onde o *baseline* é o Jogo 0. Modelos testados com: (a) programas compilados com as estratégias CFF, SI e BCF do `O-LLVM`, em sequência; (b) programas compilados com `-O3`; (c) programas compilados com a estratégia MCMC do `CloneGen` e (d) programas compilados com a estratégia DRLSG do `CloneGen`.

4.3.2 Avaliação do Jogo 1. A Figura 7 mostra os resultados de modelos treinados com a base de dados original, mas testados com a base de dados modificada via diferentes técnicas de transformação. Desse experimento derivamos três conclusões. A primeira delas é que diversas técnicas de ofuscação (vistas nas Figuras 7-a e 7-c) são capazes de confundir os classificadores. A segunda conclusão é que um otimizador de código pode ser tão eficiente quanto um ofuscador para obter esse efeito, conforme pôde ser visto na Figura 7-b. Nessa figura, programas são transformados via `-O3`. Finalmente, pudemos constatar—não sem surpresa—que técnicas de ofuscação ingênuas como DRLSG (de Zhang et al.) não têm qualquer efeito sobre o classificador, conforme demonstra a Figura 7-d). DRLSG insere uma série de transformações pontuais no programa, substituindo constantes por operações aritméticas, por exemplo. Porém, uma vez convertido para o formato de atribuição estática único (SSA) que LLVM usa, todas essas transformações são desfeitas.

4.3.3 Avaliação do Jogo 2. A Figura 8 mostra a acurácia dos modelos treinados e testados com a base de dados transformada. Duas

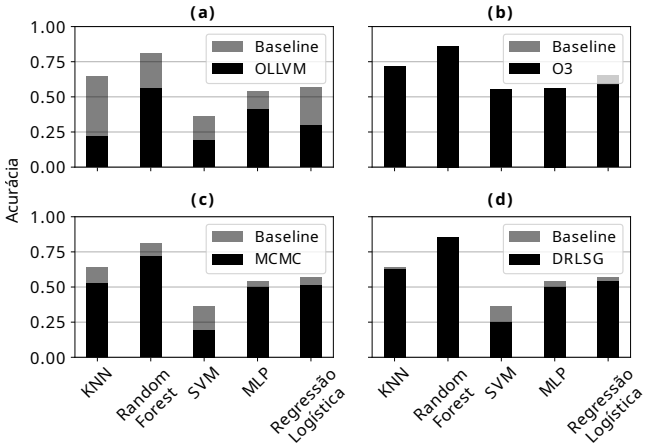

Figure 8: Resultados do Jogo 2, onde o *baseline* é o Jogo 0. Nas fases de treino e teste foram usados: (a) programas compilados com as estratégias CFF, SI e BCF do `O-LLVM`, em sequência; (b) programas compilados com `-O3`; (c) Programas compilados com a estratégia MCMC do `CloneGen` e (d) Programas compilados com a estratégia DRLSG do `CloneGen`.

conclusões podem ser derivadas desse experimento. A primeira delas é que quando o classificador tem acesso ao método de transformação aplicado pelo adversário ele consegue recuperar significativamente sua acurácia. A diferença entre as barras pretas e cinzas nos diferentes gráficos da Figura 8 torna tal fato evidente. A segunda conclusão é que o acesso ao transformador compromete mais a eficácia do otimizador que de ofuscadores como `O-LLVM`. Ainda que tal acesso aumente a acurácia do classificador contra um adversário equipado com `O-LLVM`, uma diferença notável ainda persiste para a versão base, conforme pode ser visto na Figura 8-a.

4.3.4 Avaliação do Jogo 3. A Figura 9 mostra modelos que normalizam os programas utilizando o transformador `-O3`. Existem ofuscadores cuja eficácia pode ser totalmente revertida por esse tipo de normalização de código. Tal é o caso de diferentes ofuscadores usados por Zhang et al., conforme visto nas Figuras 9-b e 9-c. Nesse cenário, o classificador se comporta como no Jogo 0. Entretanto, a normalização não recupera a acurácia do classificador frente ao ofuscador `O-LLVM`, conforme visto na Figura 9-a. A técnica BCF de `O-LLVM` insere caminhos não são executados no programa, porém nem mesmo LLVM em seu maior nível de otimização consegue remover tais caminhos. Assim, as novas características do programa inseridas pelo BCF são preservadas pelo otimizador.

5 TRABALHOS RELACIONADOS

Duas aplicações são extensivamente estudadas no contexto do problema de identificação de clones: detecção de plágio e detecção de código malicioso. Ambas essas aplicações possuem importância econômica; a segunda delas—detecção de *malware*—possui inclusive importância militar [23].

Detecção de Plágio. O plágio de código-fonte é um problema recorrente na indústria de software e em instituições de ensino [15]. A ofuscação de código dificulta a identificação de código plagiado. Existem, contudo, vários trabalhos que buscam construir detectores de plágio que sejam resistentes ao efeito de ofuscadores de código [1]. Kurtukova et al. [10], por exemplo, discute a eficácia da rede LSTM (do inglês *Long short-term memory*) na construção de detectores de plágio. Os resultados inferidos por Kurtukova et al. mostram que a ofuscação do código diminui entre 21% a 51% a acurácia de detectores de plágio tradicionais. Em termos do arcabouço discutido na Seção 2.3, os experimentos de Kurtukova et al. seriam classificados como instâncias do Jogo-1. Nesse sentido, pudemos encontrar na literatura técnica diversos trabalhos [1, 10, 15] que avaliam o Jogo-0 e o Jogo-1 descritos naquela Seção. Contudo, não encontramos estudos relacionados aos dois demais jogos. É nosso entendimento que essa análise é uma contribuição deste artigo.

Detecção de Código Malicioso. Ofuscadores de código podem ser usados para dificultar a detecção de vírus de computador. Dada a importância econômica de aplicativos móveis, o impacto das ofuscações de código nesse domínio é extensivamente estudado [12, 26, 28, 32]. A título de exemplo, recentemente Ren et al. [18] demonstraram que é possível usar sequências de otimizações para aumentar muito a capacidade de evasão de diferentes tipos de *malware*. Existem, contudo, técnicas que buscam contornar os efeitos de ofuscadores sobre a furtividade de código malicioso. Por exemplo, Preda et al. [17] usam interpretação abstrata para analisar programas a partir de sua semântica, não de sua sintaxe. O sucesso desse tipo de abordagem, contudo, conforme avaliado por Preda et al., é ainda moderado. Outra tentativa de contornar a ofuscação é usar a procedência do código, não sua sintaxe, para identificar *malware*, conforme proposto por Piskozub et al. [16]. Não obstante, a taxa de falsos positivos reportada por Piskozub et al. ainda é muito alta, para que esse tipo de abordagem possa ser considerada prática. Em nosso levantamento bibliográfico, não encontramos trabalhos que usassem um otimizador de código como um mecanismo de normalização (Jogo-3). Acreditamos, portanto, que essa seja uma contribuição original deste trabalho.

6 CONCLUSÃO

Este artigo analisou a eficácia de classificadores estatísticos de programas, quando utilizados sobre código transformados. As transformações de código preservam a semântica dos programas, porém mudam sua sintaxe. Em geral, tais mudanças são suficientes para enganar os classificadores mais populares. Essa conclusão não foi uma surpresa, uma vez que fora recentemente demonstrada anteriormente na literatura [18, 31]. Contudo, pudemos observar que é possível reverter os efeitos de diversos ofuscadores de código simplesmente otimizando o programa. Pudemos constatar também que a otimização de código é tão, ou mais, efetiva que técnicas de ofuscação para confundir classificadores. E, finalmente, mostramos que histogramas de instrução, possivelmente uma das representações de código mais simples que existem, são tão, ou mais, efetivos para habilitar a classificação de código quanto modelos mais complexos, recentemente discutidos na literatura.

ACKNOWLEDGMENTS

Este trabalho foi financiado pelo CNPq (Projeto 406377/2018-9); pela FAPEMIG (Projeto PPM-00333-18) e pela CAPES (Edital CAPES PrInt).

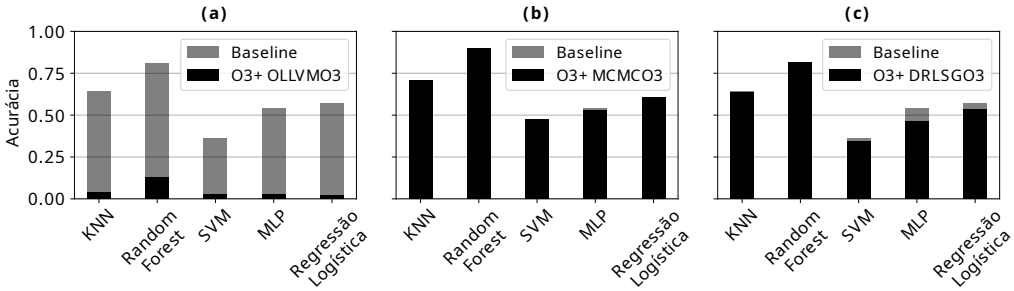

Figure 9: Resultados do Jogo 3, onde o *baseline* é o Jogo 0. Modelos treinados com programas compilados com `-O3` e testados com: (a) programas compilados com as estratégias CFF, SI e BCF do `O-LLVM`, em sequência, mais `-O3`; (b) programas compilados com a estratégia MCMC do `CloneGen` mais `-O3` e (d) programas compilados com a estratégia DRLSG do `CloneGen` mais `-O3`.

REFERENCES

[1] Bander Alsulami, Edwin Dauber, Richard Harang, Spiros Mancoridis, and Rachel Greenstadt. 2017. Source code authorship attribution using long short-term memory based networks. In *European Symposium on Research in Computer Security*. Springer, 65–82.

[2] Alexander Brauckmann, Andrés Goens, Sebastian Ertel, and Jeronimo Castrillon. 2020. Compiler-Based Graph Representations for Deep Learning Models of Code. In *CC*. Association for Computing Machinery, New York, NY, USA, 201–211. https://doi.org/10.1145/3377555.3377894

[3] Leo Breiman. 2001. Random forests. *Machine learning* 45, 1 (2001), 5–32.

[4] Xin Chen, B. Francia, Ming Li, B. McKinnon, and A. Seker. 2006. Shared Information and Program Plagiarism Detection. *IEEE Trans. Inf. Theor.* 50, 7 (sep 2006), 1545–1551. https://doi.org/10.1109/TIT.2004.830793

[5] Corinna Cortes and Vladimir Vapnik. 1995. Support-vector networks. *Machine learning* 20, 3 (1995), 273–297.

[6] Chris Cummins, Bram Wasti, Jiadong Guo, Brandon Cui, Jason Ansel, Sahir Gomez, Somya Jain, Jia Liu, Olivier Teytaud, Benoit Steiner, Yuandong Tian, and Hugh Leather. 2022. CompilerGym: Robust, Performant Compiler Optimization Environments for AI Research. In *CGO*, Jae W. Lee, Sebastian Hack, and Tatiana Shpeisman (Eds.). IEEE, 92–105. https://doi.org/10.1109/CGO53902.2022.9741258

[7] Kai-Bo Duan and S. Sathiya Keerthi. 2005. Which is the Best Multiclass SVM Method? An Empirical Study. In *Proceedings of the 6th International Conference on Multiple Classifier Systems* (Seaside, CA) *(MCS'05)*. Springer-Verlag, Berlin, Heidelberg, 278–285. https://doi.org/10.1007/11494683_28

[8] Evelyn Fix and Joseph L. Hodges. 1952. *Discriminatory Analysis. Nonparametric Discrimination: Consistency Properties.* Technical Report. USAF School of Aviation Medicine.

[9] Pascal Junod, Julien Rinaldini, Johan Wehrli, and Julie Michielin. 2015. Obfuscator-LLVM–software protection for the masses. In *International Workshop on Software Protection*. IEEE, Washington, DC, US, 3–9.

[10] Anna Kurtukova, Aleksandr Romanov, and Alexander Shelupanov. 2020. Source Code Authorship Identification Using Deep Neural Networks. *Symmetry* 12, 12 (2020), 2044.

[11] C. Lattner and V. Adve. 2004. LLVM: a compilation framework for lifelong program analysis amp; transformation. In *International Symposium on Code Generation and Optimization, 2004. CGO 2004.* IEEE, Washington, DC, US, 75–86.

[12] Zhiqiang Li, Jun Sun, Qiben Yan, Witawas Srisa-an, and Yutaka Tsutano. 2019. Obfusifier: Obfuscation-resistant Android malware detection system. In *International Conference on Security and Privacy in Communication Systems*. Springer, 214–234.

[13] Lili Mou, Ge Li, Lu Zhang, Tao Wang, and Zhi Jin. 2016. Convolutional Neural Networks over Tree Structures for Programming Language Processing. In *AAAI*. AAAI Press, Palo Alto, CA, US, 1287–1293.

[14] Mircea Namolaru, Albert Cohen, Grigori Fursin, Ayal Zaks, and Ari Freund. 2010. Practical Aggregation of Semantical Program Properties for Machine Learning Based Optimization. In *CASES*. ACM, New York, NY, USA, 197–206. https://doi.org/10.1145/1878921.1878951

[15] Matija Novak, Mike Joy, and Dragutin Kermek. 2019. Source-code similarity detection and detection tools used in academia: a systematic review. *ACM Transactions on Computing Education (TOCE)* 19, 3 (2019), 1–37.

[16] Michal Piskozub, Fabio De Gaspari, Freddie Barr-Smith, Luigi Mancini, and Ivan Martinovic. 2021. *MalPhase: Fine-Grained Malware Detection Using Network Flow Data.* Association for Computing Machinery, New York, NY, USA, 774–786. https://doi.org/10.1145/3433210.3453101

[17] Mila Dalla Preda, Mihai Christodorescu, Somesh Jha, and Saumya Debray. 2008. A Semantics-Based Approach to Malware Detection. *ACM Trans. Program. Lang. Syst.* 30, 5, Article 25 (2008), 54 pages. https://doi.org/10.1145/1387673.1387674

[18] Xiaolei Ren, Michael Ho, Jiang Ming, Yu Lei, and Li Li. 2021. *Unleashing the Hidden Power of Compiler Optimization on Binary Code Difference: An Empirical Study.* Association for Computing Machinery, New York, NY, USA, 142–157. https://doi.org/10.1145/3453483.3454035

[19] Henry Gordon Rice. 1953. Classes of recursively enumerable sets and their decision problems. *Trans. Amer. Math. Soc.* 74, 2 (1953), 358–366. https://doi.org/10.2307/1990888

[20] Frank Rosenblatt. 1958. The perceptron: a probabilistic model for information storage and organization in the brain. *Psychological review* 65, 6 (1958), 386.

[21] Abdullah Sheneamer, Swarup Roy, and Jugal Kalita. 2018. A detection framework for semantic code clones and obfuscated code. *Expert Systems with Applications* 97 (2018), 405–420.

[22] J. C. Shepherdson and H. E. Sturgis. 1963. Computability of Recursive Functions. *J. ACM* 10, 2 (apr 1963), 217–255. https://doi.org/10.1145/321160.321170

[23] Daniele Ucci, Leonardo Aniello, and Roberto Baldoni. 2019. Survey of machine learning techniques for malware analysis. *Computers & Security* 81 (2019), 123–147.

[24] S. VenkataKeerthy, Rohit Aggarwal, Shalini Jain, Maunendra Sankar Desarkar, Ramakrishna Upadrasta, and Y. N. Srikant. 2020. IR2VEC: LLVM IR Based Scalable Program Embeddings. *ACM Trans. Archit. Code Optim.* 17, 4, Article 32 (2020), 27 pages. https://doi.org/10.1145/3418463

[25] Andrew Walker, Tomas Cerny, and Eungee Song. 2020. Open-Source Tools and Benchmarks for Code-Clone Detection: Past, Present, and Future Trends. *SIGAPP Appl. Comput. Rev.* 19, 4 (jan 2020), 28–39. https://doi.org/10.1145/3381307.3381310

[26] Wei Wang, Mengxue Zhao, and Jigang Wang. 2019. Effective android malware detection with a hybrid model based on deep autoencoder and convolutional neural network. *Journal of Ambient Intelligence and Humanized Computing* 10 (2019). https://doi.org/10.1007/s12652-018-0803-6

[27] Martin White, Michele Tufano, Christopher Vendome, and Denys Poshyvanyk. 2016. Deep Learning Code Fragments for Code Clone Detection. In *ASE*. Association for Computing Machinery, New York, NY, USA, 87–98. https://doi.org/10.1145/2970276.2970326

[28] Yanfang Ye, Tao Li, Donald Adjeroh, and S. Sitharama Iyengar. 2017. A Survey on Malware Detection Using Data Mining Techniques. *ACM Comput. Surv.* 50, 3, Article 41 (jun 2017), 40 pages. https://doi.org/10.1145/3073559

[29] André Felipe Zanella, Anderson Faustino da Silva, and Fernando Magno Quintão. 2020. YACOS: a complete infrastructure to the design and exploration of code optimization sequences. In *Proceedings of the 24th Brazilian Symposium on Context-Oriented Programming and Advanced Modularity*. 56–63.

[30] Muhan Zhang, Zhicheng Cui, Marion Neumann, and Yixin Chen. 2018. An End-to-End Deep Learning Architecture for Graph Classification. In *Proceedings of the AAAI Conference on Artificial Intelligence* (New Orleans, Riverside, USA) *(AAAI '32)*. Association for the Advancement of Artificial Intelligence, New York, NY, USA, 4423–4445.

[31] Weiwei Zhang, Shengjian Guo, Hongyu Zhang, Yulei Sui, Yinxing Xue, and Yun Xu. 2021. Challenging Machine Learning-based Clone Detectors via Semantic-preserving Code Transformations. arXiv:2111.10793 https://arxiv.org/abs/2111.10793

[32] Xiaolu Zhang, Frank Breitinger, Engelbert Luechinger, and Stephen O'Shaughnessy. 2021. Android application forensics: A survey of obfuscation, obfuscation detection and deobfuscation techniques and their impact on investigations. *Forensic Science International: Digital Investigation* 39 (2021), 301285.