FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Sistema de recomendação para proteção de código

Luís Filipe Rodrigues Carvalho

Mestrado Integrado em Engenharia Informática e Computação

Orientador: João Pedro Mendes Moreira

27 de Junho de 2018

Sistema de recomendação para proteção de código

Luís Filipe Rodrigues Carvalho

Mestrado Integrado em Engenharia Informática e Computação

27 de Junho de 2018

Resumo

Com a globalização das aplicações web, a maior parte das organizações começou a preocupar-se com a propriedade intelectual e com a integridade das suas aplicações JavaScript. Com vista a resolver este problema, optaram pela utilização de ferramentas de proteção de código.

A proteção de software tem como principal objetivo tornar os softwares resistentes a ataques, impedindo os atacantes de realizar adulteração de dados, engenharia reversa, ou até mesmo distribuição ilegal de software.

Todavia, este tipo de ferramentas, por vezes, não providencia o resultado desejado ao utilizador, dado que possui diferentes tipos de transformações de código e, ainda, outras sub-opções relacionadas com cada tipo de transformação. Isto leva a que o utilizador se depare com dificuldades na escolha do modelo adequado para a sua aplicação.

Desta forma, no âmbito deste projeto, foi desenvolvida uma ferramenta que permitirá que as organizações usem este tipo de proteções de uma maneira mais eficiente e fiável, diminuindo drasticamente a configuração do sistema de proteção para obtenção de um resultado ótimo para as suas aplicações. Esta ferramenta enquadrada-se na área dos sistemas de recomendação, servindo-se de um conjunto de métricas bem definidas que permitem, não só recomendar transformações, como também avaliar o impacto das mesmas no código original.

A avaliação do modelo de recomendação proposto foi conseguida através de material disponibilizado pela *Jscrambler*. Este dispõe de um grande potencial, uma vez que é constituído por um conjunto variado de estruturas, que permite uma avaliação mais pormenorizada. Durante esta fase de avaliação, foram recolhidas informações que permitiram averiguar o esforço exigido ao atacante, aquando a realização de ataques.

Por fim, os resultados obtidos evidenciam que é necessário encontrar um equilíbrio entre o nível de complexidade e o custo das transformações aplicadas. Isto porque um aumento exagerado no nível de complexidade pode traduzir-se num custo maior, ou seja, a performance - tempo e espaço - do ficheiro original será influenciada negativamente.

Abstract

With the globalisation of web applications, many companies have begun to worry about the intellectual property and integrity of their JavaScript applications. In order to solve this problem, they opted for the use of code protection tools.

The main aim of software protection is to make software resistant to attacks, preventing hackers from tampering with data, reverse engineering, or even the illegal distribution of software.

However, these types of tools do not always provide the user with the desired results, given the different types of code transformations and also other sub-options related to each type of transformation. This leads to the user encountering difficulties in choosing the most suitable model for their application.

Thus, under the scope of this project, a tool has been developed that will enable companies to use this type of protection in a more efficient and reliable manner, drastically reducing the protection system's settings to obtain an optimal result for their applications. This tool is within the framework of recommendation systems, using a set of well-defined metrics that not only enable the recommendation of transformations, but also evaluate their impact on the original code.

The evaluation of the proposed recommendation model was attained through material made available by *Jscrambler*. This has a great potential, since it is constituted by a varied set of structures, which allow for a more detailed evaluation. With the information gathered during this evaluation phase, it was possible to check the effort required on behalf of the hacker, when conducting hacks.

Finally, the results show that it is necessary to find a balance between the level of complexity and the cost of the applied transformations. This is due to the fact that an exaggerated increase in the level of complexity may translate into a higher cost, i.e., the performance - time and space - of the original file will be negatively influenced.

Palavras Chave

- Ofuscação de Código
- Sistemas de Recomendação
- Proteção de Software
- Complexidade Cognitiva
- Complexidade Espacial de Código
- Nível de Ofuscação

Keywords

- Code Obfuscation
- Recommednation Systems
- Software Protection
- Cognitive Complexity
- Code Spatial Complexity
- Obfuscation level

Agradecimentos

Ao Professor João Pedro Mendes Moreira, o meu orientador da Faculdade de Engenharia da Universidade do Porto, pela sua total disponibilidade e auxílio prestado.

À *Jscrambler*, proponente desta dissertação, pelo apoio prestado ao longo da sua realização, em especial ao Vítor Magano, colaborador da empresa, pelos seus presados conselhos e orientação.

Aos meus pais, António Luís Gonçalves Carvalho e Teresa Augusta Freitas Rodrigues Carvalho, pelo apoio incondicional durante todo o meu percurso académico.

Gostaria também de agradecer aos meus amigos, em especial à minha querida Carolina, por me proporcionarem bons momentos de descontração e também, por me incentivarem a fazer sempre mais e melhor.

Luís Filipe Rodrigues Carvalho

"It always seems impossible until it's done "

Nelson Mandela

Conteúdo

Lista de Figuras

LISTA DE FIGURAS

Lista de Tabelas

Abreviaturas e Símbolos

JS	*JavaScript*
AJAX	*Asynchronous JavaScript and XML*
IP	*Internet Protocol*
DBSCAN	*Density-based spatial clustering of applications with noise*
OPTICS	*Ordering points to identify the clustering structure*
SVD	*Singular value decomposition*
MATE	*Man-at-the-End*
CLI	*Interface de Linha de Comandos*

Capítulo 1

Introdução

Este projeto consiste na criação de um sistema de recomendação orientado à proteção de software, com o principal intuito de auxiliar as organizações na escolha de ferramentas que permitam satisfazer as suas necessidades.

Desta forma, este capítulo tem como objetivo a apresentação do tema do presente documento.

Inicialmente, é explorado o contexto em que o problema se insere. Posteriormente, procede-se à identificação da motivação e dos objectivos necessários à resolução do mesmo.

1.1 Contexto

Com a chegada da *"Era digital"*, verificámos que o número de empresas de software está a aumentar cada vez mais, assim como o seu valor [Sul18]. Desta forma, a cotação destas empresas deixa de estar concentrada numa só infraestrutura, ou num objecto físico, passando a ser o software que produzem o seu recurso mais importante.

Consequentemente, o *"software piracy"* e o *"code theft"* são problemas que este tipo de empresas enfrentam no seu dia-a-dia. Assim sendo, é necessário encontrar ferramentas de proteção de código que permitam combater este tipo de dificuldades.

A proteção de software tem como principal objectivo tornar os softwares resistentes a este tipo de ataques, impedindo os atacantes de realizar adulteração de dados, engenharia reversa, ou até mesmo distribuição ilegal de software [CTL97].

Porém, nem sempre é fácil para as organizações escolher o tipo de ferramenta adequada, devido à enorme variedade disponível. Assim sendo, seria bastante útil a criação de um sistema que permitisse a interpretação e análise das preferências e necessidades das organizações.

Com vista à resolução deste problema, surgem os sistemas de recomendação, os quais, através do cruzamento de informação de vários utilizadores e itens, permitem obter previsões sobre os que mais se enquadram com as necessidades do utilizador.

Deste modo, este projeto tem como finalidade apoiar as organizações na manutenção da integridade das suas aplicações, através da recomendação de ferramentas que permitam resolver os problemas que as mesmas enfrentam no seu dia-a-dia. Esta advertência é fruto da criação de um sistema de recomendação orientado à proteção de software.

Por fim, é pertinente referir que este projeto foi proposto pela empresa *Jscrambler*. Esta empresa, através do Vítor Magano, um dos colaboradores da empresa, acompanhou sempre de perto o desenvolvimento da dissertação, prestando auxílio na tomada de decisões e sempre que necessário. A *Jscrambler* possui um produto de nome "*Jscrambler*"que é usado para proteger código-fonte JavaScript e HTML. Este produto tem várias funcionalidades, sendo o seu principal objetivo *blindar* o código-fonte contra possíveis furtos e adulteração.

1.2 Motivação e Objetivos

Atualmente, verifica-se que a maior parte das empresas de software começa a mostrar um maior interesse pela utilização de ferramentas de proteção de código nos seus produtos. Este surge especialmente devido à globalização das aplicações web.

Neste contexto, as empresas necessitam de ferramentas que sejam capazes de manter a propriedade intelectual e a integridade das suas aplicações.

Este tipo de ferramentas, designadas ferramentas de proteção de código, possuem diferentes tipos de transformações associadas e, ainda, outras sub-opções relacionadas com cada tipo de transformação. Isto leva a que o utilizador se depare com dificuldades na escolha do modelo que satisfaça as necessidades da sua aplicação.

De forma a facilitar este processo de seleção, seria útil a existência de uma aplicação que, de acordo com as preferências e necessidades do utilizador, sugerisse um caminho a seguir (opções a selecionar) que permitisse ao mesmo um resultado próximo do ideal, isto é, um sistema de recomendação direcionado à proteção de código. Este sistema, assim como todos os sistemas de recomendação da atualidade, teria como base os métodos de *filtragem colaborativa*, que preveem as preferências de um utilizador, através da análise do comportamento de outros e *filtragem baseada em conteúdos*, um método que compara o perfil do utilizador com as caraterísticas dos serviços/produtos disponíveis [FRK11].

Desta forma, o objetivo inicial a que esta dissertação se propunha era a implementação de um sistema que proporcionasse um output de proteção de código que fosse de encontro às preferências das organizações: custo, potência e resiliência. Desse modo, o sistema permitiria que as organizações usassem esse tipo de ferramentas de uma maneira mais eficiente e fiável, diminuindo significativamente a configuração do sistema de proteção para obtenção de um resultado ótimo para as suas aplicações.

Porém, após a fase de investigação constatou-se que as preferências do utilizador (custo, potência e resiliência) não trariam resultados viáveis, uma vez que a recomendação destas ferramentas depende exclusivamente do programa ou do código a que se destinam. No mesmo sentido, medir a qualidade de ofuscação era um desafio, não existindo uma forma objetiva de o fazer. A partir daí, o

trabalho focou-se na recolha de elementos do programa, através da análise estática ao código, que permitissem identificar ferramentas de proteção adequadas para o mesmo, bem como a avaliação do impacto dessas proteções, isto é, mensurar a qualidade de ofuscação dos resultados obtidos.

1.3 Estrutura do relatório

Para além da introdução, este relatório tem mais quatro capítulos:

- No capítulo 2, é efetuado o levantamento do estado da arte e são apresentados os principais conceitos necessários para a realização desta dissertação, assim como tecnologias e trabalhos relacionados.
- O capítulo 3 apresenta a especificação e detalhe de implementação da ferramenta. Neste capítulo podem-se encontrar expostas as etapas envolvidas no desenvolvimento da solução, as metodologias e tecnologias utilizadas, bem como as principais funcionalidades da ferramenta. Por fim, termina com uma análise geral a todo o trabalho envolvido durante a fase de implementação
- No capítulo 4 é realizada a validação e análise dos resultados. Para isso, são expostas duas situações diferentes que demonstram o comportamento do modelo de recomendação implementado.
- Finalmente, o capítulo 5, expõe as conclusões retiradas na realização do projeto e expressa algum trabalho futuro.

Capítulo 2

Revisão Bibliográfica

Neste capítulo é efetuada uma revisão bibliográfica focada principalmente em sistemas de recomendação e proteção de software. Porém, serão também apresentados os principais conceitos necessários para a realização desta dissertação, assim como as tecnologias e os trabalhos relacionados.

2.1 JavaScript

JavaScript (JS) é uma linguagem de programação interpretada, que surgiu em 1995 por obra de *Brendan Eich* [ECM]. O JS foi originalmente implementado como parte dos navegadores web para que os scripts fossem executados no lado do cliente, ao invés destes scripts passarem pelo servidor. Em novembro de 1996, a Netscape anunciou que tinha submetido esta linguagem para ECMA internacional, como candidato a padrão industrial. Isto resultou numa versão padronizada chamada ECMAScript [ECM]. Desde então, o JS tornou-se numa das linguagens de programação mais usadas no desenvolvimento Web.

Com a chegada do AJAX, anos mais tarde, o JS respondeu às críticas de muitos profissionais que até então se demonstravam descontentes. Como resultado, as frameworks e bibliotecas existentes multiplicaram-se o que permitiu o uso do JavaScript em ambientes para além dos navegadores web.

2.2 Esprima

O *Esprima* é uma ferramenta que permite efetuar análise lexical ou sintática a programas desenvolvidos em JavaScript [cha18]. Apesar de ser possível efetuar estes dois tipos de análise, é importante referir que, no desenvolvimento deste projeto, apenas foi utilizada a análise sintática.

Esta última, também designada por *Parsing*, constitui o processo de análise de uma sequência de entrada (um programa, por exemplo) para determinar a sua estrutura gramatical, segundo uma

determinada gramática formal [RS95]. No caso do *Esprima*, de forma a realizar este tipo de análise, é necessário disponibilizar-lhe uma *string* que represente um programa válido de JavaScript. Posteriormente, este retorna uma árvore sintática ordenada, que descreve a estrutura sintática do programa [cha18].

Desta forma, através da análise à árvore sintática disponibilizada por esta ferramenta, é possível recolher informações acerca das estruturas presentes no programa analisado. Esta árvore é constituída por vários nós, sendo que cada nó pode inserir-se nas seguintes categorias:

- Objetos, que são estruturas que podem ter propriedades associadas.
- Variáveis, estruturas que permitem o armazenamento de informação.
- Funções, são blocos de construção fundamentais em JavaScript. Uma função é um procedimento, isto é, um conjunto de instruções que executa uma tarefa ou calcula um valor.
- Expressões de condição, como *If*'s e *Switch*'s.
- Ciclos, nomeadamente *For*'s, *While*'s e *DoWhile*'s.

Por fim, resta dizer que esta ferramenta foi recomendada pela *Jscrambler* e que, desempenhou um papel bastante importante na construção do Modelo de Recomendação, como veremos mais à frente.

2.3 Proteção de software

Atualmente existe um tipo de ataques contra software, denominados *Man-at-the-end* (MATE) [CDG+11], que tem despertado o interesse de várias áreas ligadas à segurança informática. O seu principal objetivo é comprometer o software disponibilizado pelas organizações. Em pequena escala, os MATE podem violar a privacidade e a integridade de dados visualizados do lado do cliente que em alguns casos podem ser designados de dados críticos como por exemplo registos médicos ou então, numa escala ainda maior, podem criar graves problemas como por exemplo, paralisar uma infraestrutura nacional.

Esta nova forma de comprometer integridade das aplicações tem-se demonstrado mais poderosa que as tradicionais existentes (e.g. ataques *Man-in-The-Middle*) [VRT+16]. Isto deve-se sobretudo ao facto dos ataques terem origem em indivíduos confiáveis dentro da organização [CDG+11]. Desta forma, é-lhes garantido o acesso privilegiado ao software e também a um vasto conjunto de ferramentas, tais como: *"debuggers"*, *"decompilers"* e *analisadores sintáticos* e *dinâmicos*.

Desta forma, a proteção de software tem como principal objectivo tornar os softwares resistentes a este tipo de ataques, impedindo os atacantes de realizar adulteração de dados, engenharia reversa, ou até mesmo distribuição ilegal de software.

2.3.1 Ofuscação de código

A ofuscação de código é uma das ferramentas de proteção de código mais usadas para combater os ataques MATE.

De acordo com *Colberg* et al [CTL97], e tal como está demonstrado na figura 2.1, o objectivo principal da ofuscação é transformar o software, de forma a que a sua compreensão seja mais difícil para o atacante, garantindo a preservação da funcionalidade original do mesmo (ambos recebem *x* e retornam *y* como resultado).

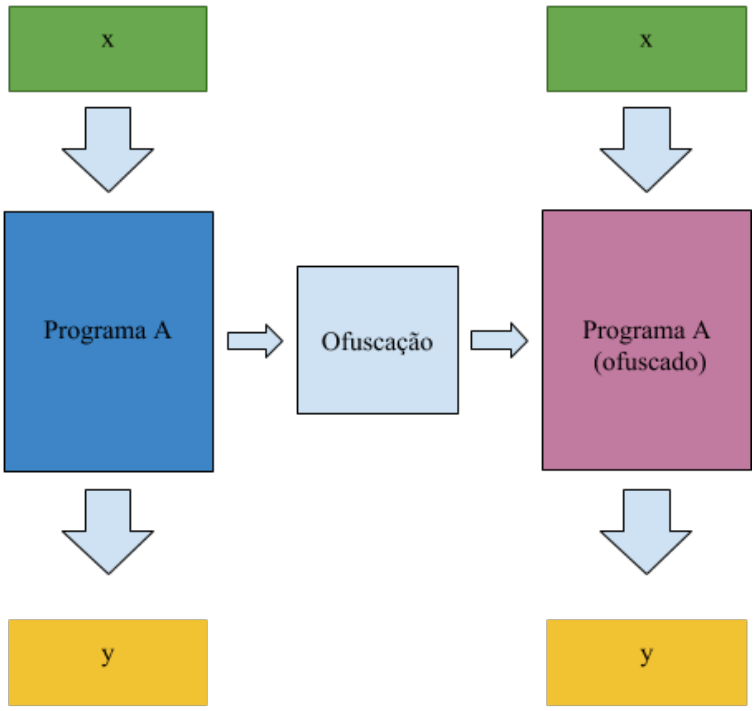

Figura 2.1: Transformação do programa A na sua versão ofuscada

Atualmente, os atacantes encontram-se desencorajados [VRT+16], visto que a utilização desta estratégia torna o ataque um processo mais complexo e, na maior parte das vezes, economicamente desfavorável, dado que é necessário a utilização de ferramentas mais sofisticadas.

A ofuscação de código atinge diferentes alvos no código fonte de um programa. Os alvos mais significativos são apresentados nos três grupos seguintes [CTL97]:

- *Ofuscação da disposição*, que engloba pequenas alterações de disposição, eliminação de informação ou formatação, e que são dispensáveis para o funcionamento do código.
- *Ofuscação de dados*, que engloba transformações ao nível das estruturas de dados.
- *Ofuscação de controlo*, que engloba transformações ao nível do fluxo do código.

O primeiro grupo é constituído pelas técnicas de ofuscação polimórficas, onde as transformações alteram apenas a disposição do código (e.g. remoção de comentários e espaços) e a

forma (e.g. codificação ou encriptação). Os últimos dois grupos, que são os mais representativos, caracterizam-se por alterações profundas no fluxo de execução e nas estruturas de dados. Estas técnicas serão abordadas nas secções seguintes.

2.3.2 Qualidade de ofuscação

A qualidade de ofuscação é avaliada em 4 critérios. Estes critérios são a potência da ofuscação, a resiliência a ataques de inversão automática de ofuscação, o custo que é adicionado pela ofuscação e a capacidade de não deteção da aplicação da ofuscação.

O nível de ofuscação será tanto maior, quanto maior a potência, a resiliência e a não detenção, e menor o custo de ofuscação. Estes critérios serão descritos nas secções seguintes.

Potência

A Potência mede a dificuldade de compreensão de código por um humano. Quanto maior for a complexidade do código, mais difícil será a compreensão do mesmo.

A avaliação da complexidade de código é feita de diferentes formas. Porém, as métricas que mais se destacaram na implementação da solução foram a Complexidade Cognitiva e a Complexidade espacial de código.

A Complexidade cognitiva define o esforço necessário que um ser humano terá de despender para interpretar um determinado excerto de código [WH16]. Esta métrica abandona o uso de modelos matemáticos para avaliar o fluxo de um programa, em detrimento de um conjunto de regras simples que permitem avaliar o grau de perceção do ser humano. O valor de complexidade é calculado de acordo com três regras básicas:

- Ignorar estruturas que permitem que múltiplas instruções sejam reduzidas a uma;
- Incrementar (adicionar um ao valor de complexidade) sempre que existir uma paragem no fluxo linear do código;
- Incrementar quando estruturas de quebra de fluxo estão aninhadas.

Por outro lado, a Complexidade espacial de código tem como principal foco a declaração e utilização de funções. Quanto maior a distância em linhas de código entre a declaração de uma função e a sua utilização, maior será o esforço cognitivo que um ser humano terá de despender para perceber a conexão entre estes módulos [Chh11].

A utilização destas métricas permite obter uma avaliação qualitativa da potência de cada técnica de ofuscação. Porém, é necessário ter em conta que a sua utilização é subjetiva, uma vez que estamos a medir a capacidade de compreensão por humanos.

Resiliência

Ao invés da potência de ofuscação, a resiliência mede a dificuldade de inversão da ofuscação por mecanismos automáticos de inversão.

De acordo com *Colberg* et al [CTL97], a resistência pode ser avaliada medindo o tempo necessário para a construção de um programa de inversão (esforço de programação).

Não deteção

A não deteção é a capacidade de uma técnica de ofuscação passar despercebida, aquando a análise do código ofuscado.

Por vezes, códigos altamente resilientes, ou seja, bastante resistentes a ferramentas de inversão automática, caracterizam-se por transformações bastante exageradas, tornando-as facilmente detetáveis. Assim, é necessário encontrar um equilíbrio entre resiliência e não deteção.

Custo

O custo de ofuscação é o custo adicionado - tempo e espaço - ao código original.

2.3.3 Técnicas de ofuscação: polimórficas

O polimorfismo de código é a capacidade deste sofrer mutações. Este mecanismo altera a forma do código sempre que é criada uma nova cópia, e como em todas técnicas de ofuscação, a funcionalidade do programa original é preservada. [CJ03]

Nas secções seguintes são abordas algumas das técnicas de ofuscação polimórficas mais comuns.

Renomeação de identificadores

Um identificador disponibiliza uma forma de referenciar um elemento. É usual que a atribuição de nomes seja feita de forma a facilitar a identificação do elemento.

Como podemos verificar na figura 2.2, ao substituírem-se esses nomes por carateres sem qualquer significado, é de esperar que um humano tenha mais dificuldades em interpretar o programa.

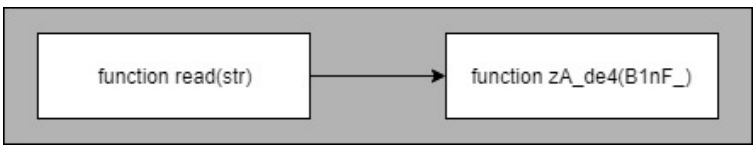

Figura 2.2: Exemplo de mudança de identificadores

Remoção de comentários

A remoção de comentários é essencial, uma vez que estes transmitem informação em linguagem natural sobre a funcionalidade ou propósito do código a que se referem.

Modificação da formatação

A modificação da formatação passa pela eliminação de espaços verticais e horizontais, de forma a dificultar a perceção de código.

Substituição de números

Esta técnica passa pela substituição deste tipo de dados por um sistema de numeração diferente.

Na figura 2.3, podemos verificar a substituição de um número pela sua representação em hexadecimal.

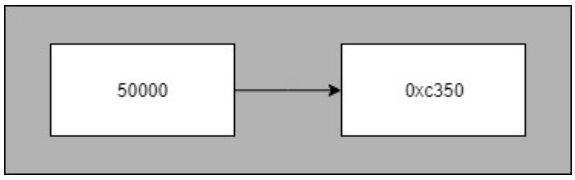

Figura 2.3: Exemplo de substituição de números.

Substituição de caracteres

Esta transformação implica a substituição de cadeias de caracteres, por exemplo, pela representação ASCII em hexadecimal.

Alterar a codificação da informação

Corresponde à formatação de informação atribuída a uma forma de representação simples, para uma expressão matemática que devolva o mesmo valor.

Codificação e Encriptação

Estas técnicas constituem a primeira camada de proteção de código e contribuem também para a alteração da apresentação do código.

2.3.4 Técnicas de ofuscação: metamórficas

À semelhança do polimorfismo, o principal objetivo do metamorfismo é dificultar a compreensão aos atacantes.

Porém, o metamorfismo do código consiste na capacidade de aplicação de transformações no seu fluxo de execução e estruturas de dados. [CJ03]

As principais técnicas que constituem este tipo de ofuscação são apresentadas a seguir.

Inserção de código irrelevante

Esta técnica adiciona ao código original excertos de código que nada contribuem para a funcionalidade do programa, dificultando apenas a perceção do mesmo.

Expansão das condições de terminação de ciclo

Expandir as condições de fim de ciclo significa aumentar a potência da ofuscação sem modificar o resultado da condição original (figura 2.4).

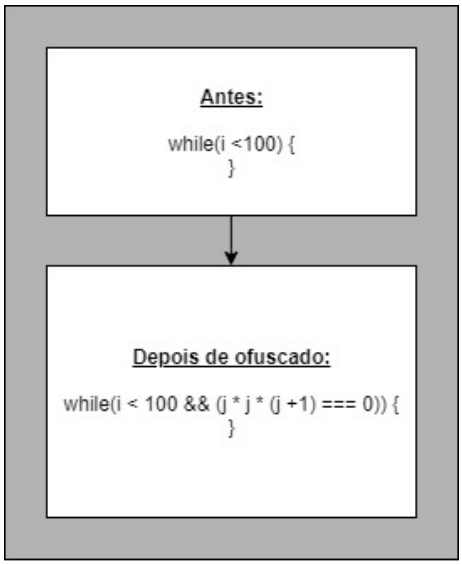

Figura 2.4: Exemplo da expansão de condição de terminação de ciclo de acordo com Colberg et al [CTL97]

Inlining e Outlining de funções

O Inlining de funções consiste na substituição da chamada a uma função pelo seu corpo. Por outro lado, o Outlining é exatamente o oposto.

Fusão e clonagem de funções

Esta técnica consiste na junção de duas ou mais funções numa só, ou na criação de cópias de uma função. O objectivo principal será aumentar o esforço na análise de código.

Transformação de ciclos

Existem algumas transformações possíveis de se aplicar aos ciclos, com o objetivo fundamental de contornar deficiências físicas de um sistema, ou por questões de performance.

Reordenação de elementos

A ordenação de elementos no código é uma boa prática de programação, dado facilitar a compreensão, tanto por quem lê, como por quem o desenvolve. Esta técnica aumenta a resistência à inversão automática de ofuscação.

Adição de operandos redundantes

Técnica que implica a inserção de operandos que apenas dificultam a perceção do código, não alterando o valor final da computação.

Divisão de variáveis

Esta técnica consiste na divisão de uma variável em duas ou mais que, posteriormente combinadas, devolverão o valor real da variável original.

2.3.5 Ofuscação em JavaScript

Como já foi referido em 2.1, o JavaScript é uma linguagem interpretada, isto é, o código não é compilado antes de ser executado. Esta linguagem é utilizada no desenvolvimento de aplicações Web que executam do lado do cliente.

A *Jscrambler*, empresa que disponibiliza um conjunto variado de ferramentas de Ofuscação de código, permite tornar as aplicações JavaScript auto-defensivas e resilientes a ataques, como *"Software Piracy"* e *"Man-in-the-browser"*.

As principais ferramentas de Ofuscação em JavaScript, disponibilizadas pela *Jscrambler*, serão apresentadas a seguir de acordo com as especificações apresentadas em [Jsc18].

String Splitting

Esta técnica de ofuscação divide uma *string* em múltiplas expressões, como podemos verificar pela Figura 2.5.

Boolean To Anything

Um booleano é um tipo de dados que pode ter um de dois valores: verdadeiro ou falso.

Como se pode observar pela Figura 2.6, o objetivo desta transformação é transformar este tipo de dados em expressões booleanas que retornem o mesmo valor, mas que sejam mais difíceis de compreender.

Control-Flow Flattening

Esta transformação, uma das que mais influencia a potência de ofuscação, adiciona estruturas irrelevantes ao código, que apenas contribuem para dificultar a perceção do código por parte dos atacantes.

```
// Original
"hello"

// Obfuscated
var a = "He";
a += "l";
a += "l";
a += "o";
```

Figura 2.5: Exemplo de aplicação da transformação *String Splitting*.

```
// Original
true || false;

// Obfuscated
!!1 || !{}
```

Figura 2.6: Exemplo de aplicação da transformação *Boolean To Anything*.

Dot to Bracket Notation

Como podemos observar pela figura 2.8, esta transformação substitui expressões construídas em *Dot Notation* por expressões *Bracket Notation*.

```
// Original
navigator.plugins.length

// Obfuscated
navigator["plugins"]["length"];
```

Figura 2.7: Exemplo de aplicação da transformação *Dot to Bracket Notation*.

Identifiers Renaming

Esta transformação implica a substituição do nome de identificadores de variáveis ou funções para nomes mais curtos e sem sentido. Mais uma vez, o objetivo é dificultar a compreensão por

parte dos atacantes.

Extend Predicates

Um predicado é uma expressão que pode ser avaliada como verdadeira ou falsa. Esta transformação aumenta o tamanho destes predicados, de forma a dificultar a sua compreensão.

```
// Original
var i = 0;
while (i < 10) {
  i++;
}

// Obfuscated
var i = 0, j = 10;
while (i < 10 && (j * j * (j + 1) % 2 === 0)) {
  i++;
  j = i * j - 1;
}
```

Figura 2.8: Exemplo de aplicação da transformação *Extend Predicates*.

Number To String

À semelhança da transformação *Boolean to Anything*, esta transforma todos os números numa representação em string que retorne o mesmo valor.

O resultado da aplicação desta transformação é exposto na Figura 2.9

```
// Original
user.phoneNumber = 555666777;

// Obfuscated
user.phoneNumber = '555666777'-+[];
```

Figura 2.9: Exemplo de aplicação da transformação *Number To String*.

2.4 Sistemas de recomendação

Nos últimos anos, os sistemas de recomendação (RSs) têm mudado a forma de comunicação entre as aplicações e os seus utilizadores [NT15]. Nos seus primórdios, baseavam-se apenas em

informação demográfica. Atualmente, incorporam informação social acerca dos seus utilizadores e no futuro prevê-se que usem informação proveniente de *"Internet of things"*.

Estes sistemas recolhem informação acerca das preferências dos utilizadores sobre determinados itens (e.g. filmes, músicas, livros). Esta informação pode ser adquirida explicitamente (através da recolha de classificações realizadas pelos utilizadores sobre determinados itens), ou implicitamente (geralmente através da análise do comportamento do utilizador) [NT15].

Os RSs fazem uso de diferentes fonte de informação de forma a proporcionarem aos utilizadores previsões ou recomendações que vão de encontro às suas necessidades e preferências.

Atualmente, os sistemas de recomendação são utilizados com os seguintes objetivos [FRK11]:

- aumentar o número de itens vendidos;
- aumentar a satisfação do utilizador, percebendo as suas necessidades;
- aumentar a fidelidade do utilizador.

De maneira a implementar a sua função principal, o sistema tem de saber antecipar a utilidade de alguns itens ou, pelo menos, comparar a utilidade de alguns deles e decidir os itens que deve recomendar com base nestas comparações. De acordo com *Burke* et al [SFHS07], existem várias técnicas de recomendação, entre as quais se destacam a filtragem baseada em conteúdos, a filtragem colaborativa e a filtragem hibrida. Todas estas técnicas serão abordadas nas secções seguintes.

2.4.1 Técnicas de recomendação

2.4.1.1 Filtragem baseada em conteúdos

Na filtragem baseada em conteúdos, as recomendações têm como base as preferências dos utilizadores, ou seja, são fruto de escolhas que o utilizador teve no passado [SFHS07]. Na figura 2.10 encontra-se uma situação de compra eletrónica. Inicialmente, o utilizador optou pela compra do filme de ficção (1). Desta forma e de acordo com *Burke* et al [SFHS07], o RS, como utiliza filtragem baseada em conteúdos, irá recomendar ao utilizador o filme de ficção (2), uma vez que é similar ao filme de ficção (1) e ainda não foi adquirido pelo utilizar.

Esta técnica, com base nas preferências e nas avaliações do utilizador, irá classificar outros itens desconhecidos através das suas características. Ou seja, é usada uma função que calcula o nível de semelhança entre o perfil do utilizador e os itens a serem recomendados. O resultado consiste nos itens que maximizam a função [NT15].

2.4.1.2 Filtragem colaborativa

A finalidade desta abordagem passa pela recomendação ao utilizador de itens que outros utilizadores, com gostos semelhantes, tenham avaliado anteriormente [NT15].

Na figura 2.11 observamos que os dois utilizadores tiveram preferência pelo mesmo livro. Desta forma, como o RS usa filtragem colaborativa, ao verificar que o utilizador A comprou um

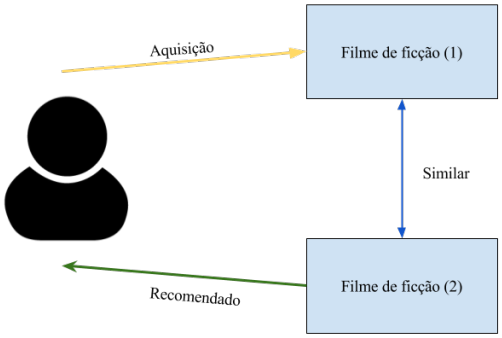

Figura 2.10: Simulação de um recomendador que utiliza filtragem baseada em conteúdos

novo livro, e dado que os utilizadores têm gostos semelhantes, é de esperar que este livro seja recomendado ao utilizador B.

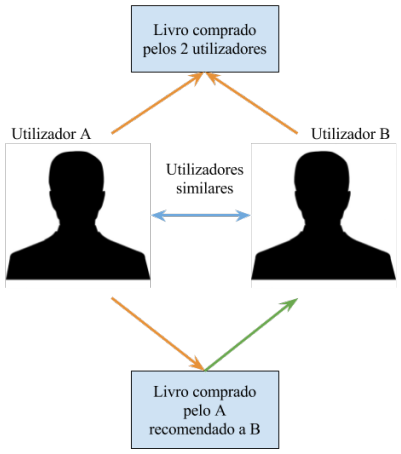

Figura 2.11: Simulação de um recomendador que utiliza filtragem colaborativa

A filtragem colaborativa tem associado a cada utilizador um histórico de classificações que permite ao sistema traçar o nível afinidade entre dois utilizadores diferentes.

Esta técnica contém normalmente uma lista de m utilizadores $\{u_1, u_2, u_3, \ldots, u_n\}$ e uma lista de n itens $\{i_1, i_2, i_3, \ldots, i_n\}$ [NT15]. Cada utilizador u_i tem uma lista de Itens I_{u_i}, que o utilizador avaliou de acordo com as suas preferências ou acerca dos quais as suas preferências foram inferidas através dos seus comportamentos.

2.4.1.3 Filtragem hibrída

A filtragem híbrida baseia-se na combinação de várias técnicas, de forma a explorar ao máximo as vantagens de cada técnica utilizada.

Estes sistemas são geralmente inspirados em algoritmos genéticos, redes neuronais, redes de Bayesian ou clustering [SFHS07].

2.4.2 Métodos de recomendação

Os métodos de recomendação são habitualmente divididos em metodologias que utilizam diretamente os dados (*memory-based*), ou em modelos gerados a partir dos dados (*model-based*) [FRK11].

Estas metodologias serão abordadas nas secções seguintes.

2.4.2.1 Memory-based

As metodologias *memory-based* dão especial importância às avaliações efetuadas pelos utilizadores aos itens. A partir destas avaliações, são usadas métricas de similaridade de forma a obter a distância (ou semelhança), entre dois utilizadores, ou objectos [FRK11].

De entre as várias métricas para medir a similaridade destacam-se a Correlação de Pearson e o Vetor cosseno. A primeira normalmente possui resultados mais fiáveis que a segunda, porém tem um custo computacional maior.

Correlação de Pearson

Esta estatística descritiva, mede o grau de correlação (e a direção dessa correlação - se positiva ou negativa) entre dois utilizadores.

Calcula-se o coeficiente de correlação de Pearson segundo a seguinte equação, onde x_1, x_2, $\ldots$, x_n e y_1, y_2, $\ldots$, y_n são classificações sobre determinados itens efetuadas pelo utilizador X e Y, respetivamente.

$$\rho = \frac{\sum_{i=1}^{n}(x_i-\bar{x})(y_i-\bar{y})}{\sqrt{\sum_{i=1}^{n}(x_i-\bar{x})^2}\cdot\sqrt{\sum_{i=1}^{n}(y_i-\bar{y})^2}} = \frac{\mathrm{cov}(X,Y)}{\sqrt{\mathrm{var}(X)\cdot\mathrm{var}(Y)}} \tag{2.1}$$

Vetor cosseno

Sendo R uma matriz utilizador-item m x n, então a similaridade entre dois itens i e j, é definida pelo cosseno do vetor n dimensional correspondente à coluna i e j da matriz. Esta similaridade pode ser calculada através da fórmula matemática seguinte [FRK11], onde $\bullet$ representa o produto escalar entre dois vetores.

$$w_{ij} = \cos(\vec{i}+\vec{j}) = \frac{\vec{i}\bullet\vec{j}}{||\vec{i}||\cdot||\vec{j}||} \tag{2.2}$$

A filtragem colaborativa corrobora a utilização destas métricas.

Na implementação da solução, a metodologia utilizada enquadra-se neste tipo de recomendação (Memory-based). Desse modo, foram implementadas métricas que permitem avaliar a potência de ofuscação de um determinado excerto de código (2.3.2), como a Complexidade Cognitiva e a Complexidade Espacial de código. Estes detalhes serão analisados nas secções seguintes.

2.4.2.2 Model-based

Nas metodologias *model-based*, são utilizados métodos como *machine learning* e *data mining*, de forma a criar um modelo que se adeque à situação em questão, ou seja, utilizam a informação obtida pelo sistema para criar um modelo que gera as recomendações [FRK11].

Estas metodologias foram criadas com a finalidade de resolver o problema de escalabilidade das aplicações.

Alguns dos algoritmos que estão associados a estas metodologias são os de clustering, dos quais será feita a análise na secção seguinte.

Algoritmos de clustering

Clustering é uma técnica de data mining para agrupar dados segundo o seu grau de semelhança.

Desta forma, um *cluster* é uma coleção (ou grupo) de dados que são semelhantes entre si dentro da mesma coleção. Estes grupos são formados de maneira a maximizar a semelhança entre os elementos do mesmo grupo e minimizar as semelhanças entre os elementos de grupos diferentes.

Os métodos de *clustering* são divididos nas três seguintes categorias: métodos de partição, métodos baseados em densidades e métodos hierárquicos. Os métodos de partição, como por exemplo o das k-médias, têm uma grande vantagem associada à sua fácil implementação. Os métodos baseados em densidades, como o *DBSCAN* e o *OPTICS*, funcionam apenas em ambientes com clusters bastante densos. Os métodos de clustering hierárquicos criam uma decomposição hierárquica da coleção de dados utilizando um determinado critério (e.g. *BIRCH*).

Algoritmos das k-Médias k-Médias é um método de partição, cujo ponto de partida é divisão do conjunto S com N itens em k subgrupos S_j, que são constituídos por itens semelhantes entre si. Cada subgrupo ou cluster contém N_j itens e um centróide γ_j. O centróide para cada cluster corresponde ao ponto onde a soma das distâncias de todos os itens é minimizada. Assim, podemos definir este algoritmo como um processo iterativo que segue a seguinte fórmula matemática [FRK11]:

$$\min E = \sum_{k=1} \sum_{n \in S_j} d(x_n, \gamma_j) \tag{2.3}$$

onde x_n é um vector que representa o item n, γ_j é o centróide do item em S_j e d a distância. Este algoritmo avança até que E não posso atingir valores menores. A implementação básica deste algoritmo é extremamente simples e eficiente [FRK11]. Porém, têm algumas falhas: (1) assume o conhecimento prévio dos dados de forma a escolher o k apropriado ; (2) pode criar um cluster vazio ; e (3) apresenta problemas quando os clusters possuem diferentes tamanhos e densidades.

Algoritmos dos fatores latentes Os algoritmos de fatores latentes, como fatorização de matrizes (SVD), constituem uma abordagem alternativa, uma vez que agrupam no mesmo espaço utilizadores e itens [FRK11].

Factorização de matrizes (SVD) Os modelos de fatorização, como já foi referido, agrupam num espaço de dimensionalidade f utilizadores e itens. Através de dados inferidos do espaço latente, é possível classificar utilizadores e itens [FRK11]. Por exemplo, se o produto a analisar for um filme, os fatores medem dimensões óbvias como categoria do filme, orientação para crianças ou quantidade de acção.

Esta técnica é muito comum para produzir aproximações de baixo grau.

2.4.3 Avaliação de sistemas de recomendação

Atualmente os sistemas de recomendação são bastantes populares na área da investigação, onde várias técnicas têm sido sugeridas e algumas melhoradas [FRK11].

Apesar de existirem várias técnicas, é necessário escolher um algoritmo que vá de encontro às necessidades das organizações. Estas possuem várias características que podem afetar a experiência do utilizador, como robustez, precisão e escalabilidade. Assim, é necessário efectuar uma análise detalhada aquando da escolha da técnica a implementar.

De acordo com *Guy Shani* e *Asela Gunawardana* et al [FRK11], é importante seguir três etapas na comparação de sistemas de recomendação:

1. *Formulação de uma hipótese*. Antes de iniciar a comparação, é necessário formular uma hipótese que seja concisa e restritiva, bem como traçar uma experiência que teste a mesma. Por exemplo, uma hipótese pode constatar que o algoritmo *A* é melhor a prever as avaliações dos utilizadores sobre determinados itens do que o algoritmo *B*. Desta forma, a hipótese deve testar a precisão das recomendações, e não outros fatores.

2. *Controlo de variáveis*. Durante a fase de teste, é estritamente necessário que os algoritmos utilizem o mesmo conjunto de dados ou amostras diferentes, mas que pertençam ao mesmo conjunto de dados. Por exemplo, supondo que queremos prever a precisão das recomendações do algoritmo *A* e *B*, se utilizarem conjuntos de dados diferentes, não podemos concluir qual o algoritmo que se comportou melhor. Isto porque, não sabemos se resultou do algoritmo utilizado ou de um conjunto de dados sobre o qual era mais fácil efectuar a previsão.

3. *Generalização do poder.* Na fase das conclusões é importante obter resultados que permitam generalizar para além das experiências efetuadas. Para aumentarmos a probabilidade de generalização, é importante efetuar experiências com vários tipos de dados e aplicações, de modo a podermos generalizar os resultados e não nos restringirmos aos dados utilizados na fase de teste.

2.5 Conclusões

Embora os sistemas de recomendação tenham evoluído ao longo do últimos tempos, na área da proteção de software ainda se encontram num estado embrionário. Isto deve-se essencialmente ao facto de não existir uma metodologia objetiva que permita avaliar a qualidade de ofuscação.

Apesar de existirem métricas, como a Complexidade Cognitiva e a Complexidade espacial de código, que permitam obter uma avaliação qualitativa do nível de ofuscação, é necessário ter em conta que a sua utilização é subjetiva, dado estarmos a medir a capacidade de compreensão por humanos.

Desta forma, o facto de não existir uma metodologia que permita classificar de forma objetiva o impacto das ferramentas de proteção num determinado software, constituiu um entrave à criação de um modelo baseado em *machine learning* ou *data mining* que se adequasse ao problema proposto.

Como resultado, a solução proposta, que será descrita no capítulo seguinte, passará pela investigação e implementação de metodologias que possam constituir propriedades ativas no modelo de recomendação.

Capítulo 3

Especificação e implementação da ferramenta

Neste capítulo é apresentada a ferramenta desenvolvida no âmbito desta dissertação.

Inicialmente são apresentados os obstáculos que se opuseram à criação de um modelo de recomendação. De seguida, é apresentada a solução ao problema proposto. Posteriormente, são abordadas as etapas envolvidas no processo de implementação e expostas as tecnologias utilizadas no desenvolvimento da ferramenta. Por último, o capítulo termina com conclusões acerca da ferramenta desenvolvida.

3.1 Oposições ao modelo de recomendação

Nos último anos, os sistemas de recomendação têm mudado a forma de comunicação entre as aplicações e os seus utilizadores. Inicialmente baseavam-se apenas em informação demográfica. Atualmente incorporam informação social acerca dos seus utilizadores e, no futuro, prevê-se que usem informação proveniente de *"Internet of Things"*.

Embora estes sistemas tenham evoluído ao longo dos últimos tempos, é importante referir que na área da proteção de software o seu desenvolvimento é praticamente inoperante. Isto deve-se essencialmente ao facto de os sistemas de recomendação atuais terem como principal foco os utilizadores e as suas necessidades [FRK11], o que não se enquadra no âmbito da proteção de software.

Com isto, e de forma a solucionar este problema, tornou-se necessário um modelo que abandonasse as preferências e necessidades dos utilizadores, e que incluísse métricas objetivas relacionadas com a ofuscação de código.

Como consequência, o desenvolvimento desta dissertação teve como principal obstáculo a criação de um modelo que conseguisse traduzir objetivamente todas as variáveis relacionadas com a ofuscação, isto é, que avaliasse a potência, resiliência, custo e não deteção. A subjetividade

inerente às métricas de potência e não deteção contribuiu para o insucesso deste modelo, bem como a dificuldade em definir objetivamente um equilíbrio entre elas.

Em conclusão, a principal adversidade que se opôs à criação do modelo inicialmente proposto foi a falta de métricas que permitissem interpretar o nível de ofuscação objetivamente.

3.2 Modelo de Recomendação

Após a fase de análise das soluções disponíveis e da exploração das dificuldades expostas na secção 3.1, optou-se por um modelo de recomendação em que a potência, através da Complexidade Cognitiva e a Complexidade espacial de código, representasse a variante principal.

A potência, uma das métricas que permite avaliar a qualidade de ofuscação, mede a dificuldade de compreensão de código por um humano. Esta métrica, apesar de ter um carácter subjetivo, é a única que está diretamente relacionada com a dificuldade oferecida aos atacantes, aquando da realização de ataques como adulteração de dados, engenharia reversa, ou até mesmo distribuição ilegal de software.

O custo e a não deteção, que também fazem parte dos critérios para avaliar a qualidade de ofuscação, não foram incluídos neste modelo, uma vez que não foi encontrada uma forma objetiva de traduzir estas métricas num valor que permitisse avaliar a qualidade de ofuscação.

À semelhança do custo e não deteção, a resiliência também foi descartada do modelo de recomendação. Apesar de *Colberg* et al [CTL97] sugerir que esta pode ser avaliada medindo o tempo necessário para a construção de um programa de inversão, após adequada ponderação considerou-se que o tempo requerido para a implementação de uma métrica com esta complexidade não iria trazer vantagens.

Assim sendo, o modelo proposto, em conformidade com os sistema de recomendação baseados em memória (2.4.2.1), tem como principal foco a análise da similaridade entre as estruturas presentes no ficheiro que se pretende ofuscar e as estruturas alvo de cada uma das ferramentas de ofuscação. Isto tudo tem como intuito encontrar as ferramentas de proteção que mais se enquadram com o ficheiro inserido no sistema.

Por fim, é importante referir, que a construção do Modelo de Recomendação dividiu-se em diferentes etapas, como veremos em 3.5.

3.3 Metodologias

O *Scrum* é uma técnica de desenvolvimento ágil [Scr], que permite a gestão e planeamento de projetos de software. Segundo esta técnica, os projetos são divididos em ciclos denominados *Sprints*, que possibilitam uma avaliação dinâmica do trabalho.

Esta metodologia foi aplicada no desenvolvimento deste projeto, proporcionando um desenvolvimento iterativo do mesmo. De referir que o desenvolvimento deste projeto foi constituído por 5 *Sprints*, cada um com duração de 2 semanas.

3.4 Ferramentas e tecnologias adotadas

Na elaboração da solução foram utilizadas tecnologias como *Jscrambler*, *Node.js*, *Commander*, *Esprima* e *GitLab*. A utilização destas teve como principal objetivo o auxílio no desenvolvimento da solução proposta.

A ferramenta *Jscrambler* colaborou na última fase deste projeto, dado ter permitido a aplicação das técnicas de ofuscação recomendadas pelo sistema.

O *Node.js* é um interpretador de código JavaScript que funciona do lado do servidor [Fou], sendo o seu principal objetivo permitir a criação de aplicações de alta escalabilidade. No contexto deste problema, esta ferramenta participou no desenvolvimento integral da aplicação.

No mesmo sentido, o *Esprima*, que corresponde a um analisador sintático de código JavaScript [cha18], permitiu a recolha de estruturas através da árvore sintática gerada.

O *Commander*, que é um pacote de *Node.js*, tornou a criação de interface de linha de comandos (CLI) mais simples.

Por fim, e de forma a facilitar o controlo de versões, foi utilizada a plataforma GitLab [git].

3.5 Etapas do desenvolvimento

De forma a facilitar o planeamento deste projeto, foram distinguidas quatro etapas, as quais serão descritas nas secções seguintes. De notar que cada uma destas etapas corresponde a um *Sprint*. O quinto (e último) *Sprint* do ciclo de desenvolvimento corresponde à escrita da dissertação.

3.5.1 Investigação

A primeira etapa da fase de desenvolvimento teve como principal foco a exploração de ferramentas que auxiliassem na recolha de estruturas, através da análise estática de código. Como resultado, optou-se pelo *Esprima*, dado este fazer parte integrante do conjunto de ferramentas utilizadas pela *Jscrambler*.

Posteriormente, priorizou-se a investigação de métricas que permitissem avaliar o nível de ofuscação de um determinado programa ou excerto código. Daqui surgiram as seguintes métricas:

- Complexidade Ciclomática, que de acordo com *McCabe* et al [McC76] é uma métrica de software usada para indicar a complexidade de um programa. Esta mede a quantidade de caminhos de execução independentes a partir de um código fonte.

- Complexidade Cognitiva, definida por *Jintender* et al [Chh11] como esforço necessário que um ser humano terá de despender para interpretar um determinado programa .

- Complexidade Espacial de código, que têm como principal foco medir o impacto da declaração e utilização de funções na complexidade do ficheiro [Chh11].

Porém, apenas a Complexidade Cognitiva foi utilizada na implementação da solução final.

No caso da Complexidade Ciclomática, verificou-se que os valores obtidos nem sempre traduziam de forma mais correta o esforço necessário para um humano interpretar um dado excerto de código, o que levou ao seu abandono. Como podemos observar pela Figura 3.1, o método *sumOfPrimes* é intuitivamente mais difícil de interpretar do que o método *getWords*.Porém, o valor de Complexidade Ciclomática é igual nos dois métodos, o que evidencia que, por vezes, é difícil lidar com resultados de ofuscação que não alterem a Complexidade Ciclomática do código mas que, por outro lado, providenciem níveis de perceção para o atacante bastante diferentes.

No mesmo sentido, a Complexidade Espacial de código não foi considerada. Apesar de inicialmente ter feito parte integrante do Modelo de Recomendação, verificou-se que a sua contribuição para medir o nível de ofuscação era praticamente nula e, por vezes, até contribuía para a existência de falhas na recomendação de ferramentas.

```
int sumOfPrimes(int max) {                  // +1
  int total = 0;
  OUT: for (int i = 1; i <= max; ++i) {     // +1
    for (int j = 2; j < i; ++j) {           // +1
      if (i % j == 0) {                     // +1
        continue OUT;
      }
    }
    total += i;
  }
  return total;
}                  // Cyclomatic Complexity 4
```

```
String getWords(int number) { // +1
    switch (number) {
      case 1:                 // +1
        return "one";
      case 2:                 // +1
        return "a couple";
      case 3:                 // +1
        return "a few";
      default:
        return "lots";
    }
  }      // Cyclomatic Complexity 4
```

Figura 3.1: Exemplo retirado de *G. Ann Campbell* et al [Cam16].

A comparação de código, assim como a análise de ferramentas de recomendação em *Node.js*, fizeram também parte desta etapa. Na comparação de código foram investigadas ferramentas que permitissem obter o nível de similaridade entre dois pedaços de código. Porém, os resultados não foram favoráveis. Na parte das ferramentas de recomendação, investigou-se o *Brain.js*[1] e *Likely.js*[2], bibliotecas de *JavaScript* orientadas à área da recomendação. Em ambos os casos a incompatibilidade com o modelo proposto não permitiu o avanço para a fase seguinte.

No final desta etapa foi discutido o valor acrescentado à solução por cada uma das tarefas realizadas, sendo que é de destacar a relevância da Complexidade Cognitiva no Modelo de Recomendação proposto.

3.5.2 Recolha de estruturas

Esta etapa passou pela construção de uma interface de linha de comandos (CLI) que permitisse recolher estruturas de um ficheiro *JavaScript* através da análise estática ao código.

Com o auxílio do *Esprima* foi possível obter a árvore sintática (AST), correspondente ao ficheiro inserido no sistema. Ao percorrer cada nó da AST recolheu-se a seguinte informação:

[1]Ferramenta orientada aos sistemas de recomendação que utiliza redes neuronais.
[2]Ferramenta que tem principal foco a filtragem colaborativa.

- *Tipo de dados*, como por exemplo, analisar o tipo de uma variável que pode ser booleano, número, palavra, nulo ou indefinido.

- *Declarações*, ou seja, declaração e utilização, tanto de variáveis como de funções.

- *Expressões*, que envolve identificar estruturas como *For*'s, *If*'s, *While*'s, entre outros.

Assim sendo, a principal funcionalidade desta CLI envolve a análise iterativa à AST e, por fim, o retorno de um ficheiro *JSON* com a informação acerca do tipo de dados e estruturas descobertos.

Contudo, existe também a possibilidade de realizar análise em bloco à árvore gerada. Uma declaração em bloco, estrutura delimitada por um par de chavetas, é utilizada para agrupar zero ou mais declarações [Doc18a]. Desta forma, o resultado será um ficheiro *JSON* com a informação detalhada acerca de cada bloco e das estruturas que o integram.

3.5.3 Complexidade Cognitiva (CC)

À semelhança da CLI descrita em 3.5.2, esta etapa passou também pela criação de uma interface de linha de comandos, com o auxílio das ferramentas *Esprima* e *Commander*.

A Complexidade Cognitiva é uma métrica que define o esforço necessário que um ser humano terá de despender para interpretar um determinado excerto de código. A sua implementação seguiu a solução proposta por *G. Ann Campbell* et al [Cam16], ou seja, adotou um conjunto de regras simples que permitissem avaliar o grau de perceção por um humano.

Por conseguinte, foram implementadas as seguintes especificações:

Ignorar estruturas que tornem o código mais percetível

Implica a remoção de estruturas que tornem o código mais acessível ao ser humano, ou seja, de mais fácil perceção.

Como podemos ver na figura 3.2, muitas das vezes esta regra aplica-se a estruturas que permitam converter múltiplas linhas de código em apenas uma.

Por conseguinte, o valor da CC não sofre qualquer tipo de incremento em estruturas deste tipo.

```
MyObj myObj = null;
if (a != null) {
  myObj = a.myObj;
}
```

```
MyObj myObj = a?.myObj;
```

Figura 3.2: Exemplo de utilização do operador *null-coasceling* retirado de *G. Ann Campbell* [Cam16].

Incrementar em quebras no fluxo de código

Outro princípio da Complexidade Cognitiva é o de que estruturas que provocam quebras no fluxo linear do código exijam um esforço maior por parte do indivíduos que procuram perceber o funcionamento de determinado programa.

Contribuem para este incremento estruturas como:

- Ciclos, ou seja, *For*'s, *DoWhile*'s, *While*'s, entre outros.
- Condições, isto é, *If*'s e *ElseIf*'s.

É também importante referir que a Complexidade Cognitiva incrementa uma unidade por cada estrutura presente no programa.

Cacthes e *Switches*

Estes tipos de estruturas possui um caráter especial.

Em relação aos *Cacthes*, estruturas que contêm código para a manipulação de erros, é importante referir que têm um comportamento semelhante às condições (*If*'s) [Doc18c]. Desta forma, o valor de complexidade é incrementado em uma unidade por cada estrutura deste tipo, não importando o número de exceções capturadas.

Por outro lado, os *Switches*, em conformidade com as cadeias *IfElse*, permitem a realização de testes condicionais [Doc18b]. Porém, como estas estruturas comparam explicitamente uma variável a um conjunto de valores literais, exigem menos esforço. Desta forma, a Complexidade Cognitiva é apenas incrementada de uma unidade, independentemente do número de comparações efetuadas.

Incrementar para estruturas aninhadas

Esta regra parte do princípio de que no papel de um indivíduo é bastante mais fácil interpretar um conjunto de *If*'s e *For*'s dispostos de forma linear, do que estas mesmas estruturas aninhadas sucessivamente.

Por conseguinte, e como podemos observar na figura 3.3, sempre que existem estruturas aninhadas é associado uma nova variante que representa o nível de *nesting*. No final, este valor é adicionado à Complexidade Cognitiva.

3.5.4 Modelo de recomendação

Este modelo identifica a fase final envolvida na implementação da solução proposta. O seu principal objetivo passa pela recomendação de transformações de código de acordo com as características do ficheiro (do tipo JavaScript) inserido no sistema.

Esta estratégia de recomendação segue a filosofia dos métodos de recomendação *Memory-based*, uma vez que utiliza um modelo matemático que permite avaliar o grau de compatibilidade de uma determinada técnica de ofuscação com o ficheiro inserido.

```
void myMethod () {
  try {
    if (condition1) {                    // +1
      for (int i = 0; i < 10; i++) {     // +2 (nesting=1)
        while (condition2) { … }         // +3 (nesting=2)
      }
    }
  } catch (ExcepType1 | ExcepType2 e) { // +1
    if (condition2) { … }                // +2 (nesting=1)
  }
}                                        // Cognitive Complexity 9
```

Figura 3.3: Complexidade Cognitiva em estruturas aninhadas. Exemplo retirado de *G. Ann Campbell* [Cam16].

A primeira fase que constitui este modelo envolve a recomendação de ferramentas de proteção de código adequadas ao input do utilizador. Esta é auxiliada pela CLI, implementada em 3.5.2, baseando-se na quantidade de estruturas de cada tipo presente no ficheiro inserido pelo utilizador. Deste modo, para cada tipo de dados é calculada a sua Frequência Relativa (F_r), que é resultado do quociente entre o número do tipo de dados correspondente (F_i) e o número total de dados (n).

$$F_r = \frac{F_i}{n} \tag{3.1}$$

Por conseguinte, e como podemos observar na Tabela 3.1, para estruturas com valores iguais ou superiores à Frequência Relativa correspondente, é recomendada não só a transformação que lhe está associada, como também a sinergia. Os valores apresentados para a Frequência Relativa de cada tipo de dados foram definidos segundo a importância de cada um. Esta importância identifica elementos com valor de Frequência Relativa reduzido, uma vez que estão mais suscetíveis de serem esmiuçados pelos atacantes, visto que na maior parte das vezes integram grande parte da lógica do programa, como é o caso dos *Predicados*.

Tipo de dados	**Frequência Relativa (%)**	**Transformação recomendada**	**Sinergia**
Boolenaos	25	*Boolean To Anything*	*String Splitting*
Números	25	*Number To String*	*String Splitting*
String's	25	*String Splitting*	*String Concealing*
Variáveis	25	*Variable Grouping*	-
Identificadores	30	*Identifiers Renaming*	-
Predicados	15	*Extend Predicates*	*Number To String*
Funções	25	*Function Outlining + Opaque Functions*	-
Dot notation	15	*Dot To Bracket Notation*	*String Splitting*
Comma Operator	15	*Comma Operator Unfolding*	-

Tabela 3.1: Especificações do modelo de recomendação.

É importante referir que o sistema apenas indica quais são as transformações que se adequam mais ao input fornecido pelo utilizador, sendo este quem escolhe quais aplicar.

A segunda fase envolve avaliar o impacto causado pelas transformações selecionadas pelo utilizador no código original. Para isso é utilizada a métrica descrita em 3.5.3, que permite observar a variação de Complexidade causada pela aplicação das transformações, entre o ficheiro de código original e o resultado da ofuscação. Desta avaliação resulta o rácio entre a complexidade cognitiva inicial e a complexidade cognitiva final (depois de aplicadas as transformações), podendo ocorrer o seguinte:

- Sendo um valor compreendido entre 10 e 15, trata-se de um valor ideal. De referir que este intervalo de valores foi resultado de uma adequada ponderação e rigorosa análise sobre várias amostras de código, através da análise da influência criada pelo o aumento da Complexidade Cognitiva no tamanho em termos de espaço adicionado ao ficheiro original.
- Dado o valor atingido não ser ótimo, são avaliadas todas as transformações que não foram utilizadas, aquelas que permitem atingir o valor mais próximo do ideal.

3.6 Conclusões

Neste capítulo ficaram expostas todas as etapas envolvidas na implementação da solução proposta, bem como as metodologias e técnicas adotadas.

No final podemos concluir, que apesar dos obstáculos enfrentados ao longo da implementação do Modelo de Recomendação, como foi explicado na secção 3.1, de uma forma geral o objetivo principal foi cumprido. Porém, existe grande espaço para melhorar a performance da solução desenvolvida, como será discutido nas secções seguintes.

Por outro lado, é de referir que a estratégia de *Scrum* contribuiu bastante para o sucesso desta etapa, uma vez que possibilitou um planeamento mais eficiente da mesma. Da mesma forma, os resultados obtidos na fase de investigação, bem como as tecnologias utilizadas, desempenharam um papel fundamental nesta fase.

Por fim, é de referir que na secção seguinte será feita uma análise cuidadosa dos resultados obtidos.

Capítulo 4

Validação e Resultados

Neste capítulo são validados os resultados obtidos pelo modelo de recomendação. Para isso, é utilizado um ficheiro de teste, disponibilizado pela *Jscrambler*, de forma a avaliar os efeitos das ferramentas de proteção de software recomendadas. A sua escolha teve origem no facto deste apresentar, embora em alguns casos em quantidades pouco significativas, praticamente todos os elementos que são alvo da primeira fase do modelo de recomendação, isto é, da recolha de estruturas. Este ficheiro está disponível no Anexo A.

A primeira fase de validação implica a compreensão da influência do levamento de estruturas na recomendação de ferramentas de proteção. Na Figura 4.1 são apresentas as estruturas recolhidas no ficheiro de teste. É importante referir que os valores apresentados correspondem à quantidade de cada um dos elementos descobertos neste ficheiro.

Seguidamente, e com o intuito de avaliar o impacto das transformações recomendadas, é medido o valor de Complexidade Cognitiva nas diferentes situações descritas.

Por fim, é realizada uma apreciação geral aos resultados obtidos.

4.1 Análise à recomendação de ferramentas de proteção de software

A primeira fase envolvida neste modelo baseia-se na recomendação de ferramentas de proteção, através do tipo e quantidade de estruturas presentes no ficheiro inserido pelo utilizador.

No âmbito deste Capítulo, e com vista à validação dos resultados, a informação necessária à consumação desta fase encontra-se exposta na Figure 4.1. Consequentemente, e de acordo com os critérios relativos a esta etapa avançados no Capítulo 3, foram recomendadas as seguintes transformações:

- *Number to String*;
- *Dot to Bracket Notation*;
- *Identifiers Renaming*;

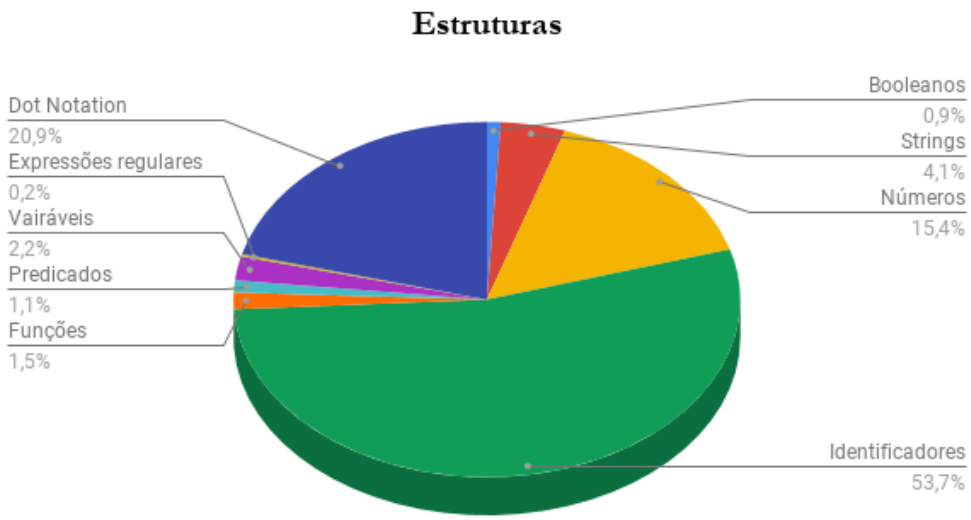

Figura 4.1: Estruturas presentes no Ficheiro de Teste.

- *Control Flow Flattening*.

Porém, e de forma a aperfeiçoar os resultados da ofuscação são também mencionadas as seguintes sinergias:

- *String Splitting*;
- *String Concealing*;
- *Duplicate Literals Removal*.

Analisando os resultados obtidos foi possível depreender que as transformações recomendadas dão resposta direta às necessidades adjacentes ao ficheiro de teste utilizado. Isto advém do facto de estas se direcionarem às estruturas que se encontram em maior quantidade, como demonstrado na Figura 4.1. Estas estruturas pelo facto de se encontrarem em maior abundância, como é o caso dos Identificadores e da *Dot Notation*, constituem um alvo fácil para os atacantes. Isto porque, no caso do identificadores, é de notar que têm como função dar nome a variáveis e funções e, por vezes, esses nomes são bastante intuitivos relativamente ao elemento/função que identificam, facilitando a tarefa dos atacantes.

Desta forma, ao protegermos este tipo de estruturas, estamos a exigir um esforço maior ao atacantes durante a realização de ataques, como adulteração de dados, engenharia reversa ou, até mesmo, distribuição ilegal de software.

No final desta fase, é de expectar que o conjunto de ferramentas recomendadas possua um grande potencial para combater as limitações do ficheiro em análise. Porém, é indispensável

avaliar o impacto destas transformações de uma forma mais precisa, ou seja, medindo o nível de ofuscação, como será demonstrado na secção seguinte.

4.2 Avaliação do impacto da aplicação das ferramentas recomendadas

De forma a avaliar o impacto das ferramentas recomendadas foram realizadas duas experiências. Na primeira experiência são usadas as transformações e respetivas sinergias expostas na secção 4.1. Por outro lado, na segunda experiência é feita uma seleção de um grupo restrito de entre transformações e sinergias recomendadas através do Modelo de Recomendação.

Na primeira experiência, como podemos observar pela Figura 4.2, foi obtido um valor para a Complexidade Cognitiva de 335,7 no ficheiro ofuscado. Este resultado evidencia um aumento bastante significativo no valor desta grandeza, comparativamente ficheiro original (9,72). No mesmo sentido, ao analisarmos o gráfico ilustrado na Figura 4.3, concluímos, também, que existiu um aumento bastante significativo, porém, neste caso desvantajoso, dado que o tamanho ocupado pelo ficheiro ofuscado aumentou 50 unidades relativamente ao ficheiro original.

Desta forma, apesar de um valor elevado de Complexidade Cognitiva ser favorável, é necessário encontrar um equilíbrio entre a CC e o custo de ofuscação, de forma a que esta aumente significativamente o esforço exigido aos atacantes, mas que, por outro lado, não provoque um aumento excessivo no tamanho do ficheiro ofuscado.

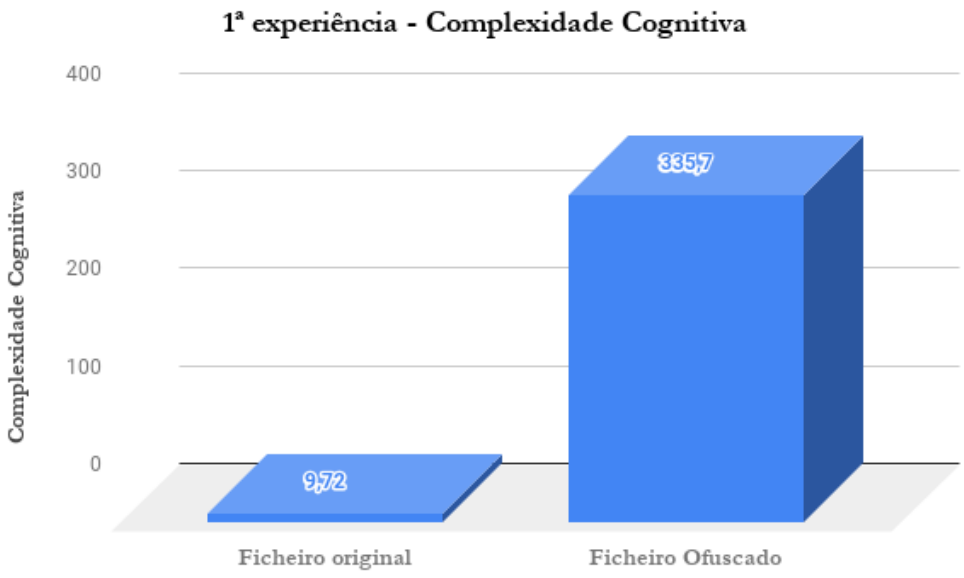

Figura 4.2: Alteração provocada na Complexidade Cognitiva do Ficheiro de teste após a Experiência 1.

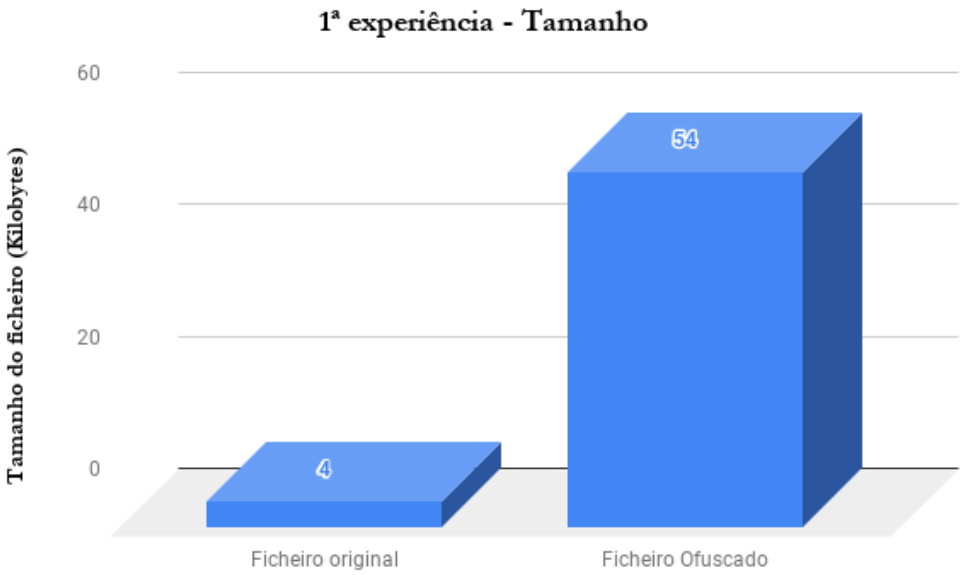

Figura 4.3: Variação no Tamanho do Ficheiro de teste após a Experiência 1.

No caso da segunda experiência, de forma a conseguir encontrar um equilíbrio entre a Complexidade Cognitiva e o Tamanho do ficheiro ofuscado, procedeu-se à análise do rácio de ofuscação (quociente entre a CC do ficheiro ofuscado e a CC do ficheiro original). Este foi exposto em 3.5.4, e define que, para valores compreendidos no intervalo 10 a 15 é possível obter um nível de ofuscação razoável, ou seja, a ofuscação é satisfatória quanto à Complexidade Cognitiva e não implica um aumento excessivo no Tamanho do Ficheiro ofuscado.

Por conseguinte, e de forma a obter um resultado ótimo, foram selecionadas as seguintes transformações do conjunto de transformações recomendadas pelo Modelo de Recomendação:

- *Number to String*;
- *Dot to Bracket Notation*;
- *Control Flow Flattening*;
- *String Splitting*;
- *String Concealing*.

É importante referir, que este conjunto de transformações é apenas um exemplo. Sendo que também seria possível a obtenção de um resultado ótimo, por exemplo, adicionando a transformação *Identifiers Renaming*.

Através da análise da Figura 4.4, foi possível calcular o rácio de ofuscação obtendo-se um valor de aproximadamente 10.5 (valor contido no intervalo definido com ótimo). Da mesma forma verifica-se que, ao contrário da Experiência 1, os valores obtidos no que toca à Complexidade

Cognitiva são significativamente mais baixos. Todavia, e de acordo com o gráfico ilustrado na Figura 4.5, o ficheiro ofuscado cresceu apenas 15 unidades em relação ao ficheiro original, um valor bastante inferior ao da Experiência 1.

Assim, é possível concluir que apesar do valor de CC ser inferior ao obtido na Experiência 1, desta vez o custo de ofuscação foi menor, isto é, existiu um equilíbrio entre Complexidade do ficheiro e o seu tamanho.

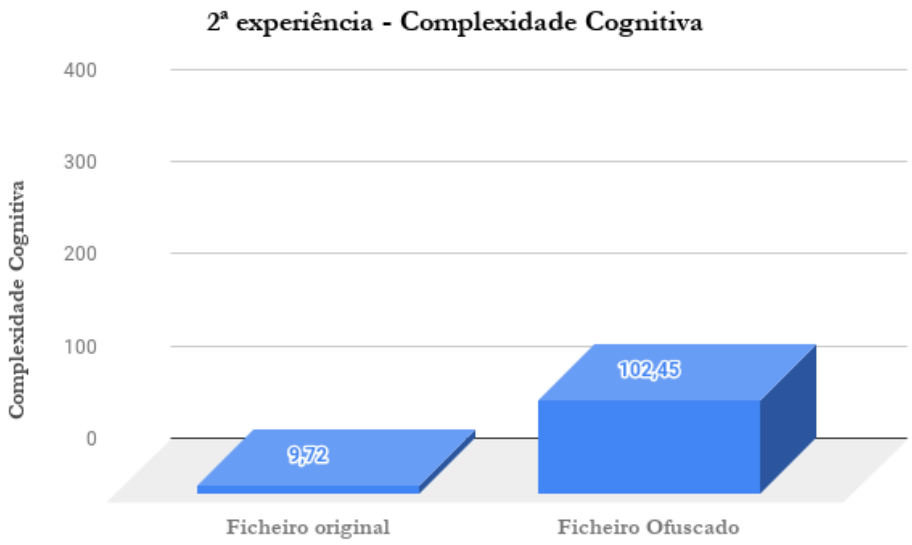

Figura 4.4: Alteração provocada na Complexidade Cognitiva após a Experiência 2.

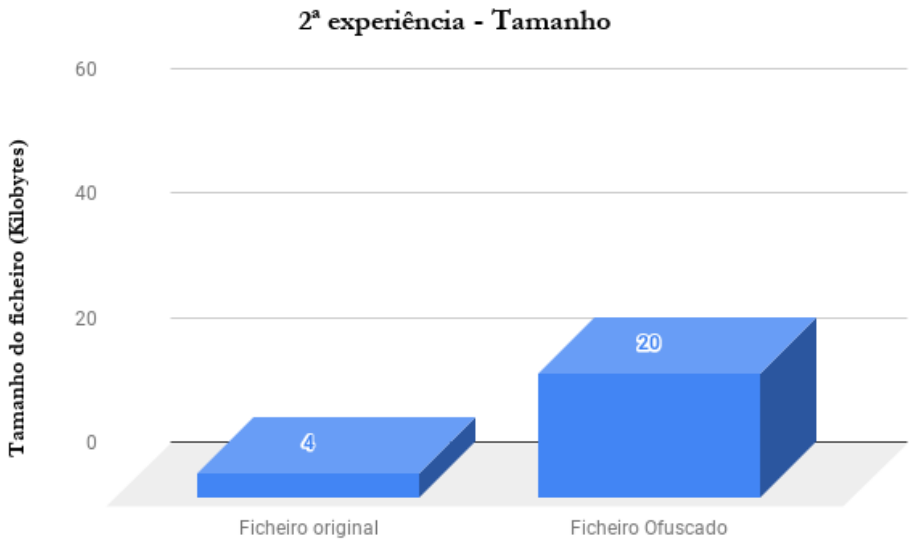

Figura 4.5: Variação no Tamanho do Ficheiro de teste após a Experiência 2.

4.3 Conclusões

No final desta etapa, foi possível concluir que o modelo de recomendação proposto têm como base não só recomendar transformações que permitam satisfazer as necessidades das organizações, mas também avaliar o seu impacto.

Além disso, verificou-se que é bastante importante que as transformações aplicadas não provoquem um aumento excessivo no nível de ofuscação, uma vez que pode traduzir-se numa variação negativa da performance. Porém, isto pode não ser visto como uma desvantagem, caso as organizações possuam margem, ou até mesmo meios para proporcionar esse aumento radical na performance das suas aplicações. Assim, caso as organizações possuam estes recursos, o nível de ofuscação pode ser elevado até um nível que é praticamente impossível para os atacantes efetuar algum tipo de adulteração. Todavia, normalmente estes recursos sãos limitados e, para isso, foi definido um intervalo para níveis de ofuscação definido como ótimo, que permite encontrar um equilibro entre complexidade e performance.

Finalmente, e concluído este capítulo, que procura validar os resultados obtidos no final do desenvolvimento deste projeto de dissertação, verifica-se que os resultados foram satisfatórios. Para isso, contribuíram as experiências realizadas, bem como uma adequada fase de ponderação.

Capítulo 5

Conclusões e trabalho futuro

Atualmente, verifica-se que a maior parte das empresas de software começa a mostrar um maior interesse pela utilização de ferramentas de proteção de código nos seus produtos. Este surge especialmente devido à globalização das aplicações web. Neste âmbito, é apresentado neste relatório um sistema de recomendação que permite que as organizações usem esse tipo de ferramentas de uma maneira mais eficiente e fiável, diminuindo significativamente a configuração do sistema de proteção para obtenção de um resultado ótimo para as suas aplicações. Os resultados experimentais efetuados até ao momento demonstram a precisão e eficiência da ferramenta desenvolvida.

Embora os sistemas de recomendação tenham progredido ao longo dos últimos tempos, na área de proteção de software o seu desenvolvimento é praticamente inoperante. Este fator tornou-se um obstáculo ao modelo inicialmente proposto, uma vez que era necessário um modelo que abandonasse as preferências e necessidades do utilizador, e que incluísse métricas objetivas relacionadas com a ofuscação. Apesar das alterações que foram efetuadas ao modelo de recomendação inicial, expostas no capítulo 3, é importante referir que foi possível dar resposta ao principal e basilar objetivo.

A ferramenta desenvolvida, que se enquadra no métodos de recomendação *Memory-based*, pode ser decomposta em duas fases. A primeira, que faz uso do analisador sintático *Esprima* para a recolha de elementos alvo do extrato de código ou programa, oferece ao utilizador não só um conjunto de ferramentas de proteção, como também as sinergias correspondentes. Em seguida, a segunda fase envolve a análise do resultado obtido pelas ferramentas de proteção selecionadas pelo utilizador, ou seja, mensurar o nível de ofuscação. A análise relativa aos resultados obtidos por este modelo de recomendação, confirmou as vantagens descritas. Não obstante ter sido desenvolvido um modelo de recomendação com diversas potencialidades, existe ainda um grande trabalho a desenvolver, no sentido de tornar este modelo mais fiável e preciso, com a integração de novas métricas, por exemplo.

Como trabalho futuro pretende-se continuar a estudar a evolução dos sistemas de recomendação, bem como investigar métricas, para além da Complexidade Cognitiva e Complexidade Espacial de código, que contribuam para a avaliação do nível de ofuscação. O passo seguinte seria a implementação de um sistema que permitisse recomendar transformações baseando-se apenas na similaridade de código, isto é, apoiando-se na similitude do programa inserido pelo utilizador com programas devidamente ofuscados. Do mesmo modo, a criação de um sistema de ranking que ordenasse as ferramentas de proteção por ordem crescente de importância de acordo com o ficheiro inserido pelo utilizador, constitui uma alternativa válida. Posteriormente, e de forma a consolidar os resultados, aconselha-se à análise da opinião dos utilizadores acerca das transformações recomendadas pelo sistema, com intuito de avaliar o seu nível de satisfação.

Referências

[Cam16] G. Ann Campbell. *Cognitive Complexity - A new of measuring understandability*. SonarSource S.A, 2016. URL: https://www.sonarsource.com/docs/CognitiveComplexity.pdf.

[CDG+11] C. Collberg, J. Davidson, R. Giacobazzi, Y. X. Gu, A. Herzberg e F. Y. Wang. Toward digital asset protection. *IEEE Intelligent Systems*, 26(6):8–13, Nov 2011. doi:10.1109/MIS.2011.106.

[cha18] Ecmascript parsing infrastructure for multipurpose analysis, 2018. [Online; acedido 7-Abril-2018]. URL: http://esprima.readthedocs.io/en/latest/syntactic-analysis.html.

[Chh11] Jitender Chhabra. Code cognitive complexity: A new measure. 2191, 07 2011.

[CJ03] Mihai Christodorescu e Somesh Jha. Static analysis of executables to detect malicious patterns. In *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*, SSYM'03, pages 12–12, Berkeley, CA, USA, 2003. USENIX Association. URL: http://dl.acm.org/citation.cfm?id=1251353.1251365.

[CTL97] Christian Collberg, Clark Thomborson e Douglas Low. A taxonomy of obfuscating transformations, 1997.

[Doc18a] MDN Web Docs. Sintaxe, blocos, 2018. [Online; acedido 2-Maio-2018]. URL: https://developer.mozilla.org/pt-PT/docs/Web/JavaScript/Reference/Extratos_e_declara%C3%A7%C3%B5es/block.

[Doc18b] MDN Web Docs. Switchstatement, 2018. URL: https://developer.mozilla.org/pt-PT/docs/Web/JavaScript/Reference/Statements/switch.

[Doc18c] MDN Web Docs. Ty and catch, statements, 2018. [Online; acedido 11-Junho-2018]. URL: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Statements/try_catch.

[ECM] Draft ecma-262 / february 1, 2018. [Online; acedido 18-Dezembro-2017]. URL: https://tc39.github.io/ecma262/.

[Fou] Node.js Foundation. About. [Online; acedido 13-Janeiro-2018]. URL: https://nodejs.org/en/about/.

[FRK11] Bracha Shapira Francesco Ricci, Lior Rokach e Paul B. Kantor. *Recommender Systems Handbook*. Springer, Boston, MA, First edition, 2011.

REFERÊNCIAS

[git] The only single product for the complete devops lifecycle. [Online; acedido 19-Janeiro-2018]. URL: https://about.gitlab.com/.

[Jsc18] Jscrambler. Obfuscation techniques, 2018. [Online; acedido 5-Março-2018]. URL: https://docs.jscrambler.com/code-integrity/documentation/transformations.

[McC76] T. J. McCabe. A complexity measure. *IEEE Transactions on Software Engineering*, SE-2(4):308–320, Dec 1976. doi:10.1109/TSE.1976.233837.

[NT15] P. Nagarnaik e A. Thomas. Survey on recommendation system methods. In *2015 2nd International Conference on Electronics and Communication Systems (ICECS)*, pages 1603–1608, Feb 2015. doi:10.1109/ECS.2015.7124857.

[RS95] J. Rekers e A. Schurr. A graph grammar approach to graphical parsing. In *Proceedings of Symposium on Visual Languages*, pages 195–202, Sep 1995. doi:10.1109/VL.1995.520809.

[Scr] Scrum.org. What is scrum? [Online; acedido 16-Janeiro-2018]. URL: https://www.scrum.org/resources/what-is-scrum.

[SFHS07] J. Ben Schafer, Dan Frankowski, Jon Herlocker e Shilad Sen. *Collaborative Filtering Recommender Systems*, pages 291–324. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[Sul18] Frost Sullivan. Global security vendors focus on holistic web protection solutions to fulfill rising demand for optimal user experience, 2018. [Online; acedido 3-Fevereiro-2018]. URL: https://www.prnewswire.com/news-releases/global-security-vendors-focus-on-holistic-web-protection-solutions-to-fulfi
html.

[VRT+16] A. Viticchié, L. Regano, M. Torchiano, C. Basile, M. Ceccato, P. Tonella e R. Tiella. Assessment of source code obfuscation techniques. In *2016 IEEE 16th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 11–20, Oct 2016. doi:10.1109/SCAM.2016.17.

[WH16] Dinuka Rukshani Wijendra e K. P. Hewagamage. Automated tool for the calculation of cognitive complexity of a software. In *2016 2nd International Conference on Science in Information Technology (ICSITech)*, pages 163–168, Oct 2016. doi:10.1109/ICSITech.2016.7852627.

Anexo A

Ficheiro de Teste

O Ficheiro apresentado a seguir foi disponibilizado pela *Jscrambler*.

```
/**
 * HTML5 canvas animated clock.
 * https://developer.mozilla.org/en-US/docs/Web/API/Canvas_API/Tutorial/
    Basic_animations#An_animated_clock
 */
(function () {
  // Clean up HTML body
  var body = document.querySelector('body');
  while (body.firstChild) {
    body.removeChild(body.firstChild);
  }

  // Create canvas
  var canvas = document.createElement('canvas');
  canvas.setAttribute('id', 'canvas');
  canvas.setAttribute('width', '150px');
  canvas.setAttribute('height', '150px');
  body.appendChild(canvas);
  var ctx = document.getElementById('canvas').getContext('2d');

  function clock () {
    ctx.save();
    ctx.clearRect(0, 0, 150, 150);
    ctx.translate(75, 75);
    ctx.scale(0.4, 0.4);
    ctx.rotate(-Math.PI / 2);
    ctx.strokeStyle = "black";
    ctx.fillStyle = "white";
    ctx.lineWidth = 8;
    ctx.lineCap = "round";

    hourMarks();
```

```
32      minuteMarks();
33
34      var now = new Date();
35      var sec = now.getSeconds();
36      var min = now.getMinutes();
37      var hr = now.getHours();
38      hr = hr >= 12 ? hr - 12 : hr;
39
40      ctx.fillStyle = "black";
41
42      writeHours(hr, min, sec);
43      writeMinutes(min, sec);
44      writeSeconds(sec);
45
46      ctx.beginPath();
47      ctx.lineWidth = 14;
48      ctx.strokeStyle = 'black';
49      ctx.arc(0, 0, 142, 0, Math.PI * 2, true);
50      ctx.stroke();
51
52      ctx.restore();
53
54      window.requestAnimationFrame(clock);
55    }
56
57    function hourMarks () {
58      ctx.save();
59      for (var i = 0; i < 12; i++) {
60        ctx.beginPath();
61        ctx.rotate(Math.PI / 6);
62        ctx.moveTo(100, 0);
63        ctx.lineTo(120, 0);
64        ctx.stroke();
65      }
66      ctx.restore();
67    }
68
69    function minuteMarks () {
70      ctx.save();
71      ctx.lineWidth = 5;
72      for (i = 0; i < 60; i++) {
73        if (i % 5 != 0) {
74          ctx.beginPath();
75          ctx.moveTo(117, 0);
76          ctx.lineTo(120, 0);
77          ctx.stroke();
78        }
79        ctx.rotate(Math.PI / 30);
80      }
```

```
81      ctx.restore();
82    }
83
84    function writeHours (hr, min, sec) {
85      ctx.save();
86      ctx.rotate(hr * (Math.PI / 6) + (Math.PI / 360) * min + (Math.PI / 21600) * sec
            )
87      ctx.lineWidth = 14;
88      ctx.beginPath();
89      ctx.moveTo(-20, 0);
90      ctx.lineTo(80, 0);
91      ctx.stroke();
92      ctx.restore();
93    }
94
95    function writeMinutes (min, sec) {
96      ctx.save();
97      ctx.rotate((Math.PI / 30) * min + (Math.PI / 1800) * sec)
98      ctx.lineWidth = 10;
99      ctx.beginPath();
100     ctx.moveTo(-28, 0);
101     ctx.lineTo(112, 0);
102     ctx.stroke();
103     ctx.restore();
104   }
105
106   var hexCodeValidator = /^#([A-Fa-f0-9]{6}|[A-Fa-f0-9]{3})$/;
107   var color;
108   if (hexCodeValidator.test("#D40000")) {
109     color = "#D40000";
110   }
111
112   function writeSeconds (sec) {
113     ctx.save();
114     ctx.rotate(sec * Math.PI / 30);
115     ctx.strokeStyle = color;
116     ctx.fillStyle = color;
117     ctx.lineWidth = 6;
118     ctx.beginPath();
119     ctx.moveTo(-30, 0);
120     ctx.lineTo(83, 0);
121     ctx.stroke();
122     ctx.beginPath();
123     ctx.arc(0, 0, 10, 0, Math.PI * 2, true);
124     ctx.fill();
125     ctx.beginPath();
126     ctx.arc(95, 0, 10, 0, Math.PI * 2, true);
127     ctx.stroke();
128     ctx.fillStyle = "rgba(0,0,0,0)";
```

```
129     ctx.arc(0, 0, 3, 0, Math.PI * 2, true);
130     ctx.fill();
131     ctx.restore();
132   }
133
134   window.requestAnimationFrame(clock);
135 })();
```