FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Ferramenta de Ofuscação de Código Javascript

Filipe Manuel Gomes Silva

Relatório de Projecto
Mestrado Integrado em Engenharia Informática e Computação

Orientador: Professor Doutor João Manuel Paiva Cardoso

Junho de 2009

Ferramenta de Ofuscação de Código Javascript

Filipe Manuel Gomes Silva

Relatório de Projecto
Mestrado Integrado em Engenharia Informática e Computação

Aprovado em provas públicas pelo Júri:

Presidente: Doutor Pedro Alexandre Guimarães Lobo Ferreira do Souto

Arguente: Doutor João Alexandre Baptista Vieira Saraiva

Vogal: Doutor João Manuel Paiva Cardoso

16 de Julho de 2009

Resumo

O código *Javascript* é disponibilizado na sua totalidade ao cliente para ser executado pelo navegador. Por essa razão fica exposto a análise e a possíveis modificações. É sugerido neste projecto a utilização de técnicas de ofuscação de código e anti-depuração (algumas especificas para *Javascript malware*) para contrariar esse tipo de ataques. Faz-se assim a investigação do estado da arte da ofuscação, estudando as técnicas empregues divididas em dois grupos: técnicas polimórficas, reconhecidas pela alteração da forma do programa (e.g., codificação, encriptação) e técnicas metamórficas conhecidas pela alteração da estrutura do código e fluxo de execução. São também estudadas as técnicas de anti-depuração e as técnicas e ferramentas que servem de ataque à ofuscação.

No âmbito deste projecto foi desenvolvida uma ferramenta que automatiza o processo de ofuscação de código *Javascript*. É apresentada a arquitectura da ferramenta bem como o detalhe de implementação das transformações desenvolvidas e um resumo destas, apresentando-as quanto ao tipo de transformação, alvo da transformação, esforço de desenvolvimento, potência de ofuscação e resistência a inversão automática de ofuscação esperados e custo de ofuscação.

Os resultados experimentais focam a avaliação do desempenho em alguns dos navegadores mais populares, demonstrando que estes podem condicionar a ofuscação por necessitarem de tempos de execução mais elevados, e por vezes, pela quebra da funcionalidade do código. Apresenta-se uma análise detalhada que inclui avaliações dos tempos de execução do código ofuscado, a performance da ferramenta na aplicação das transformações e o crescimento do tamanho do código.

Os resultados evidenciam a importância da relação entre custo e eficácia de ofuscação, aconselhando-se que seja reduzida a potência das transformações mais onerosas em vez da sua não utilização. Torna-se evidente a promissora utilização de funções de *hash* na construção de predicados opacos e são apontadas vulnerabilidades na codificação de código – mais especificamente no contexto *Javascript* – introduzidas pelo crescimento do número de *plugins* de depuração para navegadores, cuja actividade dificilmente conseguirá ser detectada como sendo intrusiva.

Abstract

Javascript code is fully handed to the client so it can be executed by the navigator. For this reason Javascript code is exposed to analysis and to possible modifications. The goal of this project consists on the use of code obfuscation and anti-debugging techniques (e.g. Javascript malware specific) to oppose this kind of attacks. State of the art of obfuscation is thus studied, and its techniques are shown divided in two groups: polymorphic techniques, used to modify the appearance of the program (e.g., codification, encryption), and metamorphic techniques, known for applying modifications at code structure level and execution flow. Anti-debugging techniques and techniques and tools that serve as an attack to obfuscation are also studied.

This project presents a tool that automates the process of Javascript code obfuscation. The tool's architecture as well as the implementation details of the developed transformations are presented and summarized. Those transformations are presented by their transformation type, transformation targets, development effort, obfuscation potency, resilience to automatic deobfuscation and cost introduced by obfuscation.

The experimental results focus the performance evaluation on some of the most popular Web browsers, indicating that browsers may restrict the obfuscation by needing longer execution times, and/or, by breaking code's functionality. A detailed analysis is presented, including an evaluation of the obfuscated code execution times, the tool's performance in the application of transformations and the size growth of the code.

The importance of the relation between obfuscation cost and obfuscation efficiency is highlighted by the results, recommending the decrease of obfuscation level of the most onerous transformations instead of not using them at all. These results emphasize the promising use of hash functions in the construction of opaque predicates and vulnerabilities in the code codification are shown – specifically in the Javascript context –, introduced by the growth of debugging plugins for browsers, whose activity will be hardly detected as being invasive.

Palavras Chave

Ofuscação de Código

Javascript Malware

Técnicas Polimórficas

Técnicas Metamórficas

Técnicas de Anti-Depuração

Eficácia de Ofuscação

Keywords

Code Obfuscation

Javascript Malware

Polymorphic Techniques

Metamorphic Techniques

Anti-Debugging Techniques

Obfuscation Efficiency

Dedico este projecto a meus pais,

Agradecimentos

Ao Professor João Manuel Paiva Cardoso, pela total disponibilidade demonstrada e muito prezados conselhos e orientação.

À *AuditMark Lda*, pela oportunidade de realizar um projecto tão cativante e pelo apoio prestado ao longo da sua realização.

Aos meus amigos, por me oferecerem bons momentos de descontracção, sempre muito apreciados, principalmente em momentos mais exigentes como os que passaram.

Aos meus pais, Manuel Gonçalo Sousa Silva e Maria Sameiro Martins Gomes, pela constante preocupação com o meu bem-estar e por me ajudarem sempre. Pela naturalidade com que se empenham no objectivo de fazer bem e de serem melhores pessoas, qualidades que influenciam a minha maneira de ser e a forma como abordo os problemas.

À minha querida Andreia, pelo carinho e amor que me dedicou e por ter feito parte no meu entusiasmo durante a evolução do projecto.

Porto, Junho de 2009

Índice

Lista de Figuras

Lista de Tabelas

Abreviaturas

AJOT	*Advanced Javascript Obfuscation Tool*
ANTLR	*ANother Tool for Language Recognition*
API	*Application Programming Interface* (interface de programação de aplicações)
ASCII	*American Standard Code for Information Interchange*
AST	*Abstract Syntax Tree* (árvore sintáctica abstracta)
CPU	*Central Processing Unit* (unidade central de processamento)
DOM	*Document Object Model*
FEUP	Faculdade de Engenharia da Universidade do Porto
HMAC	*Hash Message Authentication Code*
HTML	*Hypertext Mark-up Language*
JAVACC	*Java Compiler Compiler*
JIC	*Javascript Interaction Code*
MD5	*Message-Digest algorithm 5*
MIEIC	Mestrado Integrado em Engenharia Informática e Computação
MSDN	*Microsoft Developer Network*
PEB	*Process Enviroment Block*
RAM	*Random-Access Memory*
SHA-1	*Secure Hash Algorithm 1*
TEB	*Thread Information Block*
UTF-8	*8-bit Unicode Transformation Format*

1. Introdução

A ofuscação de código consiste na alteração da forma e estrutura do código para dificultar a sua compreensão (sem alterar a sua funcionalidade). A Figura 1.1 ilustra o processo de ofuscação representado pela transformação do programa original na sua representação ofuscada, mantendo-se a mesma funcionalidade do programa original (ambos recebem x e devolvem como resultado y).

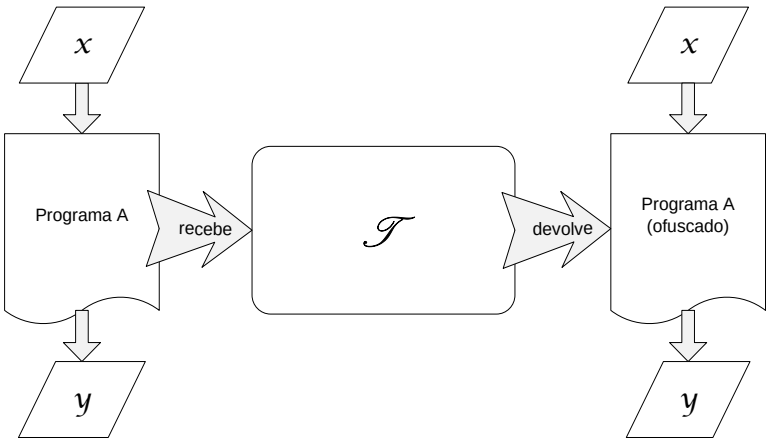


Figura 1.1 – Transformação do Programa A na sua representação ofuscada com a mesma funcionalidade.

Este trabalho consiste na implementação de uma ferramenta que automatize este processo de ofuscação para código *Javascript*. O presente capítulo contextualiza o tema do projecto e apresenta os problemas que justificam a sua realização. O capítulo contém também a explicação da motivação e objectivos do projecto e apresenta a organização deste relatório.

1.1. Contexto/Enquadramento

A transformação de um programa é a operação que utiliza um programa de origem e fornece um programa destino tendo em conta determinadas modificações do código. Esta área abrange muitas outras áreas que seguem a mesma operação mas que diferem em alguns pontos, criando assim,

uma taxonomia de transformações que se divide primeiramente em duas áreas principais [1]: tradução e refraseamento. A primeira consiste na geração de um programa descrito numa linguagem diferente da linguagem do programa original. A segunda consiste na mesma operação de transformação, mas cujo resultado é um programa na mesma linguagem utilizada no programa original. Dentro das transformações que se encontram no refraseamento (e.g., normalização de programas, optimização de programas), encontra-se a transformação que interessará estudar neste projecto: ofuscação de programas. Ao contrário de outras transformações que se encontram no mesmo grupo, cujo objectivo principal é optimizar o programa ou reduzir o seu tamanho, a ofuscação procura aumentar a complexidade do programa para que a análise da sua funcionalidade seja dificultada.

1.2. Projecto

O projecto foi proposto pela empresa *AuditMark Lda* [2] ao MIEIC como projecto de final de curso. A empresa disponibiliza um serviço de auditoria que analisa a qualidade do tráfego no clique em publicidade *online*. Este serviço contém um programa denominado de *Javascript Interaction Code* (JIC) que auxilia no processo de avaliação da qualidade do tráfego. O JIC baseia-se (na sua maioria) na utilização de *Javascript* e apresenta vulnerabilidades que merecem atenção. Sendo o programa JIC executado do lado do cliente, pode ser facilmente utilizado para análise e para possíveis tentativas de modificar a sua funcionalidade. Com o objectivo de se encontrarem soluções para um problema de completa exposição intelectual e funcional à análise mal intencionada do programa JIC, torna-se importante a implementação de uma ferramenta que aplique transformações de ofuscação para proteger o código (sem alterar a sua funcionalidade). Para isso, foi necessário estudar o estado da arte das técnicas de ofuscação e anti-depuração e testar a viabilidade da aplicação dessas mesmas técnicas ao programa JIC e a outros programas em *Javascript*.

1.3. Motivação e Objectivos

O principal objectivo deste projecto é a implementação de uma ferramenta de ofuscação de código *Javascript*. A identificação das técnicas de ofuscação do estado da arte e a análise de resultados da aplicação das transformações de ofuscação é também considerada relevante no contexto deste projecto. A motivação para a implementação de raiz de uma ferramenta de ofuscação fundamenta-se pela necessidade de encontrar uma solução que reduza a vulnerabilidade do JIC (apresentada anteriormente).

Depois da análise da oferta de serviços e ferramentas de ofuscação de código *Javascript* feita pela *AuditMark Lda*, constatou-se que o custo de tais serviços (ou ferramentas) era muito alto para o poder de ofuscação que ofereciam (na sua grande maioria, apenas transformações polimórficas). Seria mais rentável (e seguro) iniciar um projecto que estudasse o estado da arte da ofuscação e que aplicasse esse conhecimento na construção de uma ferramenta que oferecesse um leque de transformações mais alargado (e.g., polimórficas, metamórficas, anti-depuração).

1.4. Estrutura do Relatório

Este relatório encontra-se organizado em cinco capítulos:

- O segundo capítulo começa por apresentar os alvos da ofuscação e as métricas que avaliam a qualidade de ofuscação. Apresenta o estado da arte da ofuscação com as técnicas polimórficas, metamórficas e técnicas de anti-depuração. Apresenta também as técnicas de ofuscação utilizadas especificamente no código *Javascript e* os ataques que a ofuscação pode ser alvo.
- O terceiro capítulo apresenta a especificação e detalhe de implementação da ferramenta. Pode-se encontrar a descrição do âmbito e perspectiva da ferramenta, as funcionalidades a implementar, as técnicas de ofuscação e anti-depuração seleccionadas como possibilidades para a implementação e respectivos critérios de selecção, bem como outras informações relacionadas. Apresenta também as possíveis tecnologias a utilizar na implementação e as que acabaram por ser escolhidas, a arquitectura da ferramenta, o detalhe da implementação de cada uma das transformações desenvolvidas, acompanhadas de um exemplo prático do resultado da transformação, terminando com um resumo das técnicas implementadas.
- O quarto capítulo mostra os resultados experimentais obtidos em testes de performance efectuados à ferramenta na ofuscação de dois ficheiros de teste: protótipo JIC e um segundo ficheiro, constituído por uma compilação de funções *Javascript* obtidas num repositório *online* [3].
- Por fim, o último capítulo apresenta as conclusões tiradas na realização do projecto e enuncia algum trabalho futuro.

2. Ofuscação de Código

Neste capítulo serão apresentados os alvos da ofuscação, os critérios de avaliação da qualidade de ofuscação, os ataques possíveis à ofuscação, e as várias técnicas de ofuscação (e anti-depuração) reunidas nos seguintes grupos: Polimórficas, Metamórficas e Anti-Depuração. O capítulo termina com a apresentação das técnicas de ofuscação e de análise de ofuscação específicas à linguagem *Javascript*.

2.1. Introdução

A cópia de algoritmos e a sua compreensão pode pôr em causa a competitividade entre empresas ou a segurança de um sistema. Quando não é possível evitar que o código de um programa esteja vulnerável a análise, é necessário protegê-lo para que este não seja facilmente compreendido. Algumas soluções técnicas podem ser utilizadas para dificultar a análise do código. A encriptação ou a execução parcial de código num servidor são algumas das possibilidades existentes [4] [5], mas a mais interessante e a que será alvo de estudo neste projecto é a ofuscação de código. A ofuscação de código consiste na alteração da forma e estrutura do código para dificultar a sua compreensão (sem alterar a sua funcionalidade) e é actualmente uma das melhores opções de protecção para soluções que disponibilizem a totalidade do código ao utilizador [4] [5].

2.2. Alvos da Ofuscação

A ofuscação de código atinge diferentes alvos no código de um programa. Os alvos mais representativos na ofuscação de código são apresentados nos três grupos seguintes [4] [5]:

- Ofuscação da disposição (em inglês, *layout obfuscation*), tratando-se de pequenas alterações de disposição ou eliminação de informação ou formatação não necessárias ao funcionamento do código.
- Ofuscação de dados, caracterizada pela transformação a nível das estruturas de dados.

- Ofuscação de controlo, caracterizada pela transformação a nível do controlo de fluxo.

As técnicas de ofuscação pertencentes ao primeiro grupo encontram-se agrupadas na secção que apresenta as técnicas de ofuscação polimórficas, onde as transformações alteram apenas a forma (e.g. codificação, encriptação) e disposição do código (e.g., remoção de comentários, remoção de espaços). Os dois últimos grupos são os mais representativos das transformações de ofuscação e estão agrupados na secção das transformações metamórficas. Estas transformações alteram profundamente o fluxo de execução e as estruturas de dados.

2.3. Qualidade de Ofuscação

A avaliação da qualidade da ofuscação é feita tendo em conta quatro critérios [4] [5] [6]. Estes são: a potência da ofuscação, a resistência a ataques automáticos de inversão de ofuscação, a capacidade de não detecção da aplicação de ofuscação, e o custo que é acrescentado pela ofuscação. O resultado da ofuscação estará tão mais próximo do ideal quanto maior for a potência, resistência e não detecção, e menor o custo de ofuscação. A medição desses critérios é apresentada nas secções seguintes.

2.3.1. Potência

A potência de ofuscação mede a dificuldade de compreensão do código por um humano. O código será tão ou mais difícil de compreender, quanto maior for a complexidade do mesmo. A medição da complexidade do código é feita de diferentes formas. Alguns autores sugerem que o código será tão mais complexo, quanto maior for o número de predicados que tiver [7]. Outros consideram que a complexidade cresce com a profundidade de saltos condicionais e de ciclos [8], com a complexidade das estruturas de dados utilizadas [9], com o tamanho do programa [10] ou com a profundidade de ligações de herança numa linguagem orientada a objectos [11]. Qualquer técnica de ofuscação que, aplicada ao código, aumente pelo menos uma destas métricas, está a aumentar a potência de ofuscação. Estas métricas servem para (de uma forma menos informal) apresentar uma avaliação qualitativa sobre a potência de cada técnica de ofuscação. De qualquer forma, a sua utilização é subjectiva, pois não deixamos de estar a medir a capacidade de compreensão por humanos.

2.3.2. Resistência

Ao contrário da potência de ofuscação, que classifica as transformações quanto à dificuldade que adicionam na compreensão do código por humanos, a resistência mede a dificuldade da inversão da ofuscação por mecanismos automáticos de inversão. Segundo o autor [4] [5], a resistência da ofuscação é medida em função de dois critérios. Um dos critérios é o tempo necessário à construção de um programa de inversão (esforço de programação). Esse tempo aumenta com a abrangência da análise que a ofuscação do código obriga (de uma análise local até uma análise inter-processos). O outro critério é o tempo e espaço que esse programa necessita para reduzir a potência do código ofuscado (esforço de inversão). Com isto podemos dizer que quanto mais

abrangente for o alvo da inversão de ofuscação e maior o tempo e espaço necessários à inversão, mais difícil será a inversão da ofuscação (maior resistência).

2.3.3. Não Detecção

A não detecção é a capacidade da aplicação de uma técnica de ofuscação passar despercebida aquando da análise do código ofuscado. Por vezes, transformações resistentes à inversão automática de ofuscação destacam-se do código original - pelo exagero ou por estarem fora do contexto - e passam a ser facilmente detectáveis por quem está a analisar o código. Um equilíbrio tem de ser encontrado entre a qualidade de não detecção e a resistência, para dificultar tanto o trabalho do analista como dos mecanismos de inversão automática (pois sem alvo não há ataque). O exemplo seguinte (Figura 2.1) representa este problema. O enunciado é facilmente detectado por um humano devido ao exagero.

```
512-bit integer
_____________/_____________
if IsPrime(837523474 … 3853845347527) then …
```

Figura 2.1 – Exemplo retirado de Collberg [5] que representa uma condição de salto resistente, mas facilmente detectável por humanos.

2.3.4. Custo

O custo da ofuscação é o custo adicionado – em tempo e espaço – ao código original pela ofuscação. Esse custo é representado pela notação assimptótica (*Big-Oh notation*) que apresenta a forma de crescimento do tempo necessário pela computação (depois da transformação) em função da dimensão do problema ou da entrada. Se anteriormente existia uma computação com custo n e após a aplicação da técnica de ofuscação é incluído um nível de complexidade m dentro do nível n, então o custo da transformação aumentou de O(n) para O(nxm), como é ilustrado no exemplo da Figura 2.2.

Antes:

```
ciclo(cond) //n vezes
```

Depois de ofuscado:

```
ciclo(cond) //n vezes
   ciclo(cond2) //m vezes
```

Figura 2.2 – Crescimento do custo de O(n) para O(nxm) após ofuscação.

2.4. Técnicas de Ofuscação: Polimórficas

O polimorfismo (de código) é a capacidade do código sofrer mutações, alterando a sua forma sempre que é criado uma nova cópia do mesmo. Esta mutação não altera a funcionalidade do código original e pode ser conseguida, por exemplo, utilizando um algoritmo criptográfico reversível com uma chave simétrica diferente de cada vez que é criada uma nova cópia. Esta ideia nasceu da necessidade de dificultar a detecção dos vírus [12]. O que até então teria sido sempre utilizado para fins não duvidosos – a criptografia – ganha utilidade no campo do software malicioso. O vírus seria então constituído pelo código cifrado e por uma rotina cuja função é decifrar a informação cifrada que esconde o código do vírus, sempre que este é executado. Para dificultar a identificação do código malicioso alteram-se a rotina inicial e a chave, de cada vez que é criada uma nova cópia do vírus. Dificulta-se ainda mais a análise da funcionalidade do código, utilizando criptografia juntamente com técnicas de ofuscação, aumentando ainda mais a resistência a *signature-based detection*[1] [12].

Numa solução comercial em que o código do programa está vulnerável à análise, a utilização destas técnicas pode trazer vantagens, justificando assim o seu estudo. As técnicas polimórficas de ofuscação são apresentadas de seguida.

2.4.1. Renomeação de Identificadores

Os identificadores suportam o nome pelo qual nos referimos a elementos no código (e.g., variáveis, funções, classes). Um identificador oferece uma forma de referenciar o elemento que identifica, mas também, dá algumas pistas acerca da sua funcionalidade. É usual ser feita a atribuição de nomes de forma a ajudar quem analisa o código a identificar mais facilmente a funcionalidade do elemento. Ao trocarem-se esses nomes por uma sequência de caracteres sem qualquer tipo de significado, um humano terá mais dificuldade em perceber a sua funcionalidade e a fazer a sua associação com uma repetição do mesmo em outra parte do programa (ver Figura 2.3).

```
function read(str) → function zA_7d7Af(B1nF_)
```

Figura 2.3 – Exemplo de alteração dos identificadores.

Esta alteração pode ser feita de várias formas. A mais simples e eficaz é a substituição por identificadores criados aleatoriamente. Assim não se conseguirá reaver os identificadores originais e obtém-se o efeito pretendido.

[1] Técnica utilizada pelos anti-vírus para analisar o conteúdo dos ficheiros na procura de assinaturas de código malicioso já conhecidas.

2.4.2. Remoção de Comentários

A remoção de comentários é óbvia, simples e essencial. Os comentários possuem a informação em linguagem natural, sobre a funcionalidade ou propósito do código que comentam. O processo de inversão da ofuscação não é possível, pois não é possível reaver informação eliminada. Em alguns casos são acrescentados comentários falsos mas em nada melhoram a potência da ofuscação. O custo acrescentado pela remoção de comentários é nulo.

2.4.3. Modificação da Formatação

A modificação da formatação passa pela eliminação de espaços horizontais e verticais, entre os elementos existentes no código. A modificação da formatação pouca influência terá na percepção do código, introduzindo assim pouca potência ao resultado da ofuscação. O custo associado a esta transformação é nulo, pois nada foi alterado que agravasse a computação.

2.4.4. Substituição de Números

A substituição de números por um sistema de numeração diferente, também é uma transformação que altera a forma como se apresentam os dados no código. Veja-se o exemplo da Figura 2.4.

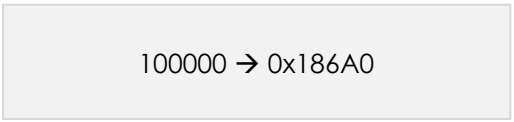


Figura 2.4 – Substituição de um número inteiro (decimal) pela sua representação em hexadecimal.

Neste exemplo um número representado em forma decimal é substituído pela representação hexadecimal equivalente.

2.4.5. Substituição de Caracteres

A substituição de cadeias de caracteres pelas suas representações codificadas pode ser feita, por exemplo, pela representação ASCII em hexadecimal (ver Figura 2.5).

Figura 2.5 – Representação de uma cadeia de caracteres na sua forma ASCII.

À semelhança da substituição de números apresentada anteriormente, também se torna penosa a inversão manual dos dados ofuscados, sendo mais eficiente recorrer a soluções automáticas. O custo adicionado é nulo pois só se alterou a representação da informação.

2.4.6. Alterar a Codificação da Informação

Esta transformação altera a formatação da informação atribuída a uma representação simples para uma expressão matemática que devolva o mesmo valor. Baseado num exemplo de Collberg [4], ilustrado na Figura 2.6, imaginemos uma variável i (do tipo inteiro) que suporta inicialmente o valor 1 e que será substituída (na transformação) pela expressão i' = C1 x i + C2 (C1 e C2 são constantes). Outras expressões podem ser utilizadas desde que a funcionalidade inicial não se perca.

Antes:

```
int i = 1;
while(i < 1000){
   ... A[i] ...;
   i++;
}
```

Depois de ofuscado:

```
int i = 11;
while(i < 8003){
   ... A[(i - 3)/8] ...;
   i+=8;
}
```

Figura 2.6 – Exemplo da alteração da codificação retirado de Collberg [4].

É necessário ter-se em atenção, se o tipo de variável consegue armazenar os novos valores. No exemplo dado, C1 e C2 terão de ser escolhidos de forma que o resultado não ultrapasse o limite que uma variável do tipo inteiro consegue armazenar (entre -2^{31} e $2^{31}-1$). Se a representação usada deixar de ser suficiente para suportar a informação, será necessário alterá-la (e.g., de int para long).

2.4.7. Codificação e Encriptação

A codificação e encriptação do código também são técnicas que alteram a forma como o código se apresenta. Estas técnicas formam a primeira camada de protecção do código, ou seja, o primeiro obstáculo à análise. O custo de ofuscação adicionado pela transformação dependerá do tamanho do código a decifrar e da rotina de descodificação. A potência de ofuscação é tanto maior quanto mais complexa for a rotina de descodificação. A resistência à inversão automática é alta, pois dificilmente se conseguirá construir uma ferramenta automática que analise, identifique, e aplique a inversão de uma transformação que tem como alvo todo o código do programa. Na

prática a abordagem a este tipo de transformação é feita através de uma análise manual, tentando utilizar o próprio código de descodificação do programa para obter o código descodificado.

2.5. Técnicas de Ofuscação: Metamórficas

O metamorfismo (de código) é caracterizado pela capacidade de aplicação de transformações no seu fluxo de execução e estrutura de dados, sempre que uma cópia do mesmo é criada [13]. À semelhança do polimorfismo, o efeito procurado é dificultar a compreensão da funcionalidade do código. Alguns *malware* possuem esta capacidade, proporcionando assim, uma melhor defesa contra *signature-based detection e pattern recognition*[2] pelos anti-vírus [12] [14]. Contudo, não é sobre a capacidade metamórfica que se irá falar neste capítulo, mas sim, sobre as transformações características do metamorfismo.

Collberg [4] [5] apresenta uma compilação de transformações metamórficas representativa das possibilidades de ofuscação. Essas transformações estão divididas em dois grupos que representam os alvos em que actuam. As transformações do controlo de fluxo, entre as quais podemos encontrar: inserção de código irrelevante, expansão das condições de terminação de ciclo, transformação de grafo de fluxo reduzível para não reduzível, processamento paralelo, *inlining* e *outlining* de funções, fusão e clonagem de funções, transformação de ciclos, reordenação de elementos, e adição de operandos redundantes. O segundo grupo representa as transformações das estruturas de dados, onde podemos encontrar: divisão de variáveis, fusão de variáveis escalares, conversão de dados estáticos em dados criados dinamicamente, reestruturação de vectores, e modificação de relações de herança. Sakabe [15] e Sosonkin [16] apresentam transformações que tiram partido do polimorfismo de classes (não confundir com o polimorfismo de código). Transformações que utilizam excepções no Java também foram propostas por Sakabe [15]. Algumas transformações metamórficas são propostas por Batchelder [17], por exemplo, transformações que tiram partido de especificidades do *Java bytecode* ou que tentam minar a obtenção do código fonte a partir do *bytecode* – vulnerabilidade encontrada em *Java* e *dotNET*. Outros autores propuseram transformações que não assentam na aplicação que se pretende criar neste projecto por impossibilidade de implementação com a linguagem a ofuscar ou por adicionarem um custo muito alto (apenas com aplicação em código malicioso).

2.5.1. Inserção de Código Irrelevante

Código irrelevante é código que em nada contribui para a funcionalidade de um programa. Como código irrelevante podemos encontrar: *dead code*, *void code* ou *duplicated code*. O *dead code* é código que nunca é executado [4] [13]. O *void code* é código que é executado mas que em nada altera a funcionalidade do programa original [13]. O *void code* poderá manipular informação que não afecte a funcionalidade do programa, ou caso afecte, terá de a restaurar no final. Uma

[2] Reconhecimento de padrões com o objectivo de classificar a informação baseada em conhecimento dedutivo ou informação estatística extraída dos padrões. Aqui referido no contexto de detecção de software malicioso.

vantagem do *void code* em relação ao *dead code* é não denunciar a sua posição no código por inactividade – vulnerabilidade típica do *dead code* relativamente à análise dinâmica de código. *Duplicated code* é uma cópia de um bloco de código que mantém a mesma funcionalidade [13]. Apesar de terem a mesma funcionalidade é importante que estas cópias aparentem ser diferentes. Isto é conseguido com a utilização de diferentes combinações de técnicas de ofuscação em cada cópia. A escolha da execução de uma destas cópias (*Duplicated code*) poderá ser feita com um salto condicional cujo resultado será determinado aleatoriamente, criando assim mais possibilidades de salto.

O custo desta ofuscação, dependerá do código irrelevante inserido ser executado, e se o for, da computação que acrescenta. O local onde é inserido também pode influenciar o custo. É aconselhável não inserir código irrelevante num ciclo com muita profundidade pois poderá aumentar o custo de execução muito para além do que é aceitável. O ideal é a inserção em locais que possam disfarçar outras ofuscações, e.g., entre ciclos resultado da transformação *loop splitting* (apresentado mais à frente). A inserção de código irrelevante e de predicados opacos resistentes à detecção, irá aumentar a potência da ofuscação. A resistência também é aumentada e é tanto maior, quanto maior for a dificuldade de detecção da irrelevância dos predicados adicionados. De qualquer forma, para reforçar a não detecção e a resistência à remoção, é aconselhável aplicar outras técnicas de ofuscação após a inserção do código irrelevante.

Um exemplo interessante da criação de um predicado opaco – pela resistência à inversão de ofuscação que oferece – foi proposto por Beaucamps [13]. Os autores propõem a utilização de funções de *hash* nas condições de salto, dificultando assim, a análise estática do código. Vejamos o exemplo apresentado na Figura 2.7.

```
h is a hash function.
Choose N randomly in a reasonable range.
Compute A = h(N) and B = h(2 * N).
Write in the obfuscated program:
int x = 0;
for (int i = 0; h(i) != A; i ++) {
x += 2;
}
if (h(x) != B) {
/* execute dead code */
}
```

Figura 2.7 – Utilização de funções de *hash* nas condições de salto para dificultar a análise estática do código.

Neste exemplo é escolhido um valor N (em tempo de ofuscação) que servirá de *input* para a função de *hash* calcular os valores de A e B. Numa análise estática do código não se conseguirá dizer quando o ciclo irá terminar, pois não sabemos quando o valor de i (como *input* da função de *hash*) devolverá um valor igual ao A, ocultando assim o número de iterações do ciclo. Por essa razão, quando o ciclo termina, também não saberemos o valor de x e assim também não conseguiremos dizer se o predicado da condição de salto será avaliado de forma a executar - o que em tempo de ofuscação sabemos ser - o *dead code*. Contudo, se for efectuada uma análise dinâmica ao programa será fácil descobrir que o código dentro da condição de salto nunca será

executado, pois os valores de i e x são sempre inicializados a zero, logo o valor de x aquando da terminação do ciclo será sempre o mesmo e assim, também o resultado da avaliação da condição de salto.

Beaucamps [13] propõe uma nova solução: inicializar as variáveis i e x no inicio do programa e reutilizar os seus valores a cada execução do bloco de código, para que nunca sejam os mesmos. Contudo, esta solução não seria viável pois chegaria a um ponto em que o ciclo não seria executado. Outras soluções são apresentadas pelo autor mas introduzem um custo tão alto que deixam de ser uma possibilidade.

2.5.2. Expansão das Condições de Terminação de Ciclo

Num ciclo existe uma condição de terminação (predicado) cuja função é terminar a execução do ciclo quando essa condição for atingida, controlando assim, o fluxo do programa. Expandir as condições de terminação de ciclo significa, adicionar um predicado opaco que aumente a potência da ofuscação sem modificar o resultado da condição de terminação (ver Figura 2.8). É importante que esse predicado não seja facilmente detectável, para que não sejam alvo de remoção.

Antes:

```
int i = 1;
while(i < 100){
   i++;
}
```

Depois de ofuscado:

```
int i = 1, j = 100;
while(i < 100 && (j * j * (j + 1) * (j + 1) % 4 == 0)){
   i++;
   j = j * i + 3;
}
```

Figura 2.8 – Exemplo da expansão da condição de ciclo retirado de Collberg [4].

2.5.3. Transformação de Grafo de Fluxo Reduzível num Não Reduzível

Facilmente se consegue encontrar, no código fonte de um programa, uma representação estruturada do seu controlo de fluxo. As representações estruturadas (e.g., for, while), ao contrário das representações que controlam o fluxo de forma básica e incondicional (e.g., goto), podem ser divididas. Esta divisão consiste na separação de blocos de código em conjuntos de elementos não divisíveis (e.g., b = 100; é não divisível) separados por predicados opacos cujo resultado é sempre o mesmo, criando assim mais possibilidades de fluxo, e tornando a compreensão do código mais difícil. Neste caso imaginemos dois blocos de código – Bloco A e Bloco B – separados por um

elemento que controla o fluxo de execução. Esses blocos são divisíveis como é ilustrado na Figura 2.9.

```
//Bloco A
a = 0;
b = 100;

ciclo(cond){

      //Bloco B
      a += 20;
      b -= 10;
}
```

Figura 2.9 – Grafo de fluxo reduzível.

Neste exemplo o Bloco A é reduzível a dois blocos: A1: a = 0; e A2: b = 100;. O Bloco B também: B1: a+=20; e B2: b-=10;. A ideia passa pela criação de mais predicados que condicionem o fluxo entre os blocos reduzidos. Os predicados são: condFV e condFF (ver Figura 2.10). A condFV é uma condição que devolve sempre falso e que só executa o código que protege se obtiver verdadeiro (e.g., condição == verdadeiro, sabendo que condição será sempre falso). A condFF é uma condição que devolve sempre falso e que executa o código que protege se obtiver falso (e.g., condição == falso, sabendo que condição será sempre falso). Após a transformação obtemos o código ilustrado na Figura 2.10.

```
a = 0;
if(condFV) then{
            ciclo(cond){
                        a += 20;
                        if (condFV)
                                    b = 100;
                        else
                                    b -= 10;
            }
}else{
            b = 100;
            ciclo(cond){
                        a += 20;
                        if(condFF)
                                    b -= 10;
                        else
                                    b = 100;
            }
}
```

Figura 2.10 – Grafo de fluxo não reduzível.

O resultado desta transformação aumenta o leque de possibilidades, fazendo parecer que o fluxo tem vários caminhos possíveis. Na realidade a funcionalidade contida no resultado desta transformação é exactamente a existente no código original, pois os predicados adicionados direccionam sempre o fluxo de execução para onde é desejado. Quanto ao custo de execução do

código resultante, este depende apenas da computação dos predicados adicionados. O resultado da transformação tem maior potência que o código original e oferece nesta solução, resistência a técnicas automáticas de inversão que removam indiscriminadamente predicados que identifiquem como sendo irrelevantes (pois o valor que devolvem é sempre o mesmo), quebrando assim a funcionalidade do código (ver Figura 2.11 e comparar com Figura 2.9).

```
//Bloco A (completo)
a = 0;
b = 100;

ciclo(cond){

        //Bloco B (incompleto)
        a += 20;
        b = 100;
}
```

Figura 2.11 – Remoção indiscriminada de predicados opacos introduzidos pela ofuscação quebra a funcionalidade do código.

2.5.4. Processamento Paralelo

Em sistemas multi-processador é sensato tirar partido de processamento em paralelo, dividindo tarefas (antes em sequência) e possibilitando que essas tarefas executem ao mesmo tempo em processadores diferentes. Existe assim, uma redução do tempo necessário para efectuar o processamento do mesmo número de tarefas, logo um aumento de performance.

Neste caso em particular, o que é interessante, é a ofuscação que a execução em paralelo pode trazer com a divisão de tarefas (ou adição de novas tarefas irrelevantes) por vários processadores, e não o aumento de performance. A resistência à análise estática por técnicas automáticas cresce com esta transformação. Isto deve-se ao crescimento exponencial das possibilidades de caminhos de execução com o crescimento do número de novos processos em execução [4]. Quando as tarefas não se podem dividir em execuções independentes e paralelas (pois possuem dependências entre elas), a execução em sequência é obrigatória. No entanto, isto não impossibilita uma abordagem semelhante se a linguagem de programação utilizada permitir a criação de *threads* sincronizada [4]. Sempre que possível, será preferível a utilização de *threads* em detrimento de processos, pois a troca de contexto é menos custosa, sem a perda de potência e resistência de ofuscação.

2.5.5. *Inlining* e *Outlining* de Funções

O *inlining* de funções consiste na substituição da chamada a uma função pelo seu corpo. Imaginemos que existem duas funções e dentro de uma dessas funções (`F2`) existe uma chamada a outra função (`F1`). Poderíamos eliminar a função `F1` e substituir a sua chamada na função `F2` pelo seu corpo como ilustrado na Figura 2.12a. O *outlining* (ver Figura 2.12b) é exactamente o oposto. Retiram-se dessa função um grupo de elementos contíguos e cria-se uma nova função com essa

sequência de elementos (F1' na Figura 2.12b). A nova função (F1') é chamada no mesmo local onde se encontrava inicialmente. Estas transformações são completamente resistentes à inversão e acrescentam alguma complexidade ao código, pois dificultam a compreensão da funcionalidade das funções ao juntar ou dividir os seus corpos.

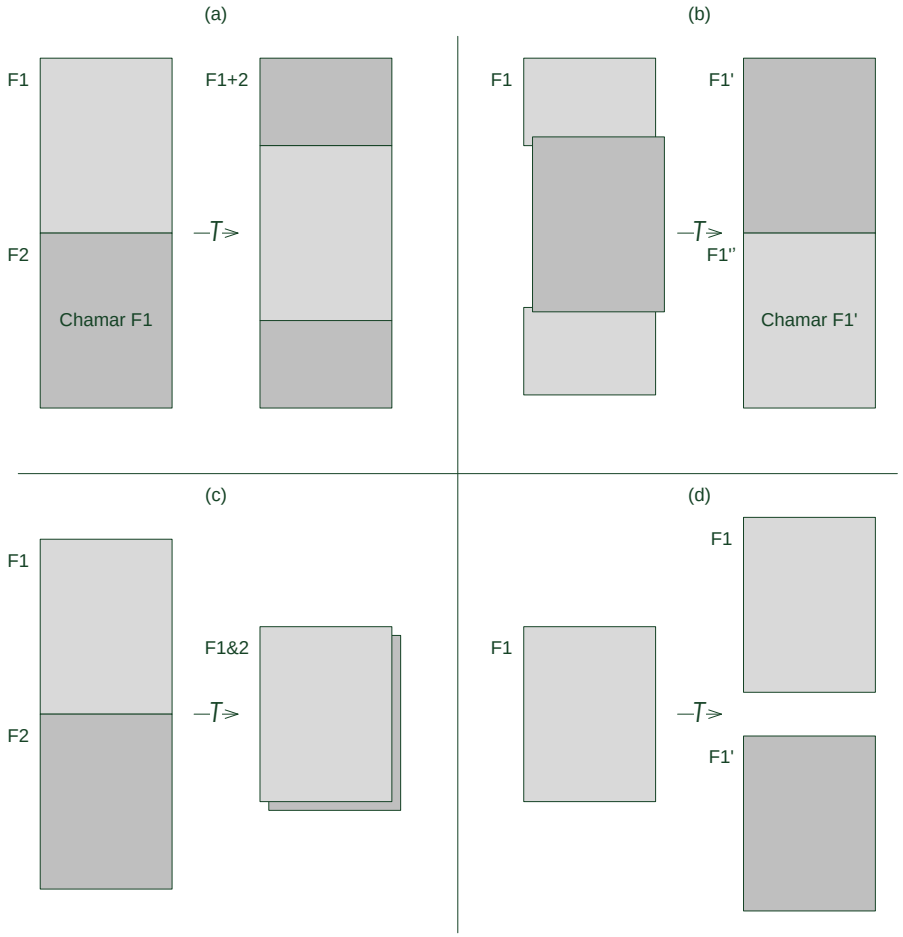


Figura 2.12 – (a) *Inlining* de funções; (b) *Outlining* de funções; (c) Fusão de funções; (d) Clonagem funções.

2.5.6. Fusão e Clonagem de Funções

Fusão de funções (ver Figura 2.12c) consiste na junção de duas ou mais funções numa só, fazendo-se a fusão dos corpos das diferentes funções, a junção dos seus argumentos e adicionando um novo argumento que faça a selecção de qual a função a executar. Idealmente a fusão de funções é feita em casos em que exista muito código comum e argumentos semelhantes entre funções. A clonagem de funções (ver Figura 2.12d) consiste na criação de cópias de uma função. Com diferentes combinações de técnicas de ofuscação obtêm-se diferentes cópias com a mesma funcionalidade (F1 e F1' têm a mesma funcionalidade). O objectivo da criação de um maior número de funções com a mesma funcionalidade será acrescentar esforço à análise feita para a engenharia inversa. Com chamadas às várias cópias aleatoriamente parecerá que diferentes pedidos estão a ser efectuados.

2.5.7. Transformação de Ciclos

Existem algumas transformações possíveis de se aplicarem aos ciclos. As razões que levaram à criação das mesmas passam quase sempre por necessidade de contornar limitações físicas de um sistema, ou por questões de performance e não para complicar a compreensão do código. O *loop blocking* (ver Figura 2.13a), o desenrolamento de ciclo (ver Figura 2.13b) e a rotura de ciclo (ver Figura 2.13c), são algumas das transformações possíveis. A ideia da rotura de ciclo é dividir o ciclo inicial em diferentes ciclos partindo o conteúdo de um ciclo em vários. O desenrolamento agrupa duas ou mais iterações do ciclo numa só iteração. O *loop blocking* consiste na divisão do espaço de iteração em blocos mais pequenos, aumentando assim a profundidade dos ciclos. Esta transformação foi concebida por razões de optimização da computação mas tem aplicação neste caso porque incrementa a potência da ofuscação. Todas as transformações incrementam a potência da ofuscação pois aumentam a profundidade dos ciclos, ou o número de predicados e expressões utilizadas.

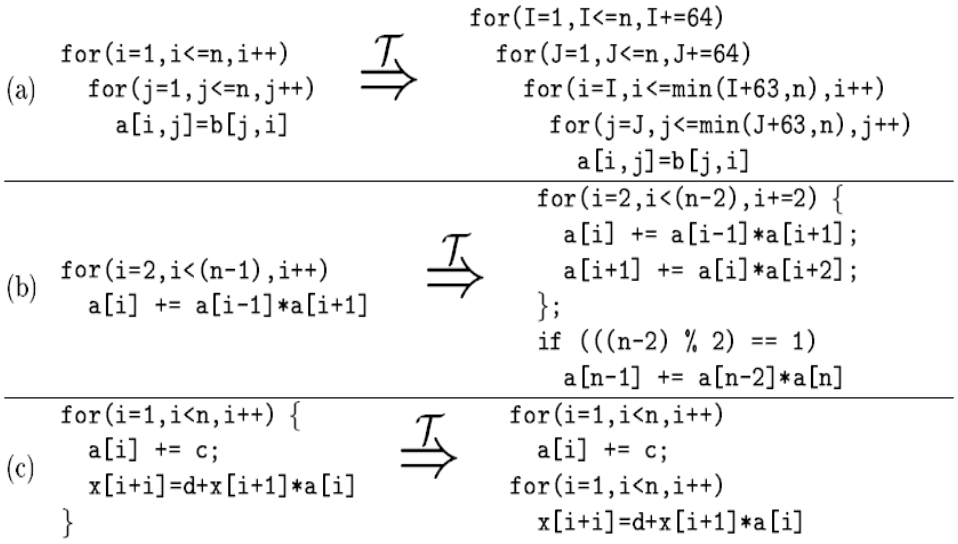


Figura 2.13 – Exemplos de transformações de ciclo obtidos em Collberg [4]: (a) *loop blocking*; (b) desenrolamento de ciclo; (c) rotura de ciclo.

2.5.8. Reordenação de Elementos

A organização de elementos no código (e.g., declaração de variáveis, posição de variáveis em expressões, blocos de código numa função, a disposição de funções no corpo do programa) é prática comum na programação, pois facilita a compreensão tanto por quem o lê, como por quem o desenvolve. A reordenação de elementos aumenta a resistência à inversão automática de ofuscação – muitas vezes sendo mesmo impossível a inversão, pois não existirem pistas que indiquem qual a disposição original. Por si só, esta técnica não é a melhor opção para aumentar a complexidade do código, necessitando da aplicação de outras técnicas em conjunto, para aumentar a potência de ofuscação. Esta reordenação não poderá ser efectuada indiscriminadamente, só sendo opção, quando não existem dependências entre elementos.

2.5.9. Adição de Operandos Redundantes

À semelhança da substituição de números esta transformação altera a computação, mas neste caso, adicionando novos operandos. Estes, são variáveis opacas, e em nada irão alterar o valor final da computação. Servem apenas para aumentar a potência da ofuscação e se forem suficientemente resistentes, não serão removidos por analisadores automáticos, tornando mais custosa a prática de engenharia inversa. O custo de execução do código resultante depende da computação acrescida com a adição dos novos operandos.

2.5.10. Divisão de Variáveis

A divisão de variáveis aplicada a variáveis de dimensão limitada é mais uma das possibilidades de ofuscação das estruturas de dados. Uma variável booleana (verdadeiro ou falso) pode ser dividida em duas ou mais variáveis que, posteriormente combinadas, devolverão o mesmo valor da representação original. Para que a nova representação seja possível, é necessário criar uma função f (p, q) que recebendo os valores de p e q, devolva o valor da variável original V, uma outra função g (V), que recebendo o valor da variável original V, devolva os valores das variáveis p e q, e novas representações das operações possíveis aplicadas às variáveis resultantes da divisão (p e q). Na Figura 2.14 estão ilustrados os resultados da aplicação dessas funções. Para percebermos melhor esta divisão analise-se um exemplo apresentado por Collberg [4], em que a variável original é do tipo booleano (variável V) e o resultado da sua divisão são duas variáveis do tipo inteiro (p e q). O valor da variável booleana V quando falso, corresponde aos valores das variáveis p e q serem iguais. O valor da variável booleana V quando verdadeiro, corresponde aos valores das variáveis p e q diferentes. A representação da variável V num número inteiro é dada pela expressão 2p + q. Esta expressão fornece os valores para as tabelas com as operações booleanas (e.g., AND, OR, XOR, NOT) aplicadas aos valores. As tabelas para o AND e o OR, encontram-se ilustradas na Figura 2.15.

$g(V)$		$f(p,q)$	
p	q	V	$2p+q$
0	0	False	0
0	1	True	1
1	0	True	2
1	1	False	3

(a)

Figura 2.14 – Representação possível, proposta por Collberg [4], da divisão de booleanos.

AND[A,B]	A=0	A=1	A=2	A=3
B=0	3	0	0	0
B=1	3	1	2	3
B=2	0	2	1	3
B=3	3	0	0	3

(c)

OR[A,B]	A=0	A=1	A=2	A=3
B=0	3	1	2	3
B=1	1	1	2	2
B=2	2	2	1	1
B=3	0	1	2	0

(d)

Figura 2.15 – Tabelas com os valores da aplicação de operações booleanas (AND e OR) às novas variáveis criadas pela transformação. Exemplo retirado de Collberg [4].

A Figura 2.16 apresenta alguns exemplos de diferentes representações após a aplicação da divisão. Vejamos por exemplo as atribuições (3) e (4) da Figura 2.16 que representavam o valor booleano false com as novas representações (b1,b2) = (0,0) e (c1,c2) = (1,1). As atribuições (5) e (6) da Figura 2.16 têm representações completamente diferentes depois da transformação, quando originalmente eram iguais.

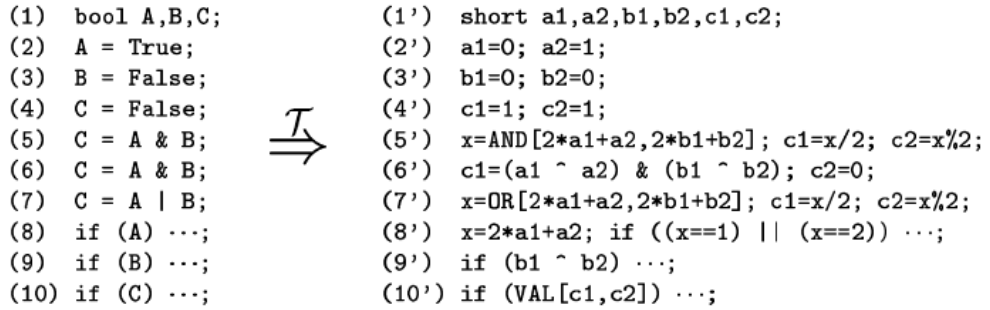


Figura 2.16 – Resultados da aplicação da transformação de ofuscação: divisão de variáveis. Exemplo retirado de Collberg [4].

2.5.11. Fusão de Variáveis Escalares

A fusão de variáveis escalares consiste na junção da informação de variáveis numa variável de tamanho superior que consiga albergar a informação de todas as variáveis sem sobreposição. Imagine-se a existência de duas variáveis de tamanho igual a 32 bits (e.g., int v1 e int v2). Com a utilização de uma variável de 64 bits (e.g., long v) podemos albergar a variável v1, nos 32 bits mais significativos da variável v e a v2 nos 32 bits menos significativos, por exemplo. O acesso e alteração dos dados seriam feitos de acordo com a fórmula $v(v1, v2) = 2^{32} \times v1 + v2$. Outra forma de fusão de variáveis escalares apresentada por Collberg [4] é a utilização de um vector que alberga as variáveis do mesmo tipo. Estas seriam acedidas posteriormente pelo índice do vector.

2.5.12. Conversão de Dados Estáticos em Dados Criados Dinamicamente

Os dados estáticos, principalmente cadeias de caracteres, contêm informação útil para a análise da funcionalidade do código [4]. Esta transformação de ofuscação consiste na divisão da informação representada estaticamente no código, por fracções dessa informação agrupadas dinamicamente em tempo de execução.

2.5.13. Reestruturação de Vectores

Existem várias formas de reestruturar um vector. É possível parti-lo em vários vectores, fundir vários vectores num só, transformar um vector de uma dimensão num vector de duas dimensões ou vice-versa. Quando um programador cria um vector ou qualquer outra estrutura de dados, fá-lo com o objectivo de representar da melhor forma (em código) a abstracção do problema que está a tentar resolver. A aplicação de qualquer uma destas transformações destrói essa representação, dificultando o trabalho do analista na compreensão da funcionalidade do código.

2.5.14. Modificação de Relações de Herança

A modificação de relações de herança é uma transformação exequível no contexto da programação orientada a objectos. Este paradigma de programação possui um elemento chamado classe, que representa a abstracção de um conceito, concebido para albergar variáveis e métodos que estejam relacionados com esse mesmo conceito – propriedades da classe. As classes podem estar associadas entre si por ligações de herança ou estarem simplesmente contidas em outras classes (*nested classes*). As ligações de herança criam um esquema que organiza as classes por grau de parentesco, transportando numa super-classe (pai), as propriedades comuns entre classes irmãs (filhas). A ideia é aproveitar este tipo de associação entre classes, não como forma de organizar as classes, mas para tornar o seu esquema, o mais complexo possível. Alguns exemplos de transformações passam pela divisão de uma classe em classes com ligações de herança entre si, adição de classes intermédias, reunião de classes sem propriedades comuns (que não façam parte do mesmo conceito) criando uma super-classe em que as variáveis do mesmo tipo se fundam e onde sejam inseridas cópias de métodos das classes filhas.

2.6. Técnicas de Anti-Depuração

Depuração é o processo de procura e redução de erros (*bugs*) no código do programa. Neste caso em particular, ao dizer-se que o código ofuscado está a ser alvo de depuração não se está a dizer que se procuram erros no código, mas sim, a dizer que se procura código não necessário ou cuja existência é fruto da ofuscação. Uma ferramenta de depuração auxilia o programador na monitorização de um programa em execução – possível análise a nível das instruções em tempo de execução [18] –, fornecendo a possibilidade de controlar essa execução, terminando-a e recomeçando-a em qualquer ponto (*breakpoints*). Estas ferramentas podem ser utilizadas numa tentativa de contrariar as técnicas de ofuscação, ajudando a simplificar e compreender o código ofuscado. Sendo assim, para além da tentativa de maximização da qualidade de ofuscação, é necessária a aplicação de técnicas de anti-depuração. A anti-depuração não é mais do que a implementação de mecanismos que contrariam o sucesso das ferramentas de depuração e consequentemente o sucesso da engenharia inversa. Algumas das verificações que estas técnicas tentam implementar são, por exemplo, a detecção de alteração de código, detecção da utilização de *breakpoints* no programa, medição do tempo entre instruções no código, a utilização de registos reservados para depuração, entre outros. De seguida são apresentadas as técnicas utilizadas para contrariar as ferramentas de depuração.

2.6.1. Baseadas na API

Os mecanismos de anti-depuração mais simples de usar são aqueles que podem ser obtidos directamente na API do sistema operativo ou numa biblioteca de funções. Esses mecanismos são funções que questionam os processos e o sistema para determinar se uma ferramenta de depuração está a ser utilizada. Um exemplo mais básico de uma dessas funções, é a função IsDebuggerPresent que se pode encontrar na biblioteca do MSDN [19]. O IsDebuggerPresent verifica se o processo que chama esta função está a ser alvo de depuração por uma ferramenta de

depuração em modo de utilizador. Esta verificação é feita consultando uma *flag* no processo que indicará estar a ser alvo de depuração, sempre que o seu valor for diferente de zero. Outras funções podem ser encontradas na mesma biblioteca. Estas funções são tão fáceis de usar e perceber como são de anular. Também existem soluções mais avançadas como funções internas do sistema operativo, em que o seu funcionamento é mais complicado de perceber pela pouca informação disponibilizada na API. Um exemplo de uma dessas funções é a NtQueryInformationProcess, também encontrada na biblioteca do MSDN [20]. Esta função obtém informação sobre o processo para o qual aponta. Com o primeiro argumento igual a -1 a função analisará o processo que a chama. Os outros argumentos servem para configurar a informação que pretendemos obter sobre o processo e verificar se este está a ser alvo de depuração.

2.6.2. Checksum

A criação de *checksums* através de um algoritmo criptográfico irreversível (e.g., MD5, SHA-1) poderá ser uma das técnicas a utilizar para verificar a integridade de parte ou da totalidade do código. O objectivo da criação destes *checksums* é a verificação, num determinado ponto do código, se este foi alterado. Essa detecção vai despoletar um mecanismo de defesa que contrariará o depurador. A forma mais simples de contrariar o depurador é terminar a execução do programa aquando da detecção da alteração. Contudo, esta não será a forma mais segura de contrariar a detecção, pois ao terminar o programa de imediato, denunciando a existência de um mecanismo de anti-depuração, como o local e as alterações que originaram a resposta desse mecanismo. Uma alternativa à terminação imediata do programa, é a execução de código irrelevante a partir do momento de detecção e consequente terminação do programa, num tempo posterior ao da detecção de depuração.

2.6.3. Meshed Integrity Control Points

Por mais difícil que seja detectar uma verificação de integridade no código de um programa, quando isso acontece basta removê-la para que seja anulada. Uma forma de impedir o sucesso após detecção é a utilização de um conjunto de verificações: *mesh techniques* [21]. Este conceito foi introduzido muito recentemente em sistemas de segurança e sugere a utilização de verificações interdependentes em vários níveis do sistema dos quais o programa dependa. Estas verificações podem utilizar informação contida na memória RAM, o *output* de uma função num determinado momento da execução de um programa, *logs* do programa, o próprio código do programa, em suma, qualquer informação que seja criada pelo programa ou da qual o programa dependa e que possa ser alvo de ataques que coloquem a integridade do programa em causa aquando da tentativa de *reverse engineering*. Com isto pretende-se que ao falhar uma das várias verificações interdependentes, as outras não sejam postas em causa e assim seja detectada a intrusão.

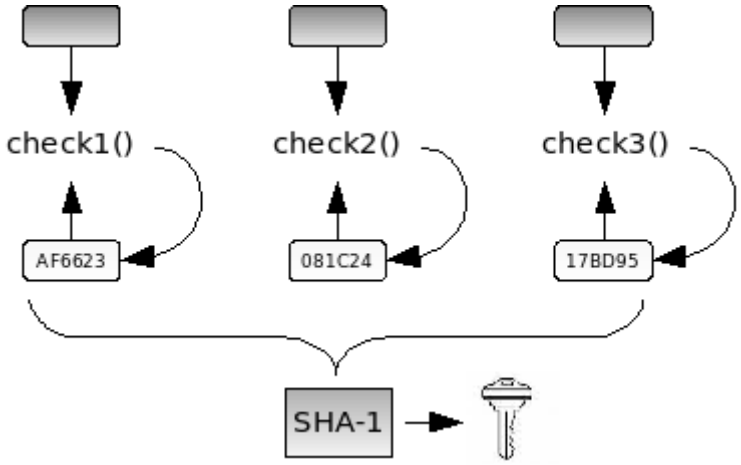


Figura 2.17 – *Hash-and-decrypt* apresentado em Lawson [22].

A *hash-and-decrypt* [22] é uma *mesh technique* que cria um *checksum* cuja existência não se justifica apenas para efeitos de verificação de integridade mas também, como chave de um algoritmo criptográfico reversível (e simétrico) para decifrar o próximo bloco de código antes de permitir a sua execução (ver Figura 2.17). Aqui a verificação de integridade é feita quando existe sucesso ao decifrar o próximo bloco de código, pois não seria possível decifrá-lo se alguma informação utilizada na criação do *checksum* tivesse sido alterada. Isto obriga que o atacante execute o programa até à sua conclusão ou que anule todas as verificações interdependentes para cada bloco.

2.6.4. Time-Checking

Time-checking é a comparação do tempo de execução com o tempo de execução estimado na tentativa de detectar a depuração. Imaginemos que foi adicionado código a um programa ou uma parte do código foi removida. Essa alteração irá ter consequências no tempo necessário à execução do programa, diminuindo ou aumentando o tempo em função da alteração. Esta verificação possibilita a detecção da inserção de *breakpoints* no programa por uma ferramenta de depuração, verificando se existem discrepâncias no tempo de execução.

Uma preocupação na utilização do *time-checking* é a sua utilização em conjunto com técnicas de ofuscação. As técnicas de ofuscação podem modificar a forma e estrutura do código e, consequentemente, o custo de computação do mesmo. Isto traz problemas na verificação do *time-checking*, e se não for tomado em consideração, a própria ofuscação poderá ser confundida como sendo um ataque de depuração. Uma possível abordagem a este problema implica calcular os tempos de execução esperados antes da ofuscação e fazer posteriormente o acerto desses tempos de acordo com o custo adicionado por cada transformação de ofuscação aplicada ou (mas menos aconselhável) adicionar os *time-checking* posteriormente à ofuscação.

2.6.5. Blocos de Processos e *Threads*

O bloco de ambiente de processos (conhecido no *Windows* como *PEB Structure*) é uma estrutura de dados que contém informação sobre os processos. Para averiguar se uma ferramenta de depuração está a analisar um processo, procura-se directamente no *PEB* se a *flag* que identifica a depuração está activa. Isso pode ser conseguido, por exemplo, com a utilização da função NtQueryInformationProcess [20], contendo como segundo argumento o ProcessBasicInformation. Verificar directamente na PEB (ou TEB) se o processo (ou *thread*) está a ser alvo de depuração é uma solução mais credível do que utilizar uma função directamente da API, pois o seu resultado pode ser manipulado pelo próprio depurador, para nos levar a acreditar que o processo (ou *thread*) não está a ser alvo de depuração.

2.6.6. Baseadas em Excepções

As técnicas de anti-depuração baseadas em excepções tiram partido da criação voluntária de excepções para detectar a acção de uma ferramenta de depuração. Quando uma ferramenta de depuração encontra uma computação que devolve uma excepção, tentará tratá-la (na tentativa de não colocar em causa a continuidade da análise), não a devolvendo ao processo em execução que está a analisar. A ideia é usar isso contra a depuração, criando computações que devolverão excepções propositadamente e tentar tratá-las no processo com funções para esse efeito (e.g., função SetUnhandledExceptionFilter da biblioteca do MSDN [23], disponibilizadas pela API do sistema operativo. Se o processo apanha a excepção, resolve-a e marca uma variável indicando que a excepção foi apanhada como era suposto. Se essa variável não for marcada é porque o tratamento da excepção foi feito por uma ferramenta de depuração. Quando o processo detecta um possível ataque poderá, por exemplo, terminar de imediato ou mudar o fluxo de execução para código irrelevante. O sucesso desta técnica estará sempre dependente da sensibilidade da ferramenta de depuração para este tipo de defesa.

2.6.7. Baseada em Hardware e Registos

As ferramentas de depuração utilizam registos disponibilizados pelo processador (numa arquitectura x86 são os registos DR0 a DR7) para criar *breakpoints* de hardware [24]. A ideia será criar um programa que utilize estes registos (criando entraves à sua utilização na engenharia inversa) ou apenas detectar a sua utilização por uma potencial ferramenta de depuração. A melhor opção é a manipulação dos registos, pois garante mais facilmente o sucesso do objectivo de anti-depuração [25]. Contudo, esta técnica não é fácil de implementar, pois estes registos não estão disponíveis pelas instruções normais em modo de utilizador.

2.7. Ataques ao Código Ofuscado

O ataque ao código ofuscado é a tentativa de descoberta daquilo que a ofuscação tenta esconder: a funcionalidade. A ofuscação de código não garante o secretismo absoluto da sua funcionalidade, só atrasa a sua compreensão. A utilização em conjunto de técnicas manuais e automáticas vão

garantir eventualmente a inversão da ofuscação, mas com um custo associado – em tempo e espaço – que pode variar até ao ponto em que deixa de ser rentável. Para travar estes ataques é necessário conhecê-los e descobrir quais são os piores casos possíveis de análise para cada um, tentando recriá-los na ofuscação. O custo que aqui se menciona não é o mesmo que foi apresentado no capítulo anterior. Esse representa o custo de computação adicionado ao código no resultado da ofuscação. O custo que aqui é mencionado é aquele que resulta da aplicação das técnicas de inversão da ofuscação. O primeiro custo deverá ser minimizado e o segundo (custo da inversão de ofuscação) maximizado.

2.7.1. Engenharia Inversa

A engenharia inversa (em inglês, *reverse engineering*) é o processo de análise da estrutura e funcionamento de uma máquina (sem a alterar) cujo objectivo é perceber o que faz ou, a partir da identificação do que faz, perceber como faz. Um exemplo mais concreto é a sua aplicação na análise de software. O interesse dessa análise pode variar, e.g., pode ser feita com o objectivo de criar uma solução semelhante à original (apenas na funcionalidade) para economizar tempo de desenvolvimento ou simplesmente, para satisfazer a curiosidade ou por constituir um desafio. A ofuscação de código tenta contrariar este ataque alterando a forma como o código se apresenta (e.g., codificação, encriptação) e transformando profundamente a sua estrutura, sendo assim mais difícil para um *reverse engineer* conseguir perceber a sua funcionalidade. Para além disso o *reverse engineering* também pode ser utilizado para tentar perceber o funcionamento das técnicas de ofuscação a partir do código ofuscado. Sendo assim, deverá existir uma tentativa de maximização da potência de ofuscação e da sua qualidade de não detecção, para contrariar tanto a compreensão da funcionalidade do código, como a compreensão da ofuscação.

2.7.2. Identificação e Avaliação de Elementos Opacos

Um elemento opaco é um predicado (expressão booleana) ou variável irrelevante cujo resultado é conhecido em tempo de ofuscação mas dificilmente deduzido por um mecanismo automático de inversão da ofuscação. Uma boa parte das transformações de ofuscação depende da qualidade de não detecção e da dificuldade introduzida pelas variáveis e predicados opacos - aos mecanismos automáticos de inversão de ofuscação - na identificação da sua irrelevância. Estes elementos são utilizados para proteger o conhecimento do fluxo de execução num programa (e.g., a condição de terminação de um ciclo, a condição de salto), oferecendo possibilidades de fluxo irrelevantes. São usados também para complicar expressões, adicionando novas variáveis que não alteram o valor do seu resultado. A análise do código tenta identificar estes elementos, verificando se a sua existência é prescindível, e se o for, tenta avaliar a possibilidade de os eliminar.

Uma forma de dificultar a identificação e avaliação destes elementos é tornando-os o mais dependentes possível de outras computações (não locais). Isso pode ser conseguido pela distribuição de computações por outras chamadas que são feitas ao longo do programa. Um exemplo disso é a utilização de *threads* [4], onde essas computações possam ser inseridas, aproveitando assim, a complexidade que é acrescentada à análise quando as dependências passam pela utilização de processamento paralelo. Outra forma de dificultar a avaliação destes elementos

é com a utilização de predicados cujo resultado não consegue ser avaliado numa análise estática do código (i.e., sem executar o programa). Para tal, podem ser utilizadas condições que utilizem o resultado pré-calculado de uma função de *hash* [13].

2.7.3. Identificação de Padrões

A identificação de padrões é uma técnica utilizada para inverter a ofuscação, identificando no código que analisa, padrões semelhantes aos que registou como sendo possíveis resultados de transformações de ofuscação. Este tipo de ataque não se evita com técnicas de ofuscação pouco resistentes ou facilmente detectáveis. É necessária uma boa combinação de técnicas para evitar o sucesso deste tipo de ataque (evitar a utilização repetida de *templates* conhecidos).

2.7.4. Fatiar Código

Algumas transformações metamórficas deslocalizam blocos de código que estariam logicamente relacionados no código original ou intercalam-no com código irrelevante. Estas ferramentas identificam os blocos de código cujos resultados das suas computações contribuem para o resultado de outra computação algures no programa (identificando todas as dependências). A junção de blocos relacionados criará uma fatia de código que na versão original haveria de se encontrar agrupada. Contrariar este mecanismo passa por aumentar o custo necessário à identificação dessas fatias, aumentando ainda mais o número de dependências entre variáveis, para que o mecanismo leve mais tempo a criar cada fatia de código.

2.7.5. Análise Estatística

Esta análise consiste no teste intensivo de um programa, executando-o várias vezes na tentativa de identificar predicados que devolvam sempre o mesmo resultado. Depois de um número de execuções considerável, se o resultado de um predicado é sempre o mesmo, existe uma grande probabilidade de ser um predicado opaco adicionado pela ofuscação. Uma outra forma de analisar se um predicado foi ou não introduzido pela ofuscação será executando simultaneamente duas versões do código ofuscado. Uma versão intacta e outra cujo predicado tenha sido substituído pelo resultado que se pensa ser sempre devolvido. Verifica-se então se com a mesma entrada em ambas as cópias, se obtém a mesma saída. Resultados semelhantes num número considerável de testes, denuncia a possibilidade do predicado ser irrelevante.

Existem algumas formas de dificultar o sucesso da análise estatística. Através da utilização de predicados opacos mas cujo resultado é aleatório. Com esta alternativa obtém-se o mesmo efeito, sem padecer da vulnerabilidade anterior. A utilização destes predicados exige no entanto, que diferentes caminhos devolvidos pelo predicado apontem para blocos de código com a mesma funcionalidade (ofuscados de forma diferente para não ser óbvia a irrelevância da sua existência). Outra forma é o desenvolvimento de predicados cuja remoção ou alteração tenha efeitos secundários no programa. Segue-se um exemplo disso, apresentado por Collberg [4].

A Figura 2.18 ilustra a transformação de uma sequência de dois blocos de código (`S1` e `S2`) numa representação em que a execução dos mesmos blocos está dependente da avaliação dos

predicados. Os dois predicados têm de ser executados o mesmo número de vezes, caso contrário ocorrerá um *overflow* e terminará o programa (tipo int apenas suporta valores entre -2^{31} e $2^{31}-1$).

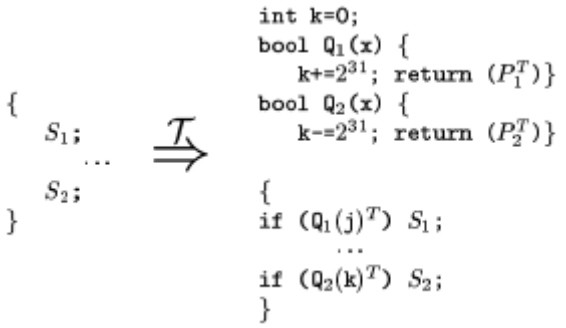


Figura 2.18 – Exemplo apresentado por Collberg [4] que ilustra a dependência da existência dos predicados opacos introduzidos pela ofuscação para a funcionalidade do código.

2.8. Ofuscação de Javascript

Javascript é uma linguagem de *scripting* utilizada para o desenvolvimento de aplicações Web que executam do lado do cliente. É uma linguagem interpretada, querendo isto dizer que o código não é compilado antes de ser executado. A ferramenta que se propõe criar neste projecto irá ofuscar código *Javascript*. A ofuscação é aplicada directamente no código fonte ao contrário do que acontece por exemplo com o *Java* ou *dotNET*, em que a ofuscação é aplicada ao *bytecode*.

O código *script* tem sido utilizado com objectivos maliciosos: para instalar *malware* através de um navegador sem o conhecimento do utilizador – também conhecido por *drive-by download* [26] –, para expor o utilizador a *spam* ou para roubar informações relacionadas com o cliente. Na maior parte dos casos, o objectivo do criador de um programa malicioso é a maximização do impacto que esse terá (no maior número possível de máquinas). Uma forma de tentar assegurar que isso acontece é atrasando a análise e detecção do seu programa através da utilização de ofuscação. Nos dois capítulos seguintes serão apresentadas as técnicas de ofuscação, as ferramentas, e as técnicas de análise aplicadas a *Javascript malware*.

2.9. Técnicas de Ofuscação Aplicadas a Javascript Malware

Existem várias técnicas apresentadas nos capítulos anteriores que podem ser aplicadas a código *Javascript*. Nesta secção apresentam-se as técnicas que são comummente utilizadas no *Javascript* ou que foram recentemente propostas.

2.9.1. (Des)codificação: Função unescape

A função escape do *Javascript* recebe uma cadeia de caracteres e codifica-a na notação %xx (representação hexadecimal de um carácter). A função unescape do *Javascript* faz exactamente o contrário, recebe uma cadeia de caracteres codificada na notação anterior e passa-a para a sua representação em caracteres. Estas funções podem ser utilizadas na ofuscação de código *Javascript*. O processo passa pela codificação em tempo de ofuscação e posteriormente em tempo de execução, com a inversão da codificação utilizando a função unescape. O código, depois de invertida a ofuscação, é adicionado à página com a utilização do método document.write[3] para ser então executado. Vejamos o seguinte exemplo (Figura 2.19), que representa a utilização desta técnica.

```
document.write(unescape("%3CHEAD%3E%0D%0A%3CSCRIPT%20 … %0A"));
```

Figura 2.19 – Descodificação da cadeia de caracteres e posterior execução.

Depois de executarmos o código anterior surge mais código onde se pode encontrar outra chamada à função unescape com mais código na mesma notação para ser descodificado novamente (ver Figura 2.20).

```
<SCRIPT LANGUAGE="Javascript">
<!—
document.write(unescape("%0D%0A%3Cscript%20 … %3C/html%3E"));
//-->
</SCRIPT>
```

Figura 2.20 – Segunda camada de codificação com unescape.

O processo de inversão da ofuscação com o unescape será feito tantas vezes, quantas as vezes em que o código foi codificado. A utilização repetida desta função tão trivial é justificada pela elevada resistência a *signature-based detection* e *anomaly-based intrusion detection*[4] que oferece ao código [26]. Outras notações podem ser usadas, e.g., Unicode, UTF-8, Octal, sendo a utilizada apenas um exemplo.

2.9.2. (Des)codificação: Algoritmos Simples

A ofuscação do código através de codificação surge com o *malware* para dificultar a detecção e compreensão do código malicioso. São frequentemente encontrados em *Javascript* ofuscado com

[3] O método document.write() é utilizado para escrever expressões HTML ou código Javascript na página Web.

[4] Classificação e detecção de actividades intrusivas e uso abusivo num sistema. A classificação é feita através de regras que definem os comportamentos expectáveis no sistema.

codificação, endereços que redireccionam o utilizador para páginas que contêm *spam*. O exemplo anterior faz a codificação do código alterando a sua representação em caracteres para hexadecimal, mas a mesma continua vulnerável a *pattern recognition* por *machine learning*[5] [27]. A alternativa apresentada é a codificação e descodificação através de algoritmos simples construídos para o efeito.

Dois exemplos do que poderá ser usado como técnica de ofuscação neste projecto são apresentados na Figura 2.21 e Figura 2.22.

```
function Decode(){
 var temp="",i,c=0,out="";  var
str="60!115!99!114!105!112!116!32!108!97!110!103!117!97!103!101!61!34!74!9
7!118!97!83!99!114!105!112!116!34!62!13!10!32!32!32!32!32!118!97!114!32!97
!49!61!39!119!105!110!100!39!44!32!13!10!32!32!32!32!32!97!50!61!39!111!11
9!46!108!111!99!97!39!44!13!10!32!32!32!32!32!97!51!61!39!116!105!111!110!
46!114!101!39!44!13!10!32!32!32!32!32!97!52!61!39!112!108!97!99!39!44!32!1
3!10!32!32!32!32!32!97!53!61!39!101!40!34!104!116!116!39!44!32!13!10!32!32!
32!32!32!97!54!61!39!112!58!47!47!39!44!32!13!10!32!32!32!32!32!97!55!61!39
!117!108!116!114!97!45!110!101!116!46!39!44!32!13!10!32!32!32!32!32!97!56!
61!39!105!110!102!111!47!116!105!99!107!101!116!47!115!101!97!114!39!44!3
2!13!10!32!32!32!32!32!97!57!61!39!99!104!46!112!104!112!63!39!44!32!13!10!
32!32!32!32!32!97!49!48!61!39!113!61!39!44!32!13!10!32!32!32!32!32!97!49!49
!61!39!67!104!101!97!112!43!65!105!114!102!97!114!101!34!41!39!59!13!10!32
!32!32!32!32!118!97!114!32!105!44!115!116!114!61!34!34!59!32!102!111!114!4
0!105!61!49!59!105!60!61!49!49!59!105!43!43!41!32!123!32!115!116!114!32!43
!61!32!101!118!97!108!40!34!97!34!43!105!41!59!32!125!13!10!32!32!32!32!32!
101!118!97!108!40!115!116!114!41!59!13!10!60!47!115!99!114!105!112!116!62!
";
   l=str.length;
   while(c<=str.length-1) {
      while(str.charAt(c)!='!')
         temp=temp+str.charAt(c++);
      c++;out=out+String.fromCharCode(temp);temp="";
   }
   document.write(out);
}
                              http://cheap-airfare-a.blogspot.com/
```

Figura 2.21 – Algoritmo simples de codificação apresentado por Chellapilla [27].

[5] Desenvolvimento de algoritmos que permitam a um computador aprender com informação (obtida através, e.g., de um sensor ou base de dados) e a reconhecer padrões e tomar decisões autonomamente.

```
var tt, kk="", mm;
tt="w|nd^w$|^c#[|^n;'([[*)!!*r^|^n$|nf^!f>>d!s>#rc($*(*]q;c(>#*+c|g#r>[>s'";
for (i=0; i<tt.length+1; i++)
{
   mm=tt.substring (i,i+1);
   if (mm=="(") mm="h";   if (mm=="*") mm="p";   if (mm=="!") mm="/";
   if (mm==">") mm="e";   if (mm=="$") mm=".";   if (mm=="[") mm="t";
   if (mm=="#") mm="a";   if (mm=="^") mm="o";   if (mm=="]") mm="?";
   if (mm=="@") mm="k";   if (mm=="{") mm="&";   if (mm==")") mm=":";
   if (mm==";") mm="=";   if (mm=="|") mm="i";   if (mm==" ") mm="+";
   kk=kk+mm;
}
eval (kk);
http://cheap-cigaretes-2007.blogspot.com/2006/11/cheap-cigarette-online-
cheap-cigarette.html
```

Figura 2.22 – Segundo algoritmo simples de codificação apresentado por Chellapilla [27].

2.9.3. Eval e Arguments.callee

A função eval recebe como argumento uma cadeia de caracteres que avalia e executa como sendo código *Javascript*. Habitualmente é uma função evitada no desenvolvimento de código *Javascript*, pois o seu uso é custoso e facilmente se encontram alternativas. No entanto, é uma mais-valia para a ofuscação, pois oferece a possibilidade de criar código dinamicamente, dificultando a análise estática.

Uma função chama o arguments.callee sempre que quiser fazer referência a si mesma. É utilizada habitualmente para criar chamadas recursivas a funções anónimas que não se conseguem referenciar a si mesmas por não terem nome. Na ofuscação de *Javascript* tem sido utilizada para detectar a alteração de código. O exemplo da Figura 2.23 ilustra a utilização de eval juntamente com o arguments.callee.

```
function r(str,t) {

   for(var sa=0; sa<str.length; sa+=arguments.callee.toString().length-444)
   {
      // Faz a descodificação de str, preparando o código para a chamada eval;
   }
   eval(ii);
};
```

Figura 2.23 – Exemplo da utilização da chamada eval e arguments.callee.

Neste exemplo existe uma contagem do número de caracteres na função com arguments.callee.toString().length que é utilizada no cálculo do valor a incrementar ao valor da variável sa do ciclo. Tipicamente, para se visualizar o código que o eval executa, altera-se a sua

chamada pela chamada ao alert[6], mostrando assim, o código numa caixa de alerta em vez de o executar. Com esta alteração, o número de caracteres é aumentado em um valor (de três caracteres com eval para quatro com alert) e assim também é alterado o valor incrementado na variável sa do ciclo. Consequentemente, o código devolvido ao eval será uma sequência de caracteres sem lógica. Existe a possibilidade de o ciclo não terminar, se o código alterado tiver o tamanho de 444 caracteres (sa+=0). Depois de descobrir o funcionamento desta verificação é fácil contorná-la, pois é apenas necessário compensar estas alterações (feitas na análise) com o acerto do valor subtraído ao arguments.callee.toString().length (de forma a devolver o valor esperado) ou substituindo-o directamente pelo resultado (sa+=1).

2.9.4. (Des)codificação: XOR

Outra forma de codificar e descodificar uma cadeia de caracteres é com a aplicação do *ou exclusivo* aos caracteres (também conhecido por XOR). Com a aplicação do XOR a cada carácter e utilizando a mesma chave para codificar todos os caracteres será utilizada a mesma chave para os descodificar. O exemplo da Figura 2.24 ilustra a utilização desta técnica.

```
str = "ru`su)(:^L^Kgtobuhno!ru`su)(!z^L^Kw`s!fgg!<!
enbtldou/bsd`udDmdldou)&nckdbu&(:^L^Kfgg
rdu@uushctud)&he&-&fgg&(:^L^Kfgg/
rdu@uushctud)&bm`rrhe&&bm&*&rh&*#e;CE#*#87B4#*&47,74@
…
ubi)d(z| |";str2 = "";for (i = 0; i < str.length; i +
+) { str2 = str2 + String.fromCharCode
(str.charCodeAt (i) ^ 1); }; eval(str2);
```

Figura 2.24 – Exemplo de algoritmo de descodificação XOR.

O código descodifica a cadeia de caracteres aplicando um XOR com o valor de 1 a cada carácter (mesmo valor aplicado na codificação) antes de a executar com a chamada eval. Note-se que existem outras notações que podem ser alvo da codificação XOR (e.g., ASCII).

2.9.5. String Splitting

String splitting consiste na representação do código em pequenas cadeias de caracteres guardadas em variáveis que permitirão posteriormente a reconstrução e execução do código. A reconstrução da cadeia de caracteres é feita com a junção de todas as variáveis que contêm as parcelas do código, executada no final pela chamada eval (ver Figura 2.26). Apesar de ser uma técnica de ofuscação fraca quando aplicada isoladamente, pode ser combinada com outras técnicas para fortalecer a resistência a inversão de ofuscação.

[6] Chamada que mostra uma caixa de alerta no ecrã, cuja mensagem é o argumento que recebe.

```
document.write(\"<iframe src=/x.htm width=0
height=0></iframe>\");
```

Figura 2.25 – Código original a ofuscar pelo *string splitting*.

```
le="rame>\");";
ok="docume";
uk="eight=0></if";
aj="nt.write(\"";
em="dth=0 h";
cg="<ifram";
nr="e src=/x.htm wi";
eval(ok+aj+cg+nr+em+uk+le);
```

Figura 2.26 – Exemplo do resultado da aplicação do *string splitting* ao código apresentado na Figura 2.25.

2.9.6. Javascript Objects: Member Enumeration

Javascript [28] é uma linguagem orientada a objectos e como tal a sua API de funções está organizada de acordo com este paradigma de programação: numa hierarquia de classes. O acesso a uma função é feito percorrendo a hierarquia de classes até chegar à classe que a contém. Como alternativa à chamada de uma função (no conjunto de classes do HTML DOM [28]) utilizando a notação classe(ponto)função (e.g., window.alert) é proposta uma outra forma de invocação. Começa-se pela utilização do this, apontando para o objecto HTML DOM onde é chamado. Percorre-se a hierarquia de classes afectando os objectos seleccionados a variáveis para serem utilizadas posteriormente – em detrimento do nome da classe – numa notação equivalente à utilizada nos vectores (e.g. classe '[' função ']'). Vejamos a Figura 2.27 com a primeira parte da ofuscação da chamada document.write('p');.

Neste caso pretende-se fazer a chamada da função write existente no objecto document do HTML DOM. Para seleccionar a classe document temos de a procurar na hierarquia de classes. Começa-se pela classe mãe – window – que é referenciada através do this. Depois de seleccionada a classe mãe, temos de começar a procurar a segunda classe (e última), neste caso a classe document. A procura é feita com o ciclo for…in e a selecção da classe é feita com três verificações (sendo apenas uma das várias possibilidades): comprimento do nome da classe, primeiro carácter do nome e último carácter do nome. Quando encontrar a classe, associa o seu nome à variável, atribuindo à variável i, o objecto document.

```
h = this;
for (i in h)
{
  if(i.length == 8)
  {
     if(i.charCodeAt(0) == 100)
     {
       if(i.charCodeAt(7) == 116)
       {
          break;
       }
     }
  }
}
```

Figura 2.27 – Exemplo de Kolisar [29]: Atribuição do objecto window à variável h e do objecto document à variável i.

O passo seguinte (ver Figura 2.28) rege-se pela mesma lógica e atribui à variável j a função write. A chamada da função write passa a ser feita da seguinte forma: h[i][j]('p');.

```
for (j in h[i])
{
  if(j.length == 5)
  {
     if(j.charCodeAt(0) == 119)
     {
       if(j.charCodeAt(1) == 114)
       {
          break;
       }
     }
  }
}
```

Figura 2.28 – Exemplo de Kolisar [29]: Atribuição do método write pertencente ao objecto document à variável j.

2.9.7. Literal Hooking

Esta técnica de ofuscação envolve as constantes no código com saltos condicionais (na notação cond?op1:op2) e valores que servem apenas como distracção. Na Figura 2.29 podemos encontrar várias condições de salto com elementos em diferentes sistemas de numeração. Com esta nova representação não é tão óbvia a afectação do objecto document à variável aaa. Para tornar esta transformação mais resistente a inversão automática deverá ser introduzida aleatoriedade na escolha da posição do valor correcto e da quantidade de condições de salto, obrigando a efectuarem-se cálculos para se descobrir o valor correcto. Caso contrário, seria apenas necessário remover todos os caracteres que envolvem o valor correcto, pois este encontrar-se-ia sempre na mesma posição.

```
aaa=(((0x4435,7.)>=(.61,9.12e2)?(1,4.033e3):(266,7.1e1)),((0x97<=.1?7.616e3:2.176e3)
,(.39<8e0?document:2032)));
```

Figura 2.29 – Exemplo da utilização do *literal hooking* retirado de Zdrnja [40].

2.10. Ferramentas e Técnicas de Análise

Nas secções seguintes serão apresentadas as técnicas e ferramentas actualmente utilizadas para efectuar depuração de *Javascript malware,* conhecido por ser rico em conteúdo ofuscado. Existem várias redes sociais na Internet que partilham o mesmo interesse comum: depuração de *Javascript malware* e constante procura de novas técnicas e ferramentas de depuração.

As técnicas e ferramentas têm como objectivo manipular o código *Javascript* de forma a descobrir o que ele faz, por exemplo, substituindo chamadas a funções típicas de *Javascript Malware – telltale indicators* [29] - por funções que devolvem o código em detrimento de o executar. As ferramentas analisadas são: *Rhino* e *SpiderMonkey*.

2.10.1. Método Preguiçoso

Esta técnica – também conhecida por *The Lazy Method* [30] – é a mais trivial de todas, mas não deixa por isso de ser bastante útil. Consiste na substituição de chamadas a funções, que executam e inserem código numa página, por funções que mostram o código no ecrã sem o executar. O eval e o document.write, apresentados no capítulo anterior, são exemplos de funções a procurar e a substituir pela função alert. A função alert irá mostrar, numa caixa de alerta no navegador, o código que iria ser executado antes da substituição.

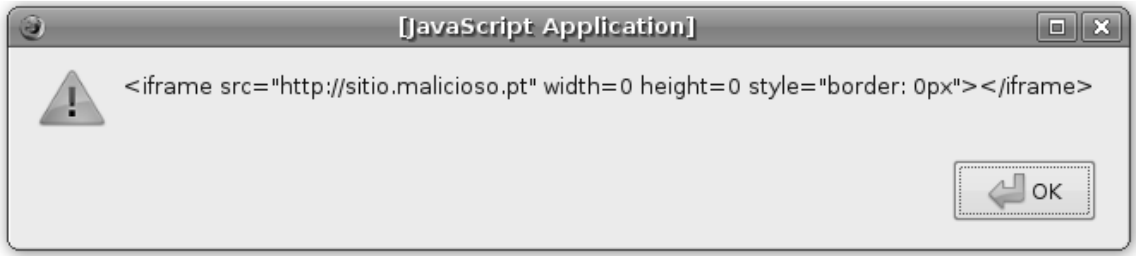


Figura 2.30 – Exemplo de código que iria ser executado mas que em vez disso foi apresentado no ecrã.

2.10.2. Utilização do Elemento Textarea

Esta técnica baseia-se na mesma ideia do método preguiçoso. Neste caso a substituição é feita, não pela função alert, mas pelo elemento (*tag*) HTML textarea [31]. A textarea irá construir uma caixa de texto no navegador, com o código como conteúdo (editável). Apesar de ser uma solução mais agradável do que o alert – porque o código fica imediatamente pronto a editar no

navegador – não é aconselhável a sua utilização, pois o navegador é um ambiente de depuração desprotegido. A Figura 2.31 apresenta um exemplo de código ofuscado.

A cadeia de caracteres vai ser descodificada com o XOR (chave 1) e será executada no final pela chamada eval. Se descodificarmos a cadeia de caracteres str obtemos o código apresentado na Figura 2.32.

```
str = "=.udyu`sd`?=hgs`ld!rsb<#iuuq;..rhuhn/l`mhbhnrn/qu#
!vheui<0!idhfiu<0!ruxmd<#cnseds;!1qy#?=.hgs`ld?";
str2 = "";

for (i = 0; i < str.length; i ++){

        str2 = str2 + String.fromCharCode(str.charCodeAt (i) ^ 1);

};
eval(str2);
```

Figura 2.31 – Cadeia de caracteres str descodificada e atribuida ao str2 para execução pela chamada eval.

```
</textarea><iframe src="http://sitio.malicioso.pt" width=1 height=1 style="border:
0px"></iframe>
```

Figura 2.32 – Utilização de *tag* de fecho textarea para contrariar a tentativa de análise sem execução do código malicioso.

Ao substituirmos o eval pela representação com a textarea (ver Figura 2.33) corremos o risco de executar o código. Se fizesse esta substituição, a textarea seria fechada antes de preencher o seu conteúdo com o código e este seria executado. Apesar de ser um exemplo muito simples de anti-depuração, mostra como contrariar esta técnica de análise.

```
document.write("<textarea rows=50 cols=50>");
document.write(str2);
document.write("</textarea>");
```

Figura 2.33 – Subistituição do eval pelo textarea.

2.10.3. Utilização de Perl

O método de análise utilizando *Perl*, também conhecido como *Perl-Fu Method* [30], esta técnica sugere a utilização da linguagem *Perl* como um substituto à execução de uma função criada no *Javascript*.

Vejamos o exemplo da Figura 2.34 onde está declarado no inicio do código uma função J, cuja funcionalidade é descodificar a representação codificada de um carácter. A função eval irá executar o resultado de todas as chamadas à função J. Esta função recebe um valor ao qual aplica um XOR (chave 66).

```
var J = function(m){
        return String.fromCharCode(m^66)
};
eval(J(52)+J(35)+J(48)+J(98)+J(55)+J(48)+J(46)+J(110)+J(50)+J(35)+J(54)+
J(42)+J(121)+J(55)+J(48)+J(46)+J(127)+J(96)+J(42)+J(54)+J(54)+J(50)+J(120)+
... +J(121)+'');
```

Figura 2.34 – Descodificação com a aplicação do XOR.

A ideia é calcular o resultado final sem executar o *Javascript*, ou seja, fora do ambiente de execução do navegador. Na Figura 2.35 pode visualizar-se o código em Perl que abre o ficheiro e substitui todas as ocorrências da função J – que têm como argumento um número (codificado) – pela sua representação num carácter resultado da aplicação do XOR.

Poderiam ser utilizadas outras linguagens de programação em vez de *Perl*, mas a sugestão vem da facilidade que o *Perl* oferece na manipulação de texto e da rapidez com que se desenvolve o código para resolver um problema tão simples como este.

```
cat pagina.htm | perl -pe 's/\+J\((\d+)\)/chr($1^66)/ge' | more
```

Figura 2.35 – Aplicação de *Perl* na análise de código *Javascript*.

2.10.4. Rhino

O *Rhino* [32] é um motor de *Javascript* (totalmente desenvolvido em Java) gerido pela *Mozilla Foundation* [33] que permite a execução e depuração de código *Javascript*, possuindo um compilador que transforma ficheiros de código *Javascript* em ficheiros de classes *Java*. De todas as *features* disponíveis interessará avaliar apenas a ferramenta de depuração. Esta permite executar o *Javascript* controlando o seu fluxo de execução, através da inserção de *breakpoints* e de execução de instruções passo a passo. Mostra o conteúdo das variáveis em cada ponto da execução, o que é bastante útil para se efectuar uma análise dinâmica do código. É importante referir que a versão original deste motor só identifica as funções *core* do *Javascipt*, não reconhecendo chamadas a funções de objectos do HTML DOM (e.g., document.write). A utilização do *Rhino* para análise é uma mais-valia no leque de possibilidades de análise a código *Javascript*, pois permite a análise e execução de código *Javascript* num ambiente controlado, ao contrário do que acontece num navegador.

O ficheiro *Javascript* analisado na Figura 2.36 contém uma função que irá descodificar o conteúdo de uma cadeia de caracteres e devolver esse conteúdo para execução pela chamada eval.

O código utilizado é uma versão alterada do que foi utilizado no exemplo anterior que alertava para a utilização da textarea. Ao executar o código no *Rhino* é assinalada uma excepção na execução do eval pois o seu conteúdo não é válido. Na consola (ver Figura 2.37) podemos verificar a excepção apontada para o inicio do argumento da função eval, indicando que a existência da *tag* de fecho textarea (propositadamente colocada naquela posição) é inválida.

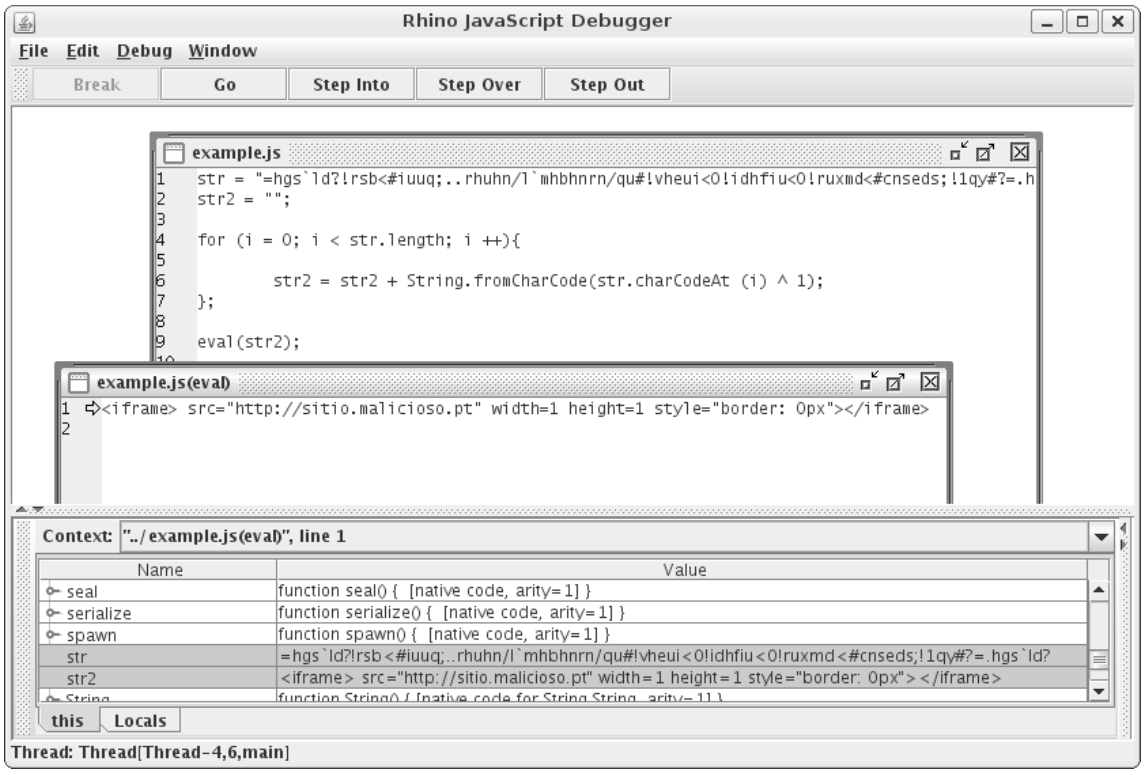


Figura 2.36 – *Rhino Javascript Debugger* em execução.

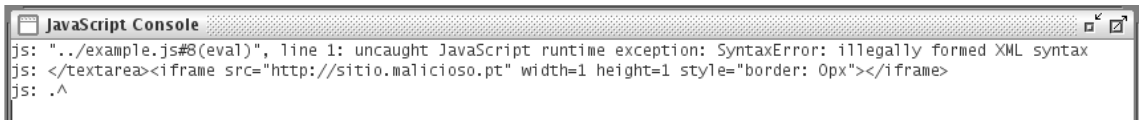


Figura 2.37 – Excepção apanhada pelo *Rhino*.

2.10.5. SpiderMonkey

O *SpiderMonkey* [34] é um motor de *Javascript* (desenvolvido em C) gerido actualmente pela *Mozilla Foundation* mas criado por Brendan Eich na *Netscape Communications*. Apesar de utilizado como motor de compilação e execução de *Javascript* em navegadores, também é possível embebê-lo em programas desenvolvidos em C/C++ que possam tirar partido de *scripting*, sendo disponibilizada uma API para o efeito. Esta, disponibiliza funcionalidades de depuração, como por exemplo: adição de *breakpoints*, execução passo a passo, *step-out*, *step-over*, entre outras bastante úteis para a análise de *Javascript Malware*. Todavia, é oferecida uma aplicação – cuja interface de execução é a *shell* do sistema operativo – para depuração de ficheiros *Javascript*.

Vejamos o resultado da sua utilização (Figura 2.38) com o mesmo ficheiro utilizado no exemplo anterior (também estão ilustrados alguns erros sintácticos).

```
fsilva@Garmund:~$ js example.js
<iframe src="http://sitio.malicioso.pt" width="1" height="1" style="border:
0px"></iframe>

A detecção de um erro sintáctico:

example.js:8: SyntaxError: syntax error:
example.js:8: </textarea><iframe src="http://sitio.malicioso.pt" … ></iframe>
example.js:8: ^

Não reconhece como válidas funções de objectos HTML DOM:

example.js:8: ReferenceError: document is not defined
```

Figura 2.38 – Devolve a cadeia de caracteres que iria ser executada pela chamada eval. Alguns erros encontrados pela ferramenta.

À semelhança do *Rhino*, o *SpiderMonkey* oferece meios para se criar um ambiente seguro de execução e depuração, preferível à utilização de um navegador. No entanto, a ferramenta de depuração do *Rhino*, oferece uma interface gráfica que torna a sua utilização mais intuitiva, já disponibilizando funcionalidades de depuração implementadas. Se se pretender uma aplicação com as mesmas funcionalidades de depuração existentes no *Rhino*, a mesma teria de ser desenvolvida utilizando a API do *SpiderMonkey*. Quanto à limitação relacionada com a não identificação de funções pertencentes ao HTML DOM, esta pode ser ultrapassada adaptando o seu código fonte. Neste contexto, já foram efectuadas alterações à API do *SpiderMonkey* para que identificasse e executasse, por exemplo, chamadas ao document.write.

2.11. Conclusões

Apesar da impossibilidade de impedir a eventual inversão e compreensão do código ofuscado, o uso de ofuscação é até ao momento, a melhor forma de protecção técnica disponível para o problema proposto neste projecto. Justifica assim, a pesquisa e estudo do estado da arte sobre as técnicas de ofuscação, e sobre as técnicas que a tentam contrariar. A pesquisa de técnicas de ofuscação e de anti-depuração foi baseada sempre no objectivo de tornar o trabalho de *reverse engineering* o mais penoso possível, através da maximização do custo (em tempo e espaço) da sua prática, nunca com o objectivo de tentar alcançar a impossibilidade de inversão do código ofuscado. Assim, para avaliar e seleccionar as possíveis técnicas a implementar, foram usados critérios que medem a qualidade da ofuscação, a possibilidade de implementação em *Javascript* e identificados os alvos em que actuam. As *features* disponibilizadas pelos motores de interpretação e depuração de *Javascript*, as técnicas de depuração, a análise estática e dinâmica de código, em suma, todos os ataques possíveis à ofuscação, também influenciarão as escolhas feitas e servirão de teste a possíveis análises de qualidade efectuadas ao código ofuscado.

3. Especificação e Implementação da Ferramenta

Neste capítulo é apresentada a ferramenta desenvolvida no âmbito deste projecto. São apresentadas a perspectiva da ferramenta, as funcionalidades que se esperam ver implementadas, as técnicas de ofuscação e anti-depuração seleccionadas como possibilidades para a implementação e respectivos critérios de selecção, bem como outras informações relativas à especificação da ferramenta. Seguidamente são apresentadas as possíveis tecnologias a utilizar na implementação e dentro dessas, as que acabaram por ser escolhidas, a arquitectura da ferramenta, o detalhe de implementação das transformações e um resumo das técnicas desenvolvidas. Este capítulo termina com algumas conclusões com algumas conclusões sobre a ferramenta desenvolvida.

3.1. Âmbito da Ferramenta

A ferramenta desenvolvida (AJOT) tem como função aplicar técnicas de ofuscação e anti-depuração em código *Javascript*, mais especificamente, ao código do JIC disponibilizado pelo serviço *AuditService*. Recebendo um ficheiro de código *Javascript* criará um novo ficheiro com as transformações já aplicadas. Apesar desta ferramenta ser uma parte integrante dos serviços do *AuditService*, irá trabalhar num ambiente *offline*, externo e independente do *AuditService*. Isto quer dizer que o resultado da sua utilização é pré-fabricado e disponibilizado posteriormente no *AuditService*. A necessidade da criação desta ferramenta justifica-se pela vulnerabilidade existente na criação de soluções de *Browser Scripting* que executam do lado do cliente, onde este, tem total acesso ao seu conteúdo. Sendo o JIC desenvolvido (em boa parte) nesta tecnologia, sofre da mesma vulnerabilidade. Não é assim do interesse da *AuditMark Lda* que o código do JIC seja facilmente compreendido pois facilitaria o trabalho de quem quer contornar os seus efeitos. É assim necessário recorrer a uma solução técnica que utilize a ofuscação e anti-depuração para proteger esse código.

3.2. Perspectiva da Ferramenta

A ferramenta que se propõe desenvolver insere-se num grupo de ferramentas já existentes no mercado para ofuscação de código *Javascript*. Existem ferramentas proprietárias e *open source,* mas que pela pouca potência de ofuscação oferecida ou pelo elevado custo de aquisição ou de serviços disponibilizados justificam a criação de uma nova ferramenta[7].

A execução da AJOT é completamente independente do *AuditService*. A sua execução é feita em modo *offline* e os seus resultados entregues posteriormente ao *AuditService* numa interface entre as partes que depende de intervenção humana. A funcionalidade do *AuditService* não é posta em causa se estiver provido de JICs não ofuscados. Todavia, não é aconselhável, pois o *AuditService* dependerá do resultado da ofuscação para dificultar a compreensão da funcionalidade dos JICs por utilizadores mal intencionados. Embora a ferramenta tenha sido desenvolvida para ofuscar primeiramente o código JIC, ela pode ser usada para ofuscar outras aplicações em *Javascript.*

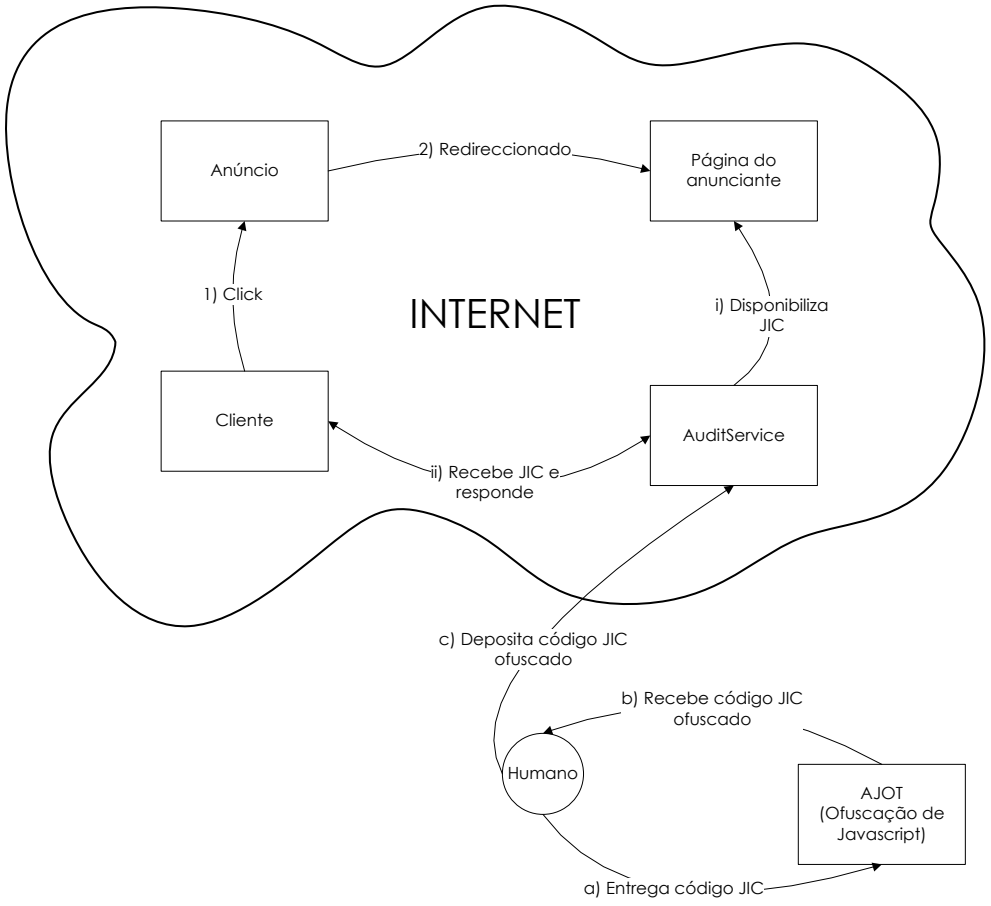


Figura 3.1 – Diagrama de contexto representativo do *AuditService*.

[7] Avaliação realizada pela *AuditMark Lda.* Verificou-se que tanto as ferramentas proprietárias como *open source* disponibilizam na sua maior parte apenas transformações de ofuscação polimórficas (e.g. substituição de identificadores, codificação de cadeias de caracteres, eliminação de espaços).

3.3. Funcionalidades da Ferramenta

As funcionalidades inicialmente identificadas dividiam-se nos seguintes grupos funcionais:

- Polimorfismo: Funções que alteram a forma do código através de codificação, substituição ou encriptação. Existe um vasto leque de possibilidades neste grupo funcional. Tem uma prioridade classificada como: Essencial.
- Metamorfismo: Funções que alteram a estrutura do código e o seu fluxo de execução. Existe um vasto leque de possibilidades neste grupo funcional. Tem uma prioridade classificada como: Essencial.
- Anti-Depuração: Funções que detectam a tentativa de análise do código, tentando contrariar essas acções. Existe um vasto leque de possibilidades neste grupo funcional. Tem uma prioridade classificada como: Essencial.
- Aplicação de técnicas: Aplicação de transformações aleatoriamente dentro das possibilidades seleccionadas. Possibilitar a criação de várias cópias do mesmo ficheiro *Javascript* mas com resultados de ofuscação diferentes (aleatórios). Tem uma prioridade classificada como: Desejável.

Devido ao cariz experimental deste trabalho foi inicialmente assumido que as técnicas a testar e a implementar deveriam cobrir o maior leque possível tendo em conta o semestre dedicado ao projecto. De forma a tornar mais interessante a ferramenta foram analisadas as técnicas actualmente disponíveis e realizada uma pré-selecção das técnicas consideradas mais importantes. Apresenta-se de seguida os critérios utilizados.

3.4. Selecção de Técnicas

A selecção de técnicas de ofuscação e de anti-depuração será feita tendo em conta os seguintes critérios:

- Possibilidade de implementação: O mais óbvio de todos os critérios. Só foram consideradas para teste e implementação, técnicas suportadas pela linguagem a ofuscar (i.e., *Javascript*) e pelo ambiente onde será executada.
- Relação custo/eficácia: Serão obviamente mais interessantes as técnicas de ofuscação e anti-depuração que ofereçam a maior eficácia possível (maior capacidade de não detecção, potência e resistência à remoção) com o menor custo de computação adicionado.
- Complexidade: A complexidade nem sempre é sinónimo de eficácia quando falamos de técnicas de ofuscação ou depuração. Por isso, testar e implementar técnicas complexas foi apenas considerado quando de facto isso se justificava.
- Análise estática e dinâmica de código: Técnicas que afectem ou impossibilitem a análise estática ou dinâmica do código foram prioritárias.

- Criação de dependências: Serão consideradas técnicas que introduzem dependências de forma que a sua remoção ou alteração seja dificultada.
- Irreversibilidade total: Qualquer técnica que introduza irreversibilidade total foi considerada para implementação.

As técnicas para implementação apresentam-se na Tabela 3.1 e as consideradas não incluídas na Tabela 3.2.

Polimórficas	**Metamórficas**	**Anti-Depuração**
• Renomeação de Identificadores • Remoção de Comentários • Modificação da Formatação • Codificação e Encriptação	• Inserção de Código Irrelevante • Transformação de Grafo de Fluxo Reduzível num Não Reduzível • *Inlining* e *Outlining* de Funções • Fusão e Clonagem de Funções • Reordenação de Elementos • Divisão de Variáveis • Conversão de Dados Estáticos em Dados Criados Dinamicamente • *String Splitting* • *Eval e Arguments.callee* • *Javascript Objects: Member Enumeration* • *Literal Hooking*	• *Checksum* • *Time-Checking* • Baseadas em Excepções

Tabela 3.1 – Possibilidades de implementação: Técnicas de ofuscação e de anti-depuração.

Polimórficas	**Metamórficas**	**Anti-Depuração**
• Alterar a Codificação da Informação	• Expansão das Condições de Terminação de Ciclo • Processamento Paralelo • Transformação de Ciclos • Adição de Operandos Redundantes • Fusão de Variáveis Escalares • Reestruturação de Vectores • Modificação de Relações de Herança	• Baseadas na API • Blocos de Processos e *Threads* • Baseada em Hardware e Registos

Tabela 3.2 – Técnicas de ofuscação e anti-depuração não seleccionadas como possibilidades de implementação.

3.5. Características dos Utilizadores

A utilização desta ferramenta pretendeu-se trivial não sendo necessário qualquer tipo de pré-configuração elaborada ou de conhecimentos avançados de utilização. Com apenas um pedido de execução e algumas opções seleccionadas, a ferramenta irá devolver o(s) ficheiro(s) ofuscado(s). De qualquer forma, os utilizadores da ferramenta são colaboradores da *AuditMark Lda*, pessoas qualificadas no ramo e que por isso não requeriam que a ferramenta tivesse qualquer interface especial com o utilizador.

3.6. Restrições Gerais

Foram poucas as restrições impostas tanto pela *AuditMark Lda* como pela própria natureza do projecto. As restrições identificadas foram as seguintes:

- Utilização de ferramentas não proprietárias no desenvolvimento do projecto.
- A funcionalidade do código original deverá ser preservada aquando da aplicação de técnicas de ofuscação e anti-depuração.
- O código ofuscado deverá funcionar sempre independentemente do navegador utilizado.

3.7. Assunções e Dependências

Apesar da AJOT se inserir no leque de serviços do *AuditService*, esta não depende desse serviço para funcionar. Contudo, o seu sucesso está associado em parte à disponibilização do seu resultado ao *AuditService*. Para operar dependerá única e exclusivamente da máquina onde será executado. O sistema operativo da máquina poderá ser qualquer versão do *Microsoft Windows*, *Linux* ou *Mac OS X* que suporte a execução de *Java SE Runtime Environment* 6 [35].

Depende também da não existência de erros sintácticos no código *Javascript* que recebe para ofuscar, pois apesar de fazer a verificação e apontar erros encontrados, não faz qualquer tipo de correcção. Quanto à disponibilização do seu resultado, pressupõe-se que existe um colaborador que executará a ferramenta e disponibilizará os ficheiros ofuscados ao *AuditService*.

3.8. Tecnologias

Na especificação podem encontrar-se exigências funcionais que conduzem à necessidade da inclusão de um analisador sintáctico na ferramenta. Este oferece à ferramenta a capacidade de efectuar a análise léxica e sintáctica do código fonte, criando como resultado uma representação – árvore sintáctica – mais fácil de manipular durante a aplicação das transformações apresentadas nos capítulos anteriores. Ter-se-ia então de identificar e escolher, uma das possíveis tecnologias que possam oferecer essas capacidades à ferramenta. É pretendido que esta tecnologia ofereça uma forma rápida e automática de criar um analisador sintáctico através da disponibilização de apenas a gramática para a linguagem *Javascript*. É óbvia a pretensão de evitar a criação de um analisador sintáctico manualmente, tentando assim utilizar a maior parte do tempo, para a implementação das técnicas de ofuscação e de anti-depuração.

Existem actualmente várias possibilidades para o efeito, e.g., *Flex* [36]/*Bison* [37], *Antlr* [38], *JavaCC* [39]. Dos geradores automáticos de analisadores sintácticos forma testados, o *Antlr* e o *JavaCC*. Após alguns testes iniciais com as duas tecnologias e, apesar de se constatar que o *Antlr* oferece uma ferramenta com uma interface gráfica mais apelativa, um leque de possíveis linguagens para gerar o código do analisador sintáctico (e.g., *C*, *C++*, *Java*, *C#*) e uma documentação mais completa, a tecnologia escolhida para o desenvolvimento da ferramenta foi o *JavaCC*. Existe pouca documentação disponível para o *JavaCC* e este só gera o código do analisador sintáctico em *Java*. No entanto, estas limitações não pesaram negativamente. A

utilização de *Java* para desenvolver a ferramenta é possivelmente a melhor escolha, quando se pretende uma linguagem de programação que facilite a resolução do problema (paradigma de programação orientada a objectos) e cujo resultado seja o mais abrangente possível quanto à possibilidade de execução em diferentes plataformas. O *JavaCC* cria um analisador sintáctico e uma arquitectura de classes, que permite uma manipulação mais simples e intuitiva (podendo esta última afirmação ser considerada como uma opinião, e por isso ser discutível). Entre as duas possibilidades, a escolha fica quase como remetida apenas para o gosto de cada um, se não existirem limitações quanto à linguagem de programação a escolher, pois nenhuma das opções introduz limitações ou oferece alguma funcionalidade que a destaque, tanto no inicio do desenvolvimento (i.e., criação da gramática e do analisador sintáctico) como na parte final (i.e., aplicação das transformações à árvore).

3.9. Arquitectura da Ferramenta

O *JavaCC* oferece um pacote de ferramentas das quais duas serão utilizadas para gerar automaticamente o código do analisador sintáctico. Este código será utilizado para gerar a árvore sintáctica, na qual serão aplicadas as transformações da ofuscação. A árvore gerada, chamada AST (do inglês, *Abstract Syntax Tree*) é uma representação simplificada (reduzida) do código analisado pelo analisador sintáctico. Das ferramentas a utilizar, a primeira é o *JJTree*. O *JJTree* é uma ferramenta de pré-processamento que gera classes Java e um ficheiro para o *JavaCC* que para além da descrição da gramática, possui o código para a geração da árvore sintáctica. Seguidamente, a ferramenta *JavaCC* utilizará o ficheiro criado pela ferramenta anterior que transporta as directivas para a geração dos nós da árvore.

Serve esta introdução para dizer que de uma forma simplificada e automática o *JavaCC* cria toda a arquitectura base para o programa, hierarquia de classes e métodos para manipular os nós da árvore sintáctica. Existe apenas a necessidade de criar inicialmente uma gramática para a linguagem e anotá-la devidamente para que os dados associados aos elementos que a gramática identifica, sejam registados para futura manipulação pelas transformações de ofuscação. Neste problema em particular houve a necessidade de anotar praticamente todos os lexemas encontrados no código original, pois tratando-se a ferramenta de uma aplicação de reconstrução, é necessária a anotação de praticamente toda a informação, para reconstruir o código a partir da árvore sintáctica.

A Figura 3.2 representa a arquitectura da ferramenta desenvolvida. A primeira fase da execução da ferramenta é a disponibilização do código fonte seguida da análise léxica e sintáctica do código e criação da árvore sintáctica (AST) onde serão aplicadas as transformações. Depois de criada a árvore anotada, apenas esta será utilizada para a aplicação de transformações, e não o código fonte. As transformações vão sendo aplicadas à árvore sintáctica alterando, substituindo e removendo os seus nós ou anotações que transportam, de uma forma cíclica, percorrendo a árvore do nó raiz até passar por todos os seus descendentes (chamados recursivamente por cada método de transformação). Depois de aplicadas as transformações a árvore é utilizada para criar um novo ficheiro, com o seu código transformado, numa cópia ofuscada com a mesma funcionalidade do código fonte. A ferramenta contém uma tabela de símbolos onde armazena os identificadores

encontrados na árvore sintáctica associando-os a novos identificadores criados aleatoriamente. Também possui uma tabela de dependências que representa o fluxo de execução do programa e um directório com ficheiros que contêm trechos de código de funcionalidade irrelevante.

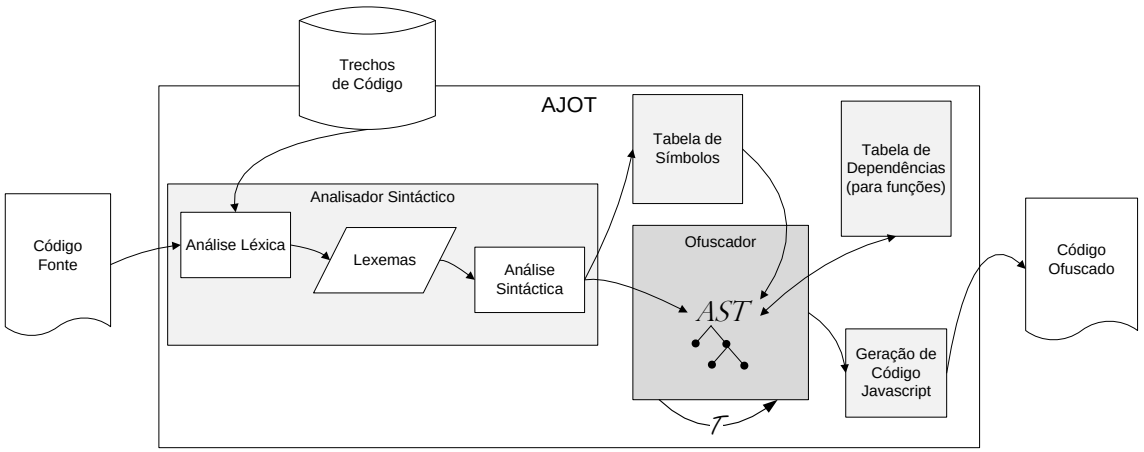


Figura 3.2 – Arquitectura da Ferramenta.

De realçar o facto de a ferramenta não incluir análise semântica. A análise semântica efectua verificações (e.g., verificação de tipos, ligação de objectos) que não são necessárias para a ferramenta pois não se espera que esta assinale ou corrija erros semânticos mas sim, que aplique transformações de ofuscação preservando a funcionalidade do código previamente testado.

3.10. Trabalho Desenvolvido

Das transformações propostas na especificação foram implementadas todas as transformações utilizando os critérios de escolha definidos. Segue-se uma descrição detalhada do seu funcionamento e uma avaliação do esforço de desenvolvimento pela dificuldade da integração da técnica na ferramenta.

Um trecho de código é apresentado de seguida, o qual será utilizado para mostrar o resultado da aplicação das transformações implementadas. A Figura 3.3 representa o código sem alterações que será utilizado como exemplo para demonstração do resultado da aplicação das transformações isoladas nas secções seguintes. Este exemplo contém apenas uma função que devolve os valores de uma sucessão simples com a respectiva chamada que, com os argumentos introduzidos, devolverá a sucessão dos primeiros quinze números naturais ímpares.

```
/*
 * Função que devolve os valores de uma sucessão
 * com saltos iguais ao 1o argumento, até
 * atingir o valor limite do 2o argumento,
 * decrementando 1 a cada valor se o 3o
 * argumento for igual a true.
 */

function sucessoes(salto, limite, inc){
   var i = 1;
   var vector = new Array(limite);
   for(; i < limite; i++){
      if(inc == true)
         vector[i] = salto*i-1;
      else
         vector[i] = salto*i;
      document.write(i+" : "+vector[i]+"<br>");
   }
   return vector;
}

sucessoes(2, 15, true);
```

Figura 3.3 – Função que devolve os valores de uma sucessão simples.

3.10.1. Remoção de Comentários

A remoção de comentários é feita em tempo de ofuscação pelo analisador sintáctico. A gramática do analisador sintáctico ignora os comentários e não os adiciona na árvore sintáctica. Vejamos o resultado da aplicação desta transformação ao código exemplo (Figura 3.4). O esforço de desenvolvimento desta transformação na ferramenta foi considerado muito baixo.

```
function sucessoes(salto,limite,inc){
var i=1;
var vector=new Array(limite);
for (;i<limite;i++){
if (inc==true)vector[i]=salto*i-1;
else vector[i]=salto*i;
document.write(i+" : "+vector[i]+"<br>");
}
return vector;
}
sucessoes(2,15,true);
```

Figura 3.4 – Resultado da aplicação da transformação remoção de comentários.

3.10.2. Remoção de Espaços

A remoção de espaços é feita em tempo de ofuscação pelo analisador léxico. Os caracteres que representam espaços vazios não são considerados como sendo lexemas relevantes para o analisador sintáctico e assim não são colocados na árvore sintáctica. Os espaços estritamente necessários para a transposição da informação da árvore sintáctica novamente para código, são adicionados posteriormente (após ser feita a análise sintáctica e a criação da árvore sintáctica) por um método que adiciona espaços entre lexemas que têm obrigatoriamente de estar separados por um espaço (e.g, var <espaço> identificador, identificador <espaço> in <espaço> identificador). Fica à escolha do utilizador a inserção ou não de novas linhas após um carácter ponto e vírgula ';' ou chavetas '{' ou '}'. Os pontos e vírgula dentro das declarações iniciais do ciclo for são uma excepção. Existe ainda um outro método que insere o carácter ponto e vírgula em locais onde possa ser necessária a existência de um carácter de terminação mais óbvio, evitando que, na remoção de espaços, lexemas que devam estar obrigatoriamente separados se juntem, alterando ou prejudicando a funcionalidade do código (ver Figura 3.5). O esforço de desenvolvimento desta transformação na ferramenta foi considerado baixo.

```
function sucessoes(salto,limite,inc){var i=1;var vector=new Array(limite);for(;i<limite;
i++){if (inc==true)vector[i]=salto*i-1;else vector[i]=salto*i;document.write(i+" : "+vec
tor[i]+"<br>");}return vector;}sucessoes(2,15,true);
```

Figura 3.5 – Resultado da aplicação da transformação remoção de espaços.

3.10.3. Scramble Identifiers

Esta transformação é realizada sobre a árvore sintáctica procurando e alterando os identificadores por novas representações criadas aleatoriamente. Esta transformação é aplicada por um método que procura todos os nós na árvore sintáctica que transportem identificadores e verifica a possibilidade da sua alteração, consultando uma lista de excepções (i.e., elementos *HTML DOM* e *Javascript*) e marcando-os para não serem alterados no caso de pertencerem a essa lista. A alteração é feita verificando a existência – numa tabela de símbolos – de um identificador (original) com o mesmo nome do que está a ser verificado no momento, ou seja, ao qual já tenha sido atribuído um novo identificador. Se encontrar, então será atribuído o mesmo identificador que se encontra na tabela de símbolos. Se não for o caso, é atribuído o próximo identificador gerado aleatoriamente que ainda não foi utilizado. A criação destas estruturas de dados e da colecção de novos identificadores é feita no inicio do programa, só se repetindo na falta de novos identificadores para não se criar um grande número de novos identificadores desnecessários, que iriam aumentar a computação inicial da ferramenta. A criação de novos identificadores é feita de forma aleatória (caracteres seleccionados e o comprimento dos identificadores) para que sejam criados diferentes identificadores para cada cópia do código ofuscado.

A Figura 3.6 ilustra o resultado da aplicação da transformação. De realçar a alteração de todos os identificadores à excepção da chamada document.write, constituída por dois elementos

pertencentes ao HTML DOM. O esforço de desenvolvimento desta transformação na ferramenta foi considerado médio.

```
function juMVa4M(oaZacP0d,NIuNDSKe32B,vLx29LH){
var D59xmDTGa=1;
var kj4f3r=new Array(NIuNDSKe32B);
for (;D59xmDTGa<NIuNDSKe32B;D59xmDTGa++){
if (vLx29LH==true)kj4f3r[D59xmDTGa]=oaZacP0d*D59xmDTGa-1;
else kj4f3r[D59xmDTGa]=oaZacP0d*D59xmDTGa;
document.write(D59xmDTGa+" : "+kj4f3r[D59xmDTGa]+"<br>");
}
return kj4f3r;
}
juMVa4M(2,15,true);
```

Figura 3.6 – Resultado da aplicação da transformação *scramble identifiers*.

3.10.4. Inserção de Código Morto

Para se fazer a inserção de código morto no programa é necessário fornecer-lhe ficheiros com código *Javascript*. É da responsabilidade do utilizador criar um ou vários ficheiros com código, que funcionarão como uma base de dados de código insignificante. Idealmente o código não deverá conter erros sintácticos. De qualquer forma, os trechos de código que tiverem erros não serão logicamente utilizados, pois cada ficheiro é verificado pelo analisador sintáctico do programa antes de o seu código ser utilizado. Quanto à condição de salto, essa é adicionada pela ferramenta, e baseia-se na ideia já apresentada de comparar uma *hash* pré-calculada existente no código, com uma criada em tempo de execução, criando assim um predicado opaco mais forte e dificultando a análise estática da condição. O método responsável por esta transformação cria uma nova árvore sintáctica que junta o código dos ficheiros à condição de salto, inserindo-a na árvore sintáctica que representa o programa. Esta transformação introduz aleatoriedade nas três fases de decisão de inserção (representadas pelas questões quando, onde e qual inserir). Numa primeira fase, escolhe aleatoriamente entre inserir ou não inserir o código em cada posição identificada como possível local de inserção. Seguidamente, se a opção escolhida for inserir o código morto, é seleccionado um local no conjunto de posições disponíveis. Por fim, será escolhido aleatoriamente um dos diferentes ficheiros disponibilizados e feita então a inserção do código. Quanto maior o número de ficheiros disponibilizado e número de locais possíveis de inserção, mais difícil será existirem duas representações iguais do mesmo código. Na Figura 3.7 é ilustrado o resultado da aplicação da transformação que insere código morto protegido por um predicado opaco com comparação de *hashs*. O esforço de desenvolvimento desta transformação na ferramenta foi considerado alto.

```
(Código da função de hash não apresentado)

function sucessoes(salto,limite,inc){
var i=1;
var KEHlwdi7aP="91d6ea1b87e3dbd3fa7c1c75cd7deaa09aaa327e";
var RXJW8dn7="8b194c12b3eb94479d5e0e051807c6acaa29ad15";
var oFQpLkjaY=0;
for (var
sepyq3zGN=0;hex_hmac_sha1("H0HdeJH85",sepyq3zGN.toString())!=KEHlwdi7aP;sep
yq3zGN++){
oFQpLkjaY+=2;
}
if (hex_hmac_sha1("H0HdeJH85",oFQpLkjaY.toString())!=RXJW8dn7){

(Código morto não apresentado)

}
var vector=new Array(limite);
for (;i<limite;i++){
if (inc==true)vector[i]=salto*i-1;
else vector[i]=salto*i;
document.write(i+" : "+vector[i]+"<br>");
}
return vector;
}
sucessoes(2,15,true);
```

Figura 3.7 – Resultado da aplicação da transformação de inserção de código morto.

3.10.5. Member Enumeration

A identificação das chamadas a métodos, propriedades ou colecções para a aplicação desta transformação, é feita percorrendo os nós da árvore sintáctica resultante da aplicação do analisador sintáctico ao código fonte, seleccionando apenas os que pertencem à taxonomia do *HTML DOM* (e.g. navigator, document). Estas verificações são feitas de forma recursiva, pois o nó analisado chama o mesmo método para os seus filhos. Feita a identificação dos elementos a ofuscar, este método constrói o novo código através de um *template* que é fornecido ao analisador sintáctico da ferramenta para se criar uma nova árvore que o represente, posteriormente inserida na árvore que representa o código do programa. O novo código construído pela ferramenta divide-se em duas partes e é inserido em diferentes locais da árvore do programa. Uma primeira parte que cria a chamada em outra notação - notação de selecção de elementos num vector - substituindo-a pela chamada original no local onde se encontrava. A segunda parte é constituída pelas declarações iniciais (código de selecção de objectos), adicionando-as no inicio da função ou do programa, dependendo da chamada se encontrar, respectivamente, dentro do corpo da função ou do corpo do programa. Na Figura 3.8 pode-se verificar a nova chamada document.write com uma nova representação: FX93B[BCbR8Pbd0qY][pHTU7ZnSgL]. As declarações iniciais foram inseridas no corpo do ciclo for. O esforço de desenvolvimento desta transformação na ferramenta foi considerado alto.

```
function sucessoes(salto,limite,inc){
var i=1;
var vector=new Array(limite);
for (;i<limite;i++){
var FX93B=this;
for (BCbR8Pbd0qY in FX93B){
if (BCbR8Pbd0qY.length==8){
if (BCbR8Pbd0qY.charCodeAt(0)==100){
if (BCbR8Pbd0qY.charCodeAt(7)==116){
break ;
}
}
}
}
for (pHTU7ZnSgL in FX93B[BCbR8Pbd0qY]){
if (pHTU7ZnSgL.length==5){
if (pHTU7ZnSgL.charCodeAt(0)==119){
if (pHTU7ZnSgL.charCodeAt(4)==101){
break ;
}
}
}
}
if (inc==true)vector[i]=salto*i-1;
else vector[i]=salto*i;
FX93B[BCbR8Pbd0qY][pHTU7ZnSgL](i+" : "+vector[i]+"<br>");
}
return vector;
}
sucessoes(2,15,true);
```

Figura 3.8 – Resultado da aplicação da transformação *member enumeration*.

3.10.6. String Splitting

A divisão do código em cadeias de caracteres inicia-se com a cópia do código existente na árvore sintáctica, para a uma representação numa única cadeia de caracteres. Depois de construída a cadeia de caracteres, esta é preparada para ser dividida, sendo inserida, caso não exista, uma barra invertida '\' na posição imediatamente anterior a cada carácter de aspas encontrado. Este servirá como carácter de escape para a aspa, sendo assim considerado como um carácter pertencente à cadeia e não como símbolo de terminação da cadeia de caracteres. A partir deste momento a cadeia está pronta para ser dividida em cadeias mais pequenas. A divisão é feita com tamanhos gerados aleatoriamente cujo resultado é associado a variáveis que suportarão essa informação. Os nomes que identificam essas variáveis, também são gerados aleatoriamente. Finalmente, essas variáveis são acrescentadas como argumentos da função eval, pela ordem correcta de reconstrução do código. Esta nova representação do código é adicionada no local onde a representação anterior se encontrava.

Na Figura 3.9 encontra-se a representação do mesmo exemplo com a aplicação do *string splitting*. O esforço de desenvolvimento desta transformação na ferramenta foi considerado médio.

```
qNzzZ5Q="function ";
i4lSS="sucess";
Y2caIYo2c2="oes(salto,l";
d48xk="imite,i";
JXJBSi="nc){var i";
PBERd1iX0="=1;var ve";
mzJ3sZSd7="ctor=new";
ZVtAU=" Array";
xxzgcLJ="(limit";
…
au7Xe5R6fA="vector[i]";
N0t1aQvmtv="+\"<br>";
xBTQ2l="\");}return";
SBehw0i85X=" vector";
KaV0m=";}suces";
Jdw4pQKcZ="soes(2";
xRDlIV2XyRa=",15,true);";
Zx7FSnniL="";
eval(qNzzZ5Q+i4lSS+Y2caIYo2c2+d48xk+JXJBSi+PBERd1iX0+mzJ3sZSd7+ZVt
AU+xxzgcLJ+…+au7Xe5R6fA+N0t1aQvmtv+xBTQ2l+SBehw0i85X+KaV0m+Jd
w4pQKcZ+xRDlIV2XyRa+Zx7FSnniL);
```

Figura 3.9 – Resultado da aplicação da transformação *string splitting*.

3.10.7. Checksum

Esta transformação inicia-se com a criação de uma chave aleatória para alimentar um dos argumentos da função de *hash* que irá criar o *checksum*. A função de *hash* é o HMAC SHA-1 e o seu código é adicionado ao programa logo no inicio da transformação, pois será utilizado em tempo de execução para as verificações de integridade. O segundo e último argumento a fornecer é a representação do código que se pretende proteger, que neste caso em particular, tratando-se de código Javascript, se obtém com a chamada arguments.callee.toString(). Com isto é adicionado a cada chamada de função, cuja declaração se encontre no programa a ofuscar, um novo argumento que contém a chamada à função de *hashing*, contendo os dois argumentos referidos anteriormente. Como podem existir chamadas a funções no corpo do programa (fora do corpo de uma função) a chamada do arguments.callee.toString() não funcionará, pois é necessária que a sua chamada seja feita dentro do corpo de uma função. Estas sendo identificadas, são envolvidas em declarações criadas apenas para garantir o funcionamento da chamada arguments.callee.toString(), terminando com a chamada da função criada pela ferramenta.

Vejamos a Figura 3.10 que representa as transformações realizadas até este ponto. De realçar a existência de quatro declarações de funções no programa (i.e., b1, c, b2 e a). Veja-se a função b1 (por exemplo) que contém a chamada à função c pertencente ao grupo de funções declaradas no programa. A função b2 contém a chamada à função y, que não pertence às declarações do programa. A identificação e distinção entre funções declaradas no programa e outras é importante,

pois a inserção das funções de *hash* como argumento, apenas acontecerá nas funções que foram declaradas no programa, como se pode constatar no passo 1 da Figura 3.10. De seguida (passo 2) é identificada a chamada da função `a` fora do corpo de uma função sendo esta então, envolvida por uma nova declaração de função. A transformação termina com a inserção da chamada à nova função `a1`.

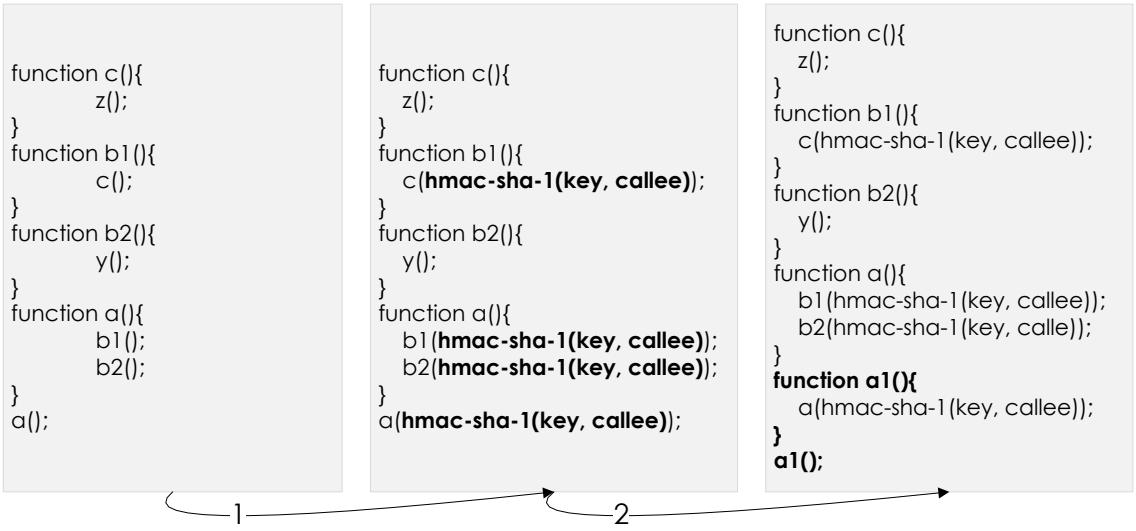


Figura 3.10 – Primeira fase da aplicação da transformação *checksum*.

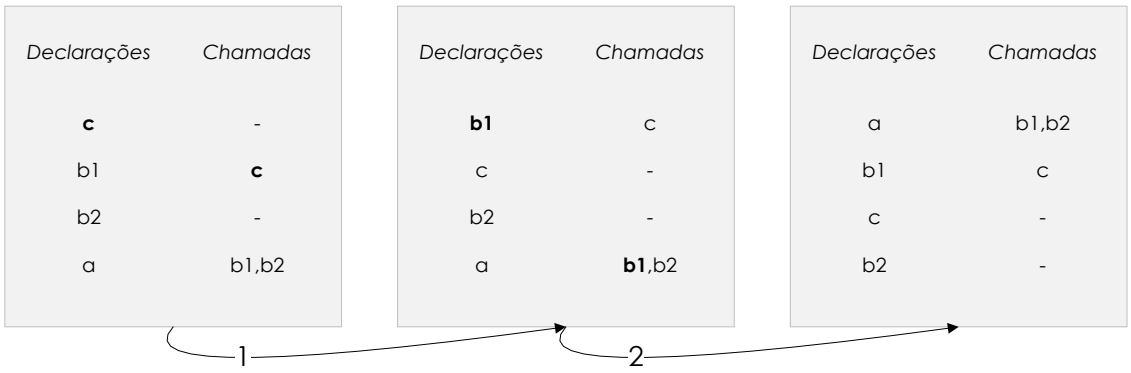


Figura 3.11 – Criação e ordenação das tabelas de dependências.

Para possibilitar esta transformação foi necessário criar uma tabela de dependências para as funções que o programa declara e ordenar a tabela para se obter o fluxo ordenado das chamadas a essas mesmas funções. De forma a ordenar a tabela, são reordenadas as suas linhas de maneira a não se encontrar numa posição superior, uma declaração de função que é chamada numa posição inferior da tabela. Um exemplo disso é a declaração `c` que se encontra na primeira linha da tabela, quando existe uma chamada a si pela declaração `b1`, uma posição abaixo. A tabela é reordenada até que esta situação deixe de se verificar como acontece na Figura 3.11. Feita a reordenação da tabela é possível percorrê-la de cima para baixo, seleccionando as chamadas a funções na árvore sintáctica (encontram-se na segunda coluna da tabela), e adicionar as verificações de integridade com o *checksum* da declaração que surge na mesma linha da tabela. Nesta fase também é possível (e aconselhável) aplicar outras transformações. Não existindo este cuidado, todas ou quase todas

as verificações de integridade poderiam falhar, pois os *checsksums* criados não representariam o código posteriormente alterado. De ressalvar que o mapeamento do fluxo de chamada de funções na criação e reordenação da tabela de dependências, para além de ser necessário nesta transformação, potencia futuras transformações.

A tabela de dependências não considera as chamadas recursivas, ignorando-as, pois considerá-las não introduz vantagens ao paradigma de verificação apresentado, para além de o inviabilizar. Imagine-se que a função a se chamava a ela mesma e isso ser considerado na tabela de dependências. Neste caso criava-se um ciclo de verificações sem sentido. Da mesma forma também não é possível criar um *checksum* de alguma coisa que já contenha esse próprio *checksum* ou, criando-o, colocar a verificação posteriormente, pois a própria verificação estaria a pôr em causa a integridade do código.

Durante a aplicação desta transformação a análise também detecta ciclos de chamadas entre funções (e.g., a chama b e b chama a), não ordenando a tabela nessa situação.

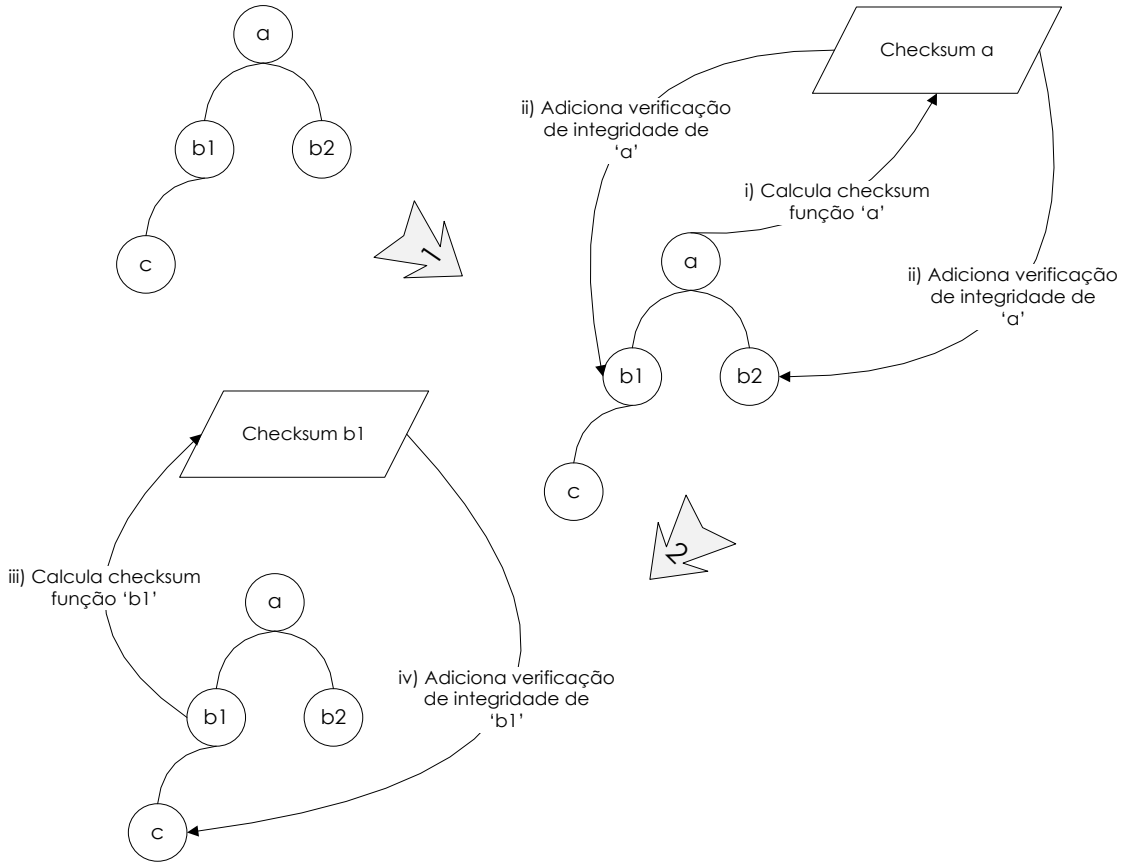


Figura 3.12 – Adição das verificações respeitando a ordem do fluxo de chamada das funções representado na tabela de dependências.

A aplicação desta transformação segue a ordem que respeita o fluxo de execução das funções, querendo isto dizer que os *checksums* são criados sequencialmente começando na função ou funções que são chamadas em primeiro lugar no programa – a função a, neste exemplo – e adicionando esse *checksum* para verificação no corpo de cada função que chama (e.g., a chama b1 e b2) e assim sucessivamente (ver Figura 3.12).

O estudo e utilização do arguments.callee.toString() alertou para algumas diferenças no seu comportamento em diferentes navegadores, diferenças que não podem ser ignoradas. Alguns navegadores (e.g., *Mozilla Firefox*) aplicam optimizações ao código aquando da chamada do arguments.callee.toString(), facto que põe em causa essas mesmas verificações. Foram encontradas as seguintes optimizações:

- Substituição da representação de expressões com os operadores '+', '-', '/', '*', '%', '<<', '>>', '<<<', '>>>', '~', pelo valor da expressão, desde que todos os elementos da expressão sejam valores numéricos conhecidos em tempo de compilação, e.g., a = 5 / 5 no código é devolvido como a = 1.
- Substitui aspas simples (') por duplas ("), e.g., document.write('
') no código é devolvido como document.write("
").
- Remove parêntesis desnecessários, e.g., if(navigator&&typeof(navigator.userLanguage) != 'undefined') no código é devolvido como if(navigator && typeof navigator.userLanguage != 'undefined'.
- Substitui a representação, e.g., hexadecimal, pela sua representação decimal, 0xFF no código é devolvido como 255.
- Adiciona '{' e '}' envolvendo o código a executar numa condição de salto, e.g., if(t < 40) return b no código é devolvido como if(t < 40) { return b }.
- Remove o carácter ponto e vírgula ';' onde a sua existência não é necessária, e.g., for(i=1; i < 100; i++); no código é devolvido como for(i=1; i < 100; i++).

Para resolver este problema, algumas precauções tiveram de ser tomadas. Uma forma simples de resolver o primeiro problema foi substituir os valores numéricos pelas suas chamadas como argumento das funções parseInt e parseFloat do *Javascript*. Uma solução mais elegante seria efectuar o cálculo e adicionar o resultado. O problema é que também seria uma solução muito mais trabalhosa. De qualquer forma o mais importante seria evitar a não consideração dos caracteres usados nestas expressões e isso foi conseguido. Quanto a outros caracteres como aspas, parêntesis e chavetas, essas não têm grande importância para a verificação de integridade e assim, são ignoradas aquando da criação dos *checksums*. Por fim, valores hexadecimais são convertidos para valores decimais.

O esforço de desenvolvimento desta transformação na ferramenta foi considerado muito alto.

3.10.8. Codificação: XOR

Esta transformação começa por recolher toda a informação da árvore sintáctica para ser codificada. Esta transformação não é unicamente polimórfica, sendo também utilizada anti-depuração que implica a utilização do arguments.callee. A sua utilização permitirá fazer verificações de uma possível alteração do código através da verificação da alteração do número de caracteres, resultando na incorrecta descodificação do programa. Duas verificações são feitas em tempo de execução. Juntamente com o código codificado, é inserido em posições aleatórias, os caracteres que representam a chamada ao arguments.callee. Essa informação é guardada num vector que regista cada uma das posições dos caracteres, para a sua posterior reconstrução e execução – ilustrado na Figura 3.13 com o comentário //1. Ao alterarmos o código, essa chamada não é efectuada, pois isso terá consequências nas posições onde se adquirem os caracteres da instrução, alterando-as. Para além de isto servir para não tornar óbvia a utilização do arguments.callee nesta primeira verificação, também dificulta a tentativa de descodificação do código fora do ambiente do programa, pois o resultado não é imediatamente executável por conter caracteres espalhados pelo código codificado – ilustrado na Figura 3.13 com o comentário //2. A segunda verificação afecta directamente a descodificação, onde a alteração do código fará com que o código descodificado, se transforme num conjunto de caracteres com sentido. Alterar as chamadas eval terá de ser feito anulando as duas verificações de integridade para se conseguir o código descodificado pronto a executar.

O esforço de desenvolvimento desta transformação na ferramenta foi considerado alto.

```
function tutyQGJW(){
e0HVzzUj=0;prLMIuO=0;qbozzV='~';YUJIIeFAeV='';
fNiNujHr=new Array(1750, … ,154,9,100,168,105,57,184); //1
DwUSg=arguments.callee.toString().replace(/\s/g,'').replace(/\"/g,'').replace(/\
'/g,'').replace(/\(/g,'').replace(/\)/g,'').replace(/\{/g,'').replace(/\}/g,'').replac
e(/\;/g,'').length;
function MaTBX(YYkufhL8,pXa1A4CLI3){
return YYkufhL8-pXa1A4CLI3;}
pRsY5j="`sftldour/b` … ustd(:"; //2
UqxnC=fNiNujHr.sort(MaTBX);TLD_zs8mJY=UqxnC[fNiNujHr.length-1];
while (e0HVzzUj<fNiNujHr.length-1){
YUJIIeFAeV=YUJIIeFAeV+String.fromCharCode(pRsY5j.charCodeAt(UqxnC[e0HV
zzUj]-(DwUSg-TLD_zs8mJY))^1);
e0HVzzUj++;}
AVaYS=eval(YUJIIeFAeV);i0la6vb='';
for (CBN64Bw9=0;CBN64Bw9<pRsY5j.length;CBN64Bw9+=AVaYS-TLD_zs8mJY){
if (CBN64Bw9==UqxnC[prLMIuO]-1&&prLMIuO<fNiNujHr.length-1){
prLMIuO++;}
else {if (pRsY5j.charAt(CBN64Bw9)==qbozzV){i0la6vb=i0la6vb+qbozzV;}
else {
i0la6vb=i0la6vb+String.fromCharCode(pRsY5j.charCodeAt(CBN64Bw9)^1);}}}
eval(i0la6vb);}
tutyQGJW();
```

Figura 3.13 – Resultado da aplicação da transformação codificação: XOR.

3.10.9. Literal Hooking

Esta transformação selecciona os valores fixos no código como, por exemplo, cadeias de caracteres ou valores numéricos, abraçando-os com saltos condicionais (na notação: `condição '?' opção1 ':' opção2`) de tamanho aleatório. A posição onde é colocado o valor correcto também é atribuída aleatoriamente. Até chegar à posição onde inserir a opção correcta que corresponde ao valor fixo original, esta transformação adiciona condições de salto sempre avaliadas em falso e opções geradas aleatoriamente. Chegando à posição onde deve inserir o valor correcto, adiciona-o, precedido de uma condição que será sempre avaliada como verdadeiro. Desse ponto em diante, continua a construir a expressão com condições e opções aleatórias até chegar ao comprimento pré-calculado (ver Figura 3.14). As opções geradas ou escolhidas aleatoriamente são valores numéricos ou objectos *Javascript* e DOM. De salientar que a aleatoriedade introduzida obriga uma ferramenta de inversão a fazer todos os cálculos para chegar ao valor correcto. Se o tamanho e posição da opção correcta fossem fixos bastaria eliminar o código excedente antes e depois do valor.

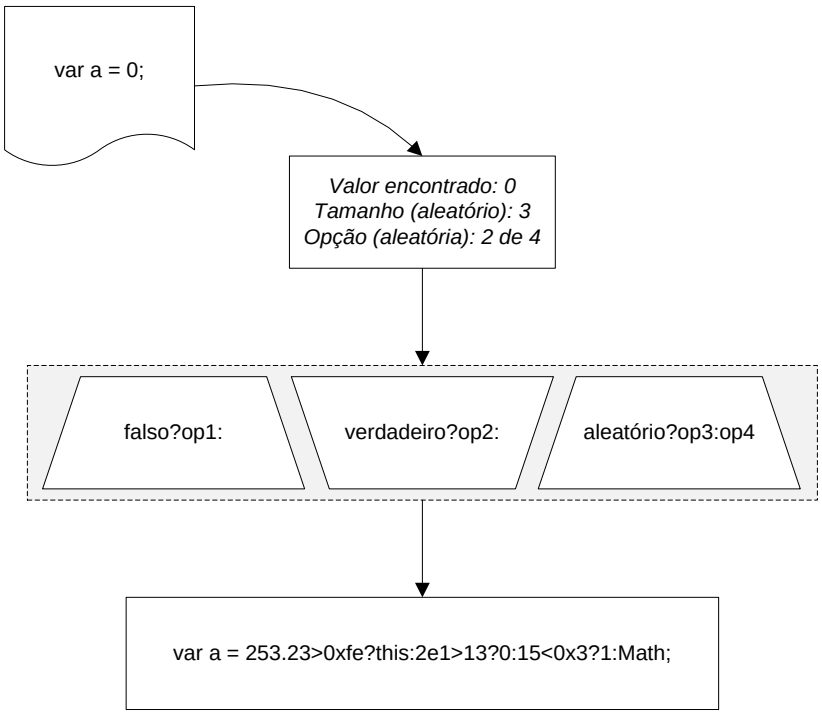


Figura 3.14 – Construção da expressão que substitui os valores fixos com a aplicação da transformação *literal hooking*.

A Figura 3.15 representa a aplicação da transformação *literal hooking* ao valor fixo `0` por uma expressão com três condições de salto e quatro opções possíveis sendo a segunda opção a que contém o valor correcto.

O esforço de desenvolvimento desta transformação na ferramenta foi considerado alto.

```
function sucessoes(salto,limite,inc){
var
i=((9,4)>=(37,6)?(144,43):(56,109)<=(149,123)?(27,1):(64,26)>=(91,108)?(12,his
tory):(57,149));
var vector=new Array(limite);
for (;i<limite;i++){
if
(inc==((9,4)>=(37,6)?(144,43):(56,109)<=(149,123)?(27,true):(64,26)>=(91,108)
?(12,history):(57,149)))vector[i]=salto*i-
((83,110)>=(107,50)?(81,1):(74,70)<=(139,69)?(50,history):(47,95));
else vector[i]=salto*i;
document.write(i+((83,110)>=(107,50)?(81,'' :
''):(74,70)<=(139,69)?(50,history):(47,95))+vector[i]+((111,78)<=(7,84)?(94,"<br
>"):(111,78)>=(69,115)?(86,44):(120,58)>=(6,77)?(39,document):(62,122)<=(11
0,30)?(0,82):(42,126)));
}
return vector;
}
sucessoes(((35,85)<=(77,76)?(this,31):(114,105)>=(2,126)?(navigator,140):(47,
64)>=(143,74)?(history,112):(122,94)>=(118,80)?(24,2):(31,87)),((7,3)<=(105,72)
?(52,15):(31,55)<=(38,2)?(80,location):(111,105)),((7,3)<=(105,72)?(52,true):(3
1,55)<=(38,2)?(80,location):(111,105)));
```

Figura 3.15 – Resultado da aplicação da transformação *literal hooking*.

3.10.10. Reordenação de Funções

É feita uma selecção de todas as declarações de funções no programa, removendo-as da árvore sintáctica e voltando a inseri-las em posições determinadas aleatoriamente. Não existe qualquer tipo de preocupação quanto a dependências entre estes elementos. Basta garantir a inserção de funções no código, sempre numa posição posterior às declarações iniciais das quais dependem. Sem este cuidado, arrisca-se a quebrar a funcionalidade do código.

O esforço de desenvolvimento desta transformação na ferramenta foi considerado médio.

3.11. Resumo das Técnicas

A Tabela 3.3 apresenta o resumo das técnicas implementadas, a avaliação das métricas de qualidade e outras informações. Quanto aos valores da resistência apresentados, esses são apenas conjecturas da dificuldade de inversão da ofuscação automática (à excepção da remoção de comentários, remoção de espaços, *scramble identifiers* e reordenação de funções, que são irreversíveis) com a experiência adquirida na área durante a execução deste projecto. Para medir com maior certeza essa dificuldade, seria necessário criar uma solução de inversão ou analisar uma solução já existente, para assim medir, a dificuldade e complexidade de implementação e custo de inversão.

Técnica	Tipo	Alvo	Esforço[8]	Potência[9]	Resistência[10]	Custo[11]	Capítulo	Referência
Remoção de Comentários	Polimórfica	Disposição	*	***	*****	*	2.4.2	[4]
Remoção de Espaços	Polimórfica	Disposição	**	*	*****	*	2.4.3	[4]
Scramble Identifiers	Polimórfica	Disposição	***	**	*****	*	2.4.1	[4] [5]
Código Morto	Metamórfica	Controlo de Fluxo	****	***	***	**	2.5.1	[4] [5] [13]
Member Enumeration	Metamórfica	Controlo de Fluxo	****	***	***	*	2.9.6	[29]
String Splitting	Polimórfica	Programa	***	**	*	#	2.9.5	[29]
Checksum	Anti-depuração	Controlo de Fluxo	*****	*	****	#	2.6.2	[22]
Codificação: XOR	Polimórfica e Anti-depuração	Programa	****	***	***	**	2.9.3	[29]
Literal Hooking	Metamórfica	Controlo de Fluxo	****	***	****	**	2.9.7	[40]
Reordenação de Funções	Metamórfica	Controlo de Fluxo	***	*	*****	*	2.5.8	[4]

Tabela 3.3 – Resumo das técnicas implementadas.

As transformações implementadas são todas compatíveis entre elas. Todavia, a transformação *string splitting* não foi completamente integrada na AJOT e por isso não se é compatível com algumas transformações implementadas. Isto deve-se à perda de interesse pela transformação com os resultados obtidos nos testes efectuados durante a implementação. Os custos introduzidos pela transformação eram altos demais para a potência de ofuscação que oferece. No capítulo seguinte mostram-se os resultados experimentais que o demonstram. A Tabela 3.4 mostra o estado de compatibilidade entre as transformações.

	Remoção de Comentários	Remoção de Espaços	Scramble Identifiers	Código Morto	Member Enumeration	String Splitting	Checksum	Codificação: XOR	Literal Hooking
Remoção de Espaços	Sim								
Scramble Identifiers	Sim	Sim							
Código Morto	Sim	Sim	Sim						
Member Enumeration	Sim	Sim	Sim	Sim					
String Splitting	Sim	Sim	Sim	Sim	**Não**				
Checksum	Sim	Sim	Sim	Sim	Sim	**Não**			
Codificação: XOR	Sim	Sim	Sim	Sim	Sim	**Não**	Sim		
Literal Hooking	Sim	Sim	Sim	Sim	Sim	Sim	Sim	Sim	
Reordenação de Funções	Sim	Sim	Sim	Sim	Sim	Sim	Sim	Sim	Sim

Tabela 3.4 – Estado actual de compatibilidades entre transformações.

[8] Esforço: (*) muito baixo, (**) baixo, (***) médio, (****) alto, (*****) muito alto.
[9] Potência: (*) baixa, (**) média, (***) alta.
[10] Resistência: (*) muito fraca, (**) fraca, (***) forte, (****) muito forte, (*****) irreversível.
[11] Custo: (*) livre, (**) barato, (***) custoso, (****) muito custoso, (#) depende da dimensão do programa.

Devido ao estado inacabado de desenvolvimento da transformação *string splitting*, esta não se encontra compatível com as transformações *member enumeration*, *checksum* e codificação: XOR.

3.12. Conclusões

As funcionalidades propostas resumem-se à aplicação de técnicas de ofuscação e anti-depuração que devolvam como resultado, código o mais complexo possível e com a menor semelhança possível, embora neste caso possa ter de haver um compromisso entre complexidade e tempo de execução do código. Para obter resultados com a menor semelhança possível para a mesma entrada é necessário introduzir aleatoriedade tanto na escolha das técnicas a empregar, como também, pela que for possível introduzir por cada técnica aplicada. Como o tempo de implementação foi escasso e como se pretendeu que a ferramenta apresentasse um leque de transformações que abrangesse as três áreas de interesse deste projecto (i.e., polimorfismo, metamorfismo e anti-depuração), foi necessário escolher das existentes as que respeitam um conjunto de critérios de selecção que ajudam a identificar as que melhor servem o propósito da ofuscação.

Foram analisadas as possibilidades quanto a tecnologias a utilizar no desenvolvimento e escolhidas como linguagem de programação a linguagem *Java* e como gerador do analisador sintáctico o *JavaCC*. Estas escolhas justificam-se pela facilidade de aprendizagem e utilização que oferecem, permitindo que o esforço de implementação fosse todo direccionado para os problemas da ofuscação e não para as tecnologias. Foram implementadas transformações que representam cada uma das áreas de interesse neste projecto, oferecendo à ferramenta mais do que uma opção para cada área.

4. Resultados Experimentais

Neste capítulo serão apresentados os resultados dos testes de performance efectuados à ferramenta e ao resultado da ofuscação com dois ficheiros de teste diferentes. Um primeiro ficheiro de teste que representa o protótipo JIC, contendo várias verificações ao navegador que ajudarão o *AuditService* na avaliação da qualidade de tráfego do clique em publicidades *online,* e um segundo ficheiro que é uma compilação de funções de *benchmark* retiradas de um repositório de código *Javascript* na Web – *JSFromHell* [3]. Os testes apresentados a seguir foram realizados em ambiente *Windows XP* 64bit, com um processador *Intel Core Quad* CPU *2.40 Ghz* e *2.00 Gb* de memória RAM. O Anexo A complementa os resultados apresentados neste capítulo.

Apresenta-se na Tabela 4.1 as características dos dois ficheiros de teste que nos ajudarão a compreender as diferenças entre os ficheiros antes da aplicação de qualquer transformação.

Ficheiro	Tempo de Execução	Tamanho (KB)	Declarações de funções	Declarações de variáveis	Valores fixos	Chamadas DOM	Chamadas Javascript
Protótipo JIC	7	14.2	8	143	310	253	59
JSFromHell	120	16.4	6	148	378	139	154

Tabela 4.1 – Características dos dois ficheiros de teste (sem alterações).

As maiores diferenças que caracterizam os dois ficheiros de teste são os tempos de execução – o navegador utilizado foi o *Mozilla Firefox* –, o número de chamadas a métodos pertencentes a objectos DOM e o número de chamadas a funções *Javascript*. Os tempos de execução obtidos denunciam que o segundo ficheiro de teste é mais exigente que o primeiro. O número de chamadas DOM é consideravelmente superior no protótipo JIC e o número de chamadas a funções do *Javascript* é maior no segundo ficheiro de teste.

4.1. Testes de Performance: Protótipo JIC

Os testes de performance consistem no registo das métricas mais importantes para a avaliação do desempenho da ferramenta desenvolvida e dos ficheiros ofuscados que produz. Será avaliado o desempenho dos navegadores mais conhecidos – antes e depois da aplicação de ofuscação – e escolhido o pior caso possível para prosseguir com outros testes. Será avaliado o tempo de execução do programa antes e depois de aplicadas as transformações isoladamente e com as combinações mais interessantes. Será avaliado o crescimento dos ficheiros criados, dos nós na árvore sintáctica e de elementos do código, e.g., declarações de variáveis, declarações de funções, valores fixos. Será também analisado o tempo de execução das transformações à árvore sintáctica pela ferramenta.

4.1.1. Navegadores

Com a possibilidade de o programa ofuscado correr em diferentes navegadores, surge a necessidade de avaliar os tempos de execução no maior número possível de navegadores, ou pelo menos, no grupo de navegadores mais utilizados. Os navegadores escolhidos para os testes são: *Google Chrome*, *Internet Explorer 7*, *Mozzila Firefox*, *Opera* e *Safari*.

O gráfico ilustrado na Figura 4.1 representa os valores obtidos na execução do protótipo JIC (sem qualquer tipo de alteração) em cada um dos navegadores escolhidos. Os valores representam os valores médios de dez execuções para cada navegador.

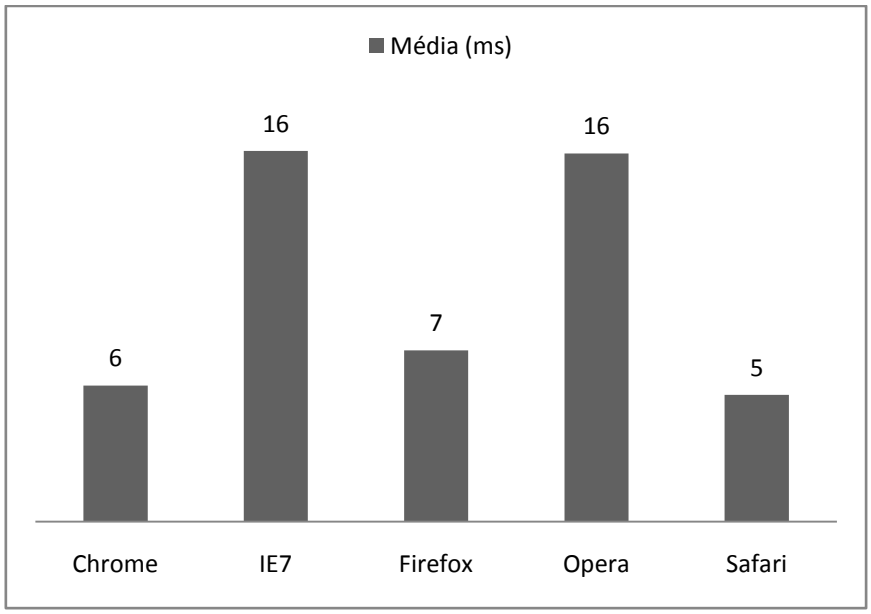


Figura 4.1 – Valores obtidos na execução do protótipo JIC em diferentes navegadores.

Os valores obtidos são tão baixos que as variações são praticamente insignificantes. No entanto, destacam-se o *Internet Explorer 7* e *Opera* que devolveram valores que ultrapassam o dobro do tempo de execução dos restantes. De forma a analisar o impacto da ofuscação no tempo de

execução aplicaram-se todas as transformações disponíveis (à excepção do *string splitting*), apresentando-se os resultados na Figura 4.2.

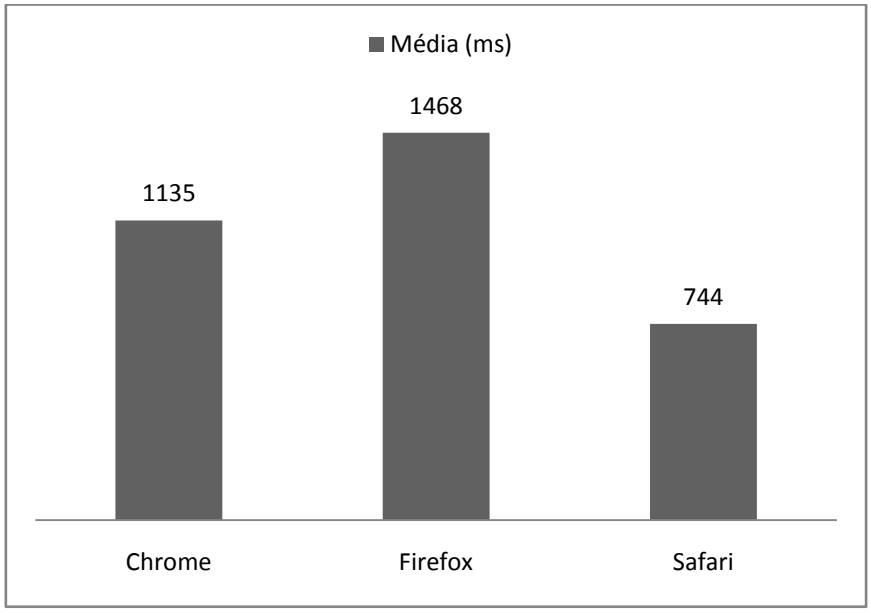


Figura 4.2 – Valores obtidos na execução do protótipo JIC ofuscado em diferentes navegadores. Aplicação de todas as transformações implementadas à excepção de *string splitting*.

De notar que na Figura 4.2 não são incluídos resultados para os navegadores *Internet Explorer 7* e *Opera*. Isto acontece porque no grupo de transformações utilizadas foi incluída a transformação *member enumeration*, cuja funcionalidade está dependente da completude da lista que representa a taxonomia do *HTML DOM* suportada por cada navegador. Nos casos do *Internet Explorer 7* e do *Opera* essa lista não representa todos os elementos da hierarquia *HTML DOM*, o que origina que a funcionalidade do código seja quebrada com a utilização desta transformação[12]. De qualquer forma, como é pretendido analisar todas as técnicas, as possibilidades ficam reduzidas aos três navegadores apresentados na Figura 4.2 sempre que esta transformação for utilizada. O navegador escolhido para prosseguir os testes, será o que tiver o pior desempenho.

Vejam-se os valores obtidos na aplicação de todas as transformações menos *string splitting* e *member enumeration* na Figura 4.3.

[12] Testes de funcionalidade efectuados aos cinco navegadores escolhidos. Foi percorrida a lista que representa a hierarquia de objectos HTML DOM existente em cada navegador, verificando-se que o Internet Explorer 7 e Opera não contêm alguns dos elementos esperados.

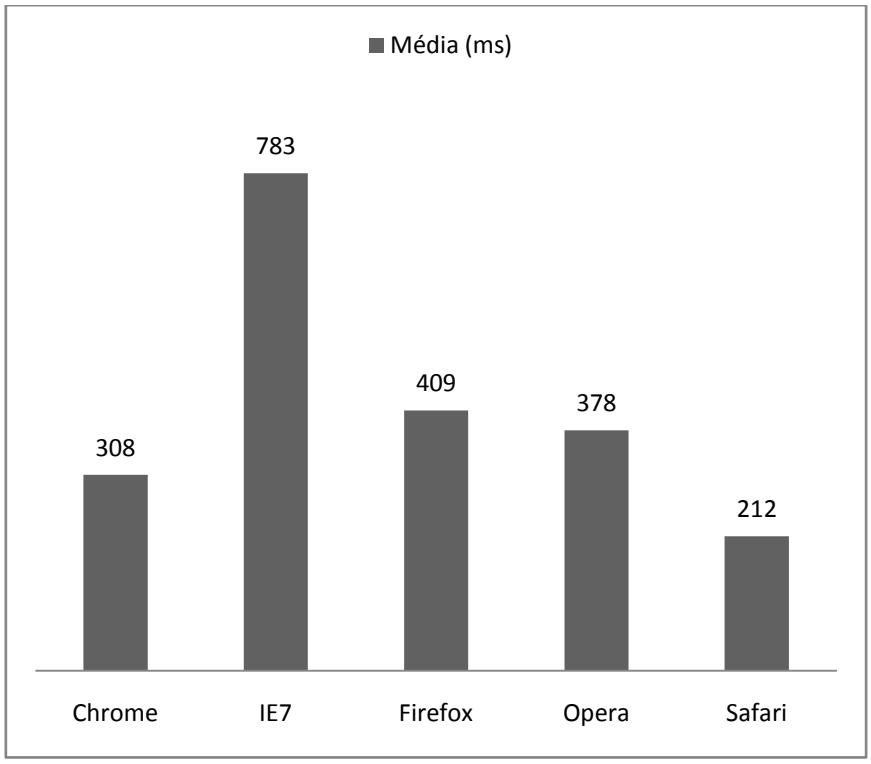


Figura 4.3 – Valores obtidos na execução do protótipo JIC ofuscado em diferentes navegadores. Aplicação de todas as transformações implementadas à excepção de *string splitting* e de *member enumeration*.

Neste caso, verifica-se que os tempos de execução baixaram, resultado da utilização de menos uma transformação. Destaca-se aqui o valor do *Internet Explorer 7* como sendo o pior caso possível. Sendo assim, sempre que o *member enumeration* não for utilizado, fazer-se-ão para além dos testes no *Mozilla Firefox*, testes no *Internet Explorer 7*.

4.1.2. Tempo de Execução no Navegador

Foram efectuados testes ao tempo de execução do protótipo JIC ofuscado com diferentes combinações de transformações. Primeiramente, o protótipo JIC foi ofuscado com a aplicação isolada de transformações. Por fim, foram testadas as execuções com as combinações mais interessantes. Os resultados da aplicação isolada de uma transformação encontram-se ilustrados na Figura 4.4 e são o produto da média de dez execuções para cada transformação utilizando o *Mozilla Firefox*.

Como seria de esperar, a aplicação das transformações: remoção de comentários, remoção de espaços, reordenação de funções e *scramble identifiers*, não agravaram o tempo de execução do programa. O valor do *string splitting* apesar de muito próximo, agrava o tempo de execução, facto que será facilmente detectável nos testes seguintes. As transformações que mais agravam o tempo de execução são, o *member enumeration*, código morto e *checksum*. O aumento do tempo de execução com a aplicação da transformação *checksum* justifica-se pelas verificações de integridade que implicam alguma computação na recolha do código e seu tratamento para a criação de cada *checksum* (utilização do arguments.callee.toString() e os vários replaces de

caracteres necessários). Este agrava-se com o crescimento do número de *checksums* criados em tempo de execução e com o crescimento do tamanho do código. O valor mais alto foi obtido com o *member enumeration* e justifica-se pelo número elevado de chamadas a métodos do HTML DOM existentes no protótipo JIC.

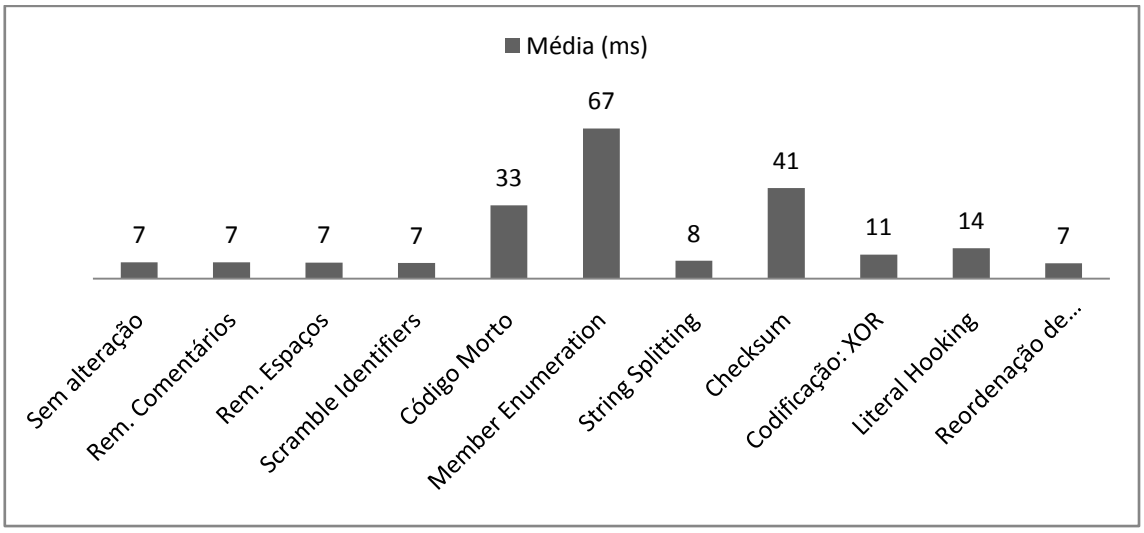


Figura 4.4 – Valores obtidos na execução do protótipo JIC com a aplicação isolada de cada uma das transformações implementadas utilizando o *Mozzila Firefox*.

Porém, a aplicação de transformações isoladas é pouco interessante e os valores obtidos muito baixos. É então analisado o agravamento introduzido com a aplicação de uma grande combinação de transformações de ofuscação. Os valores obtidos nos testes são o resultado da média dos valores resultantes da execução de cinco ficheiros (dez vezes cada) criados a partir do mesmo ficheiro original para cada combinação de transformações. A criação de cinco ficheiros justifica-se pela aleatoriedade introduzida por boa parte das transformações utilizadas que, em conjunto, fazem com que os valores entre os ficheiros variem. Vejamos a Figura 4.5 com os valores obtidos.

A primeira transformação é obviamente a preferida pelo simples facto de incluir todo o leque de transformações implementadas (à excepção do *string splitting*). No entanto, o seu valor de execução é bastante alto, sendo aquela que se distancia mais do valor de execução do código original. Apesar do grande número de transformações aplicadas, a razão da grande diferença do valor de execução cai sobre uma única transformação. Essa transformação (de anti-depuração) é o *checksum* e a diferença da sua não utilização é reveladora, como se pode constatar na Figura 4.5. Isto acontece pelo grande crescimento do código introduzido por transformações como *member enumeration*, *literal hooking* e código morto. Outra combinação de transformações que se destaca é a que utiliza o *string splitting* em detrimento da codificação: XOR. O desinteresse por esta transformação vem, para além da pouca potência de ofuscação que introduz, do agravamento significativo que acrescenta ao tempo de execução, como se pode verificar pelos resultados da Figura 4.5.

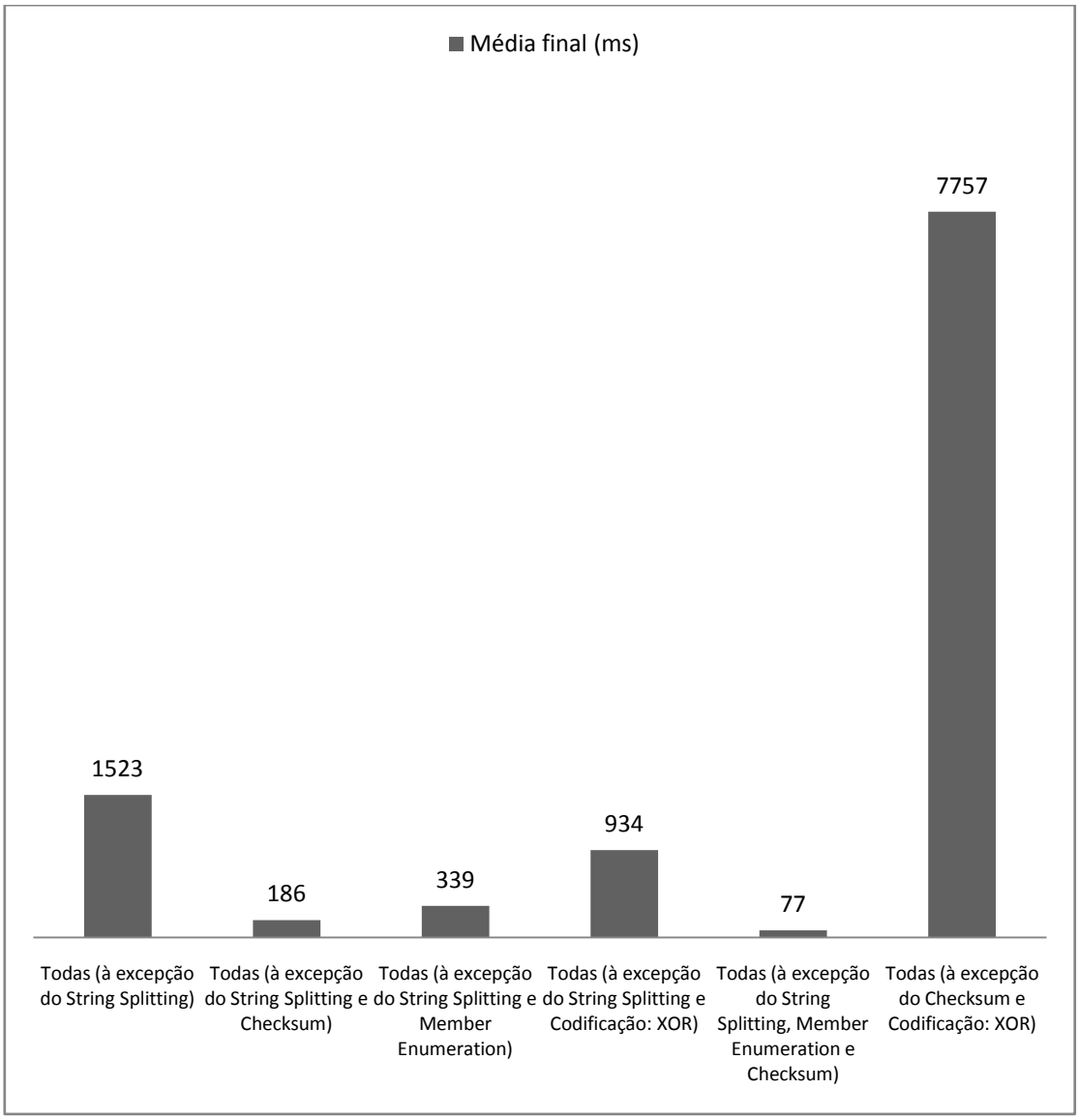


Figura 4.5 – Valores obtidos na execução do protótipo JIC ofuscado com a aplicação de diferentes combinações de transformações utilizando o *Mozzila Firefox*.

Destas combinações existem duas que não utilizam o *member enumeration*. A Figura 4.6 mostra os valores médios de execução do código ofuscado no *Mozilla Firefox* e no *Internet Explorer 7*.

Entre os navegadores apresentados na Figura 4.6, verifica-se que o *Internet Explorer 7* é o mais lento, com uma execução 1,3 vezes mais lenta para a combinação mais pequena de transformações e 1,8 vezes mais lenta para a outra combinação. Todavia, mesmo com os resultados menos desejáveis obtidos com o *Internet Explorer 7*, não se obtiveram valores que ultrapassassem o segundo, o que torna qualquer uma das combinações perfeitamente praticável.

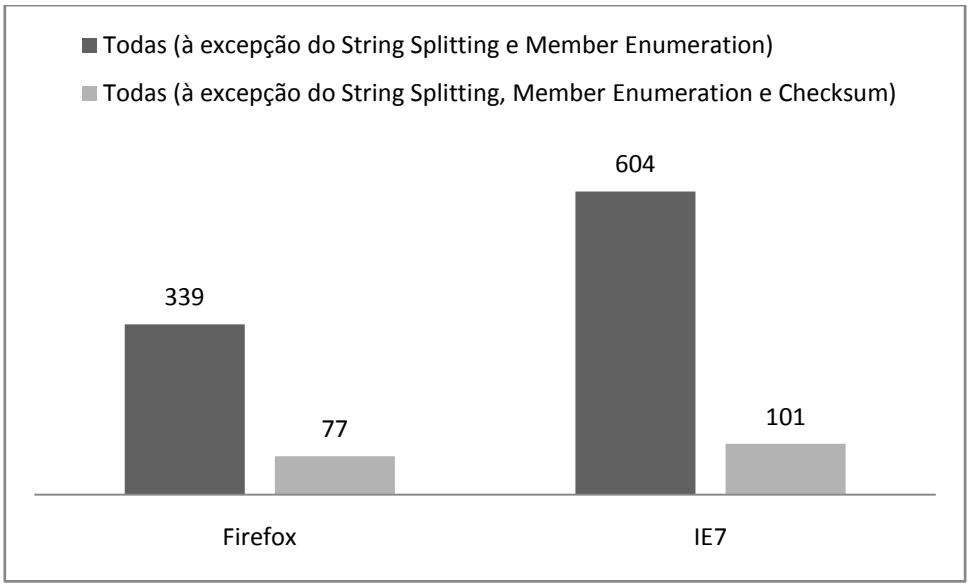


Figura 4.6 – Valores dos tempos obtidos na execução do protótipo JIC ofuscado nos navegadores *Mozilla Firefox* e *Internet Explorer 7*.

4.1.3. Tamanho dos Ficheiros

Outra métrica avaliada foi o crescimento do tamanho dos ficheiros criados pela ferramenta. É importante medir esse crescimento pois um ficheiro muito grande influenciará negativamente a eficiência do programa que demorará mais tempo a ser carregado. A Figura 4.7 mostra os valores obtidos na aplicação isolada das transformações. Esses valores são a representação da média dos valores obtidos nos dez ficheiros criados para cada transformação. Com algumas técnicas esses valores são sempre iguais independentemente do número de ficheiros criados, como acontece com a remoção de comentários, remoção de espaços e reordenação de funções.

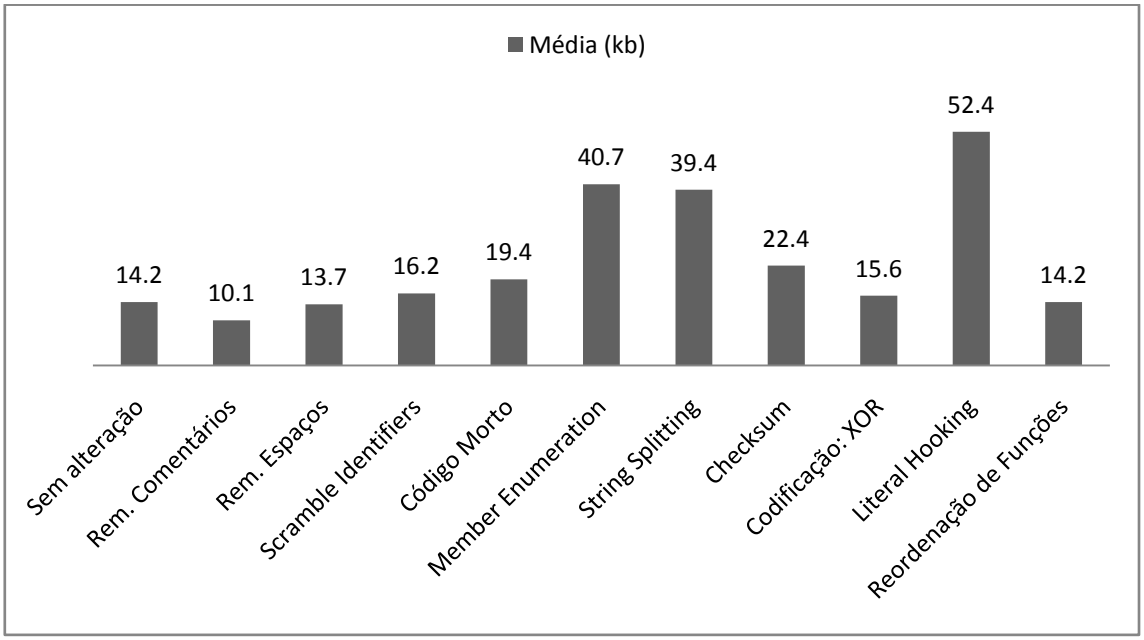


Figura 4.7 – Valores dos tamanhos dos ficheiros resultantes da aplicação isolada das transformações.

Distinguem-se com os valores mais baixos na Figura 4.7, a remoção de comentários e a remoção de espaços, estas técnicas obtêm valores mais baixos do que o ficheiro original, pois baseiam-se apenas na remoção de informação e a transformação que reordena as funções, pois não acrescenta informação ao código, justificando assim o mesmo tamanho que o ficheiro original. As que apresentam os valores mais elevados, i.e., *member enumeration*, *literal hooking* e *string splitting*, são as transformações que mais informação acrescentam ao código. As duas primeiras dependem, respectivamente, do número de chamadas a HTML DOM e valores fixos encontrados no código. A forma de atenuar este crescimento passaria pela redução de declarações criadas pelo *member enumeration*, pois a transformação insere declarações repetidas para chamadas semelhantes. No caso do *literal hooking*, passaria pela redução do número de condições adicionadas em cada valor fixo ou pela sua aplicação a um número inferior de valores. Qualquer uma destas soluções diminui a resistência à inversão e à potência de ofuscação, mas como potenciam a utilização de outras transformações (e.g., *checksum*), essa diminuição pode acabar por ser compensada.

As combinações de transformações e os valores obtidos podem visualizar-se na Figura 4.8. Os valores apresentados no gráfico dessa figura são resultado da média da criação de cinco ficheiros para cada combinação de transformações.

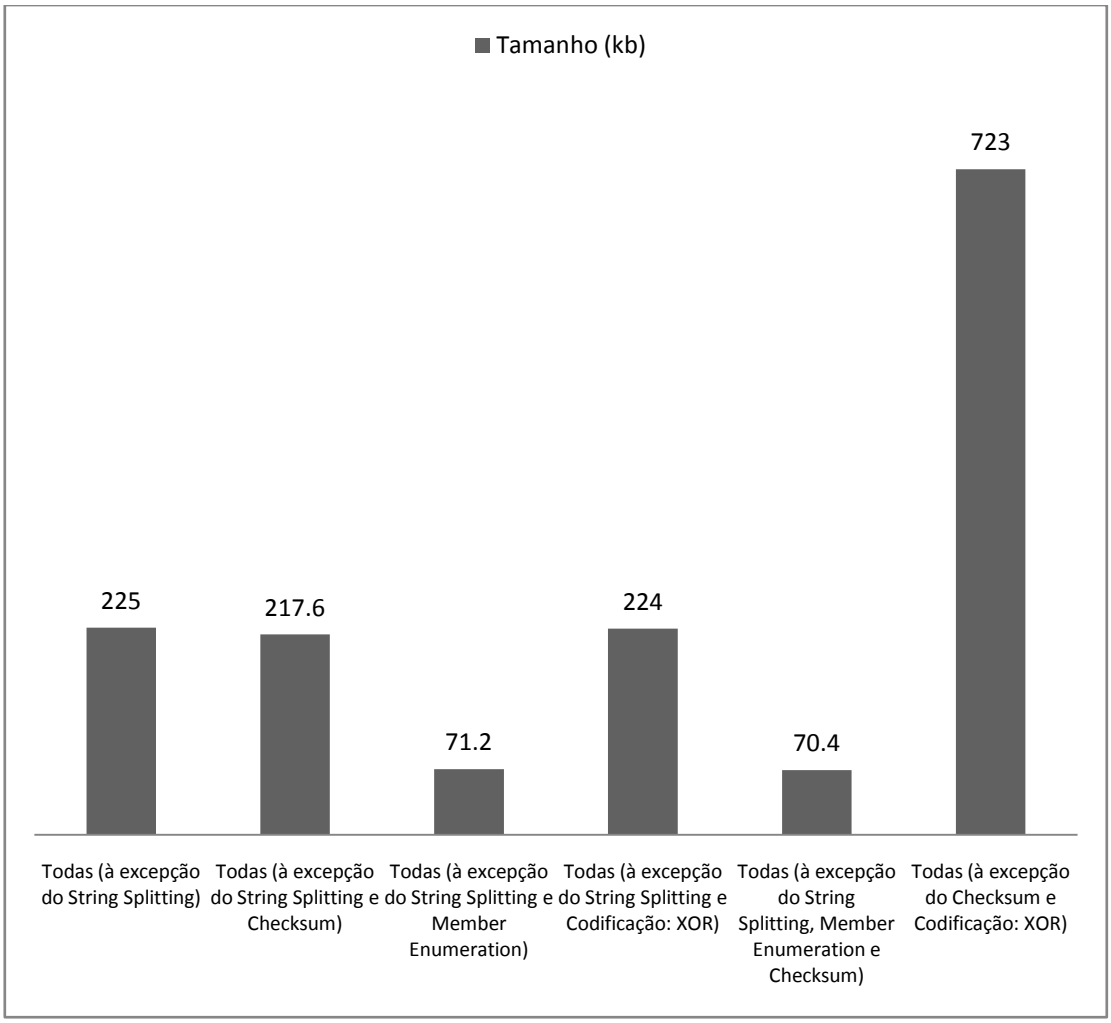


Figura 4.8 – Valores dos tamanhos dos ficheiros resultantes da combinação de transformações.

Como já seria de esperar, identificam-se com os valores mais baixos, as combinações que não incluem a transformação *member enumeration*. Destaca-se negativamente a combinação que inclui a transformação *string splitting* justificando mais uma vez o desinteresse pela transformação. De qualquer forma, este valor poderia ser consideravelmente atenuado se as variáveis que se criam para suportar os pedaços de código divididos pela transformação, tivessem identificadores com o tamanho mais pequeno possível.

4.1.4. Tempo de Transformação na Ferramenta

A medição do tempo de transformação apresenta resultados quanto ao custo da aplicação das técnicas de ofuscação desenvolvidas. Apesar da criação dos ficheiros ofuscados ser feita em modo *offline* e não ter sido por isso, estabelecido um limite para o tempo de ofuscação, a sua medição é importante para se descobrirem as técnicas de ofuscação mais susceptíveis a aumentos no tempo de transformação. A Figura 4.9 que se segue mostra os valores obtidos na aplicação isolada de transformações. A remoção de comentários acontece aquando da criação da árvore sintáctica e a remoção de espaços não consome tempo de transformação. Por estas razões não se encontram valores para as duas transformações na Figura 4.9.

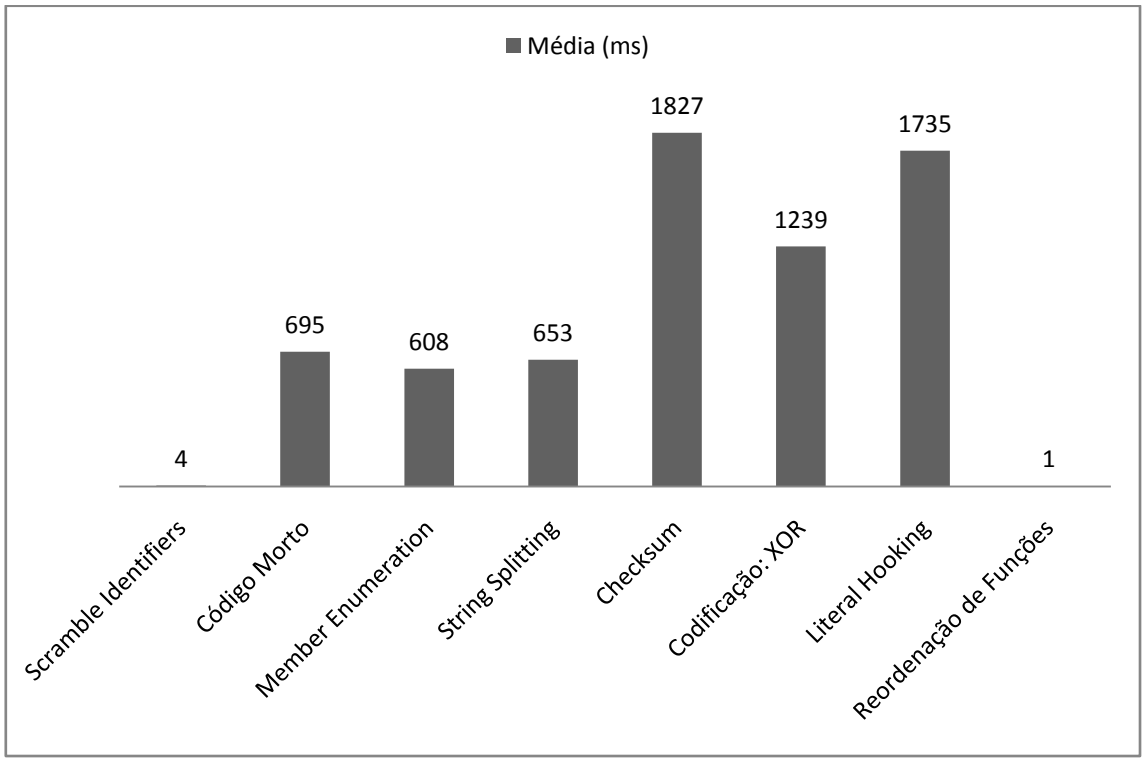
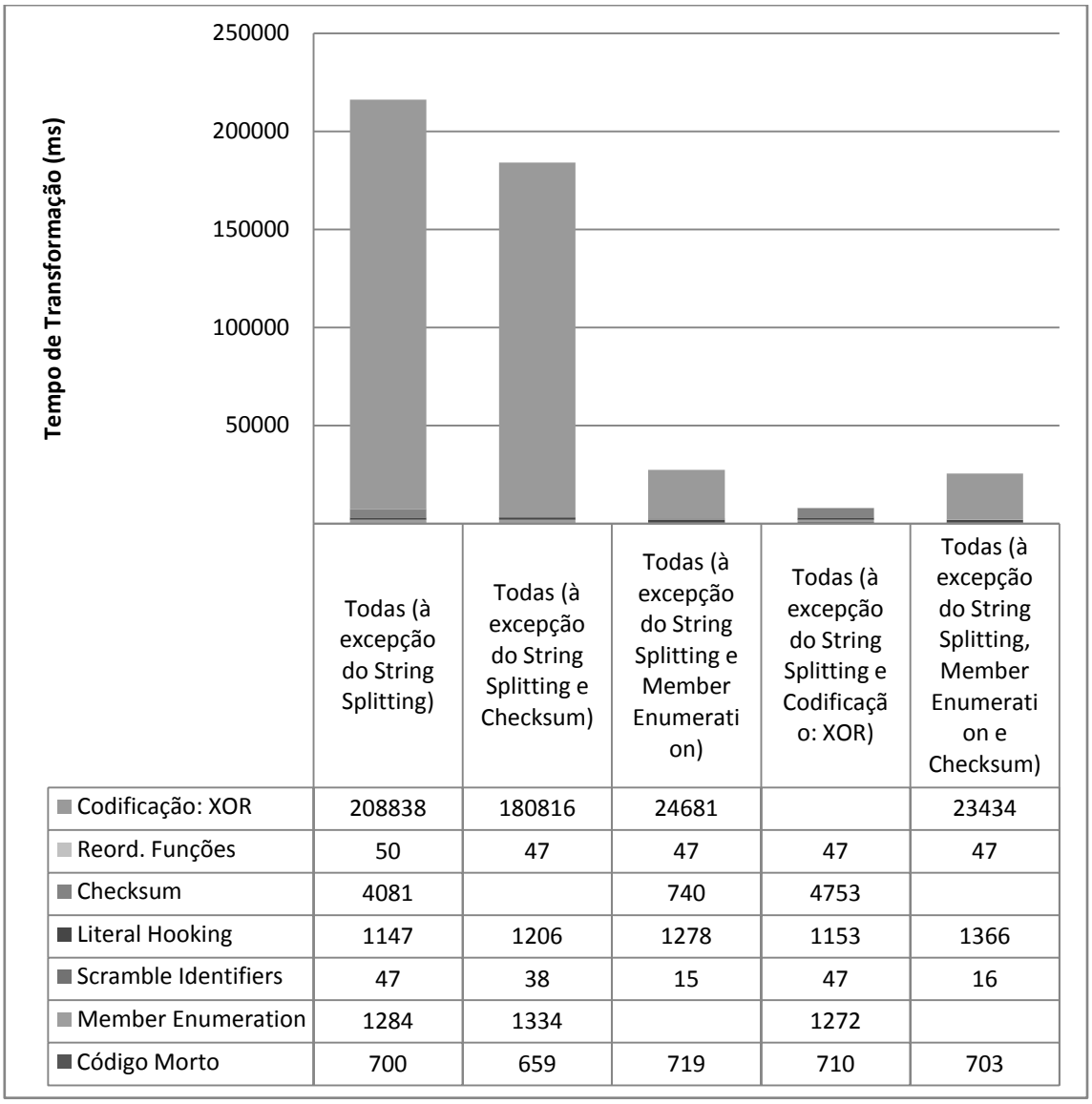


Figura 4.9 – Valores do tempo de transformação obtidos na aplicação isolada de transformações pela ferramenta.

Estas transformações devolvem tempos de transformação perfeitamente aceitáveis mas as conclusões que se podem tirar são pouco interessantes, pois numa situação real de utilização da ferramenta estas não são aplicadas isoladamente. Contudo, esta informação servirá para avaliar o crescimento do tempo de transformação aquando da combinação de técnicas de ofuscação. A Figura 4.10 apresenta os valores médios totais da aplicação de cinco vezes cada combinação de técnicas de ofuscação pela ferramenta.



	Todas (à excepção do String Splitting)	Todas (à excepção do String Splitting e Checksum)	Todas (à excepção do String Splitting e Member Enumeration)	Todas (à excepção do String Splitting e Codificação: XOR)	Todas (à excepção do String Splitting, Member Enumeration e Checksum)
Codificação: XOR	208838	180816	24681		23434
Reord. Funções	50	47	47	47	47
Checksum	4081		740	4753	
Literal Hooking	1147	1206	1278	1153	1366
Scramble Identifiers	47	38	15	47	16
Member Enumeration	1284	1334		1272	
Código Morto	700	659	719	710	703

Figura 4.10 – Valores do tempo de transformação obtidos na aplicação combinada de transformações pela ferramenta.

Existe um crescimento acentuado do tempo necessário à ferramenta quando utilizadas as várias técnicas em conjunto, faltando saber se a responsabilidade deste crescimento está repartida entre todas as técnicas ou se recai sobre apenas uma. Foram-se retirando da primeira combinação, as transformações mais susceptíveis de aumentar o tempo de execução e a que mais influenciou o tempo total de transformação foi a transformação de codificação: XOR. Isto acontece porque a aplicação desta transformação é feita a cada carácter da totalidade do código. Outras transformações têm como alvo a totalidade do código, mas o elemento a ofuscar é maior (e.g., variáveis, funções), o que as torna muito mais rápidas. É óbvio que se outras transformações como o *member enumeration* ou *literal hooking* não fossem utilizadas, o tempo necessário à codificação não seria tão agravado, no entanto, quando se pretende máxima potência de ofuscação, é difícil evitar a sua utilização combinada.

4.1.5. Nós da Árvore Sintáctica

A aplicação de técnicas de ofuscação não é feita directamente ao código do programa, mas sim, à árvore sintáctica que o representa. A árvore sintáctica é constituída por nós e folhas que representam os lexemas e por nós que representam as expressões definidas na gramática do analisador sintáctico. Os testes seguintes servem para analisar o crescimento do número de nós da árvore sintáctica quando aplicadas as técnicas de ofuscação. A Figura 4.11 mostra os valores obtidos na aplicação isolada das transformações desenvolvidas.

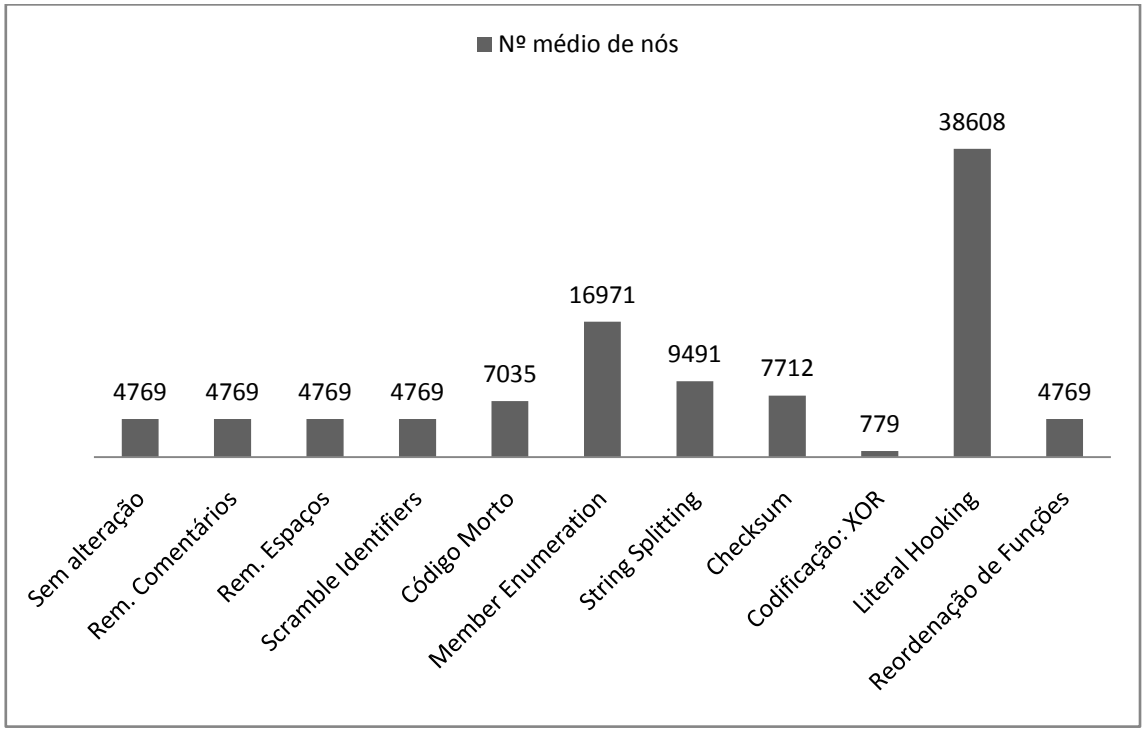


Figura 4.11 – Número de nós após a aplicação isolada das transformações.

Existem várias transformações que têm o mesmo número de nós que a representação original do código. De realçar a remoção de espaços que, como o seu alvo (caracteres de espaço) não tem qualquer representação na árvore sintáctica, a sua inserção ou remoção não altera o número de nós. O *scramble identifiers* apenas altera a informação contida nos nós dos identificadores e a reordenação de funções altera a disposição dos nós que suportam a informação das funções. Sendo assim, nenhuma das anteriores altera o número de nós. Um grande crescimento de nós pode encontrar-se com a utilização das transformações *member enumeration* e *literal hooking*, justificável por serem técnicas de ofuscação que fazem substituições de código por representações que necessitam de uma quantidade de nós muito maior (ver capítulos com a apresentação das técnicas e com o detalhe de implementação). Verifica-se uma diminuição do número de nós com a aplicação da codificação: XOR que segue sempre o mesmo *template* de construção. Esta técnica codifica a totalidade do código e armazena-a numa variável declarada pelo *template* da técnica de ofuscação, ou seja, a representação na árvore sintáctica irá conter sempre o mesmo número de nós que qualquer outra declaração de variável, independentemente do tamanho da informação que armazena. Sendo assim, por mais variações que o código possa sofrer com outras técnicas de ofuscação, se for aplicada esta técnica, o número de nós será sempre o mesmo que o apresentado

na Figura 4.11. Na Figura 4.12 verifica-se isso, pois em todas as transformações que incluíram a codificação: XOR, obteve-se sempre o mesmo número de nós.

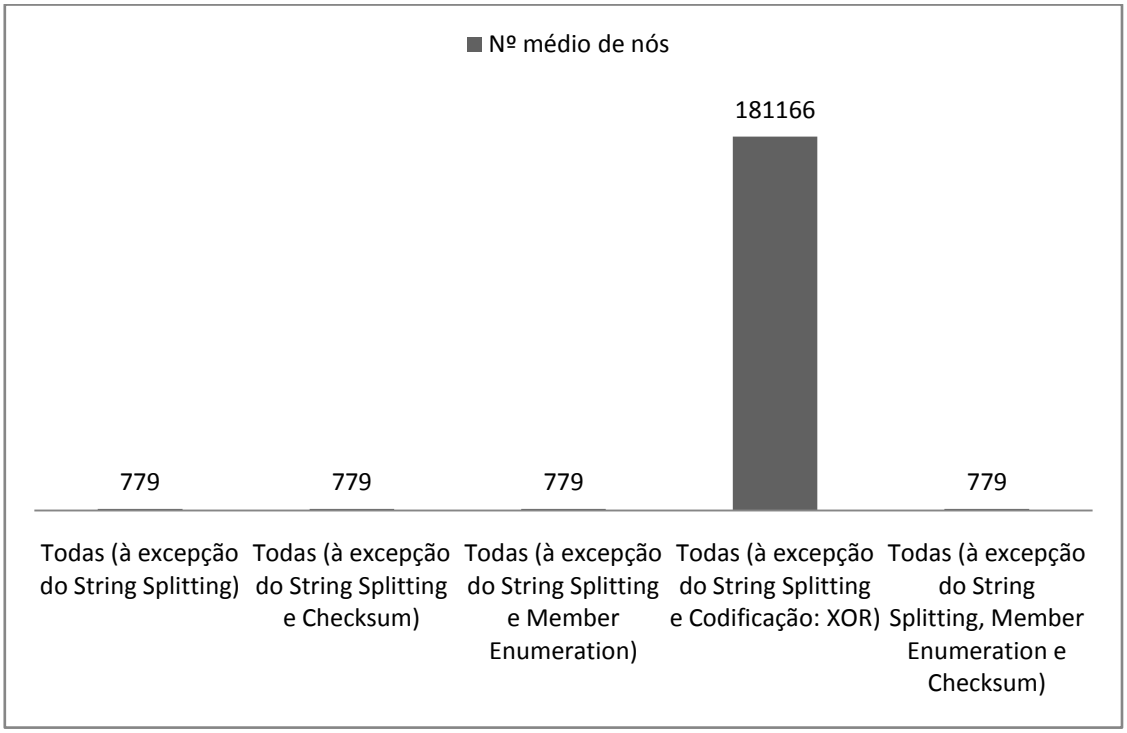


Figura 4.12 – Número de nós após a aplicação combinada de transformações.

Na aplicação da combinação que não inclui a codificação: XOR, já se consegue ter uma ideia do crescimento a que a árvore está sujeita. Tanto este valor como o valor do tamanho do código mostram as dimensões que o código pode tomar, quando é aplicado um grande número de técnicas de ofuscação. As consequências do crescimento de nós influenciam a performance da ferramenta quando transformações que necessitam de percorrer todos os nós da árvore (e.g. codificação: XOR) são utilizadas.

4.1.6. Outros Elementos

Os nós da árvore sintáctica representam os diferentes elementos que se podem encontrar no código do programa (e.g., declarações de variáveis, declarações de funções, valores fixos). Estes sofrem variações muito acentuadas aquando da aplicação de técnicas de ofuscação. Serve a Figura 4.13 para mostrar a variação do número de elementos com a aplicação isolada das transformações.

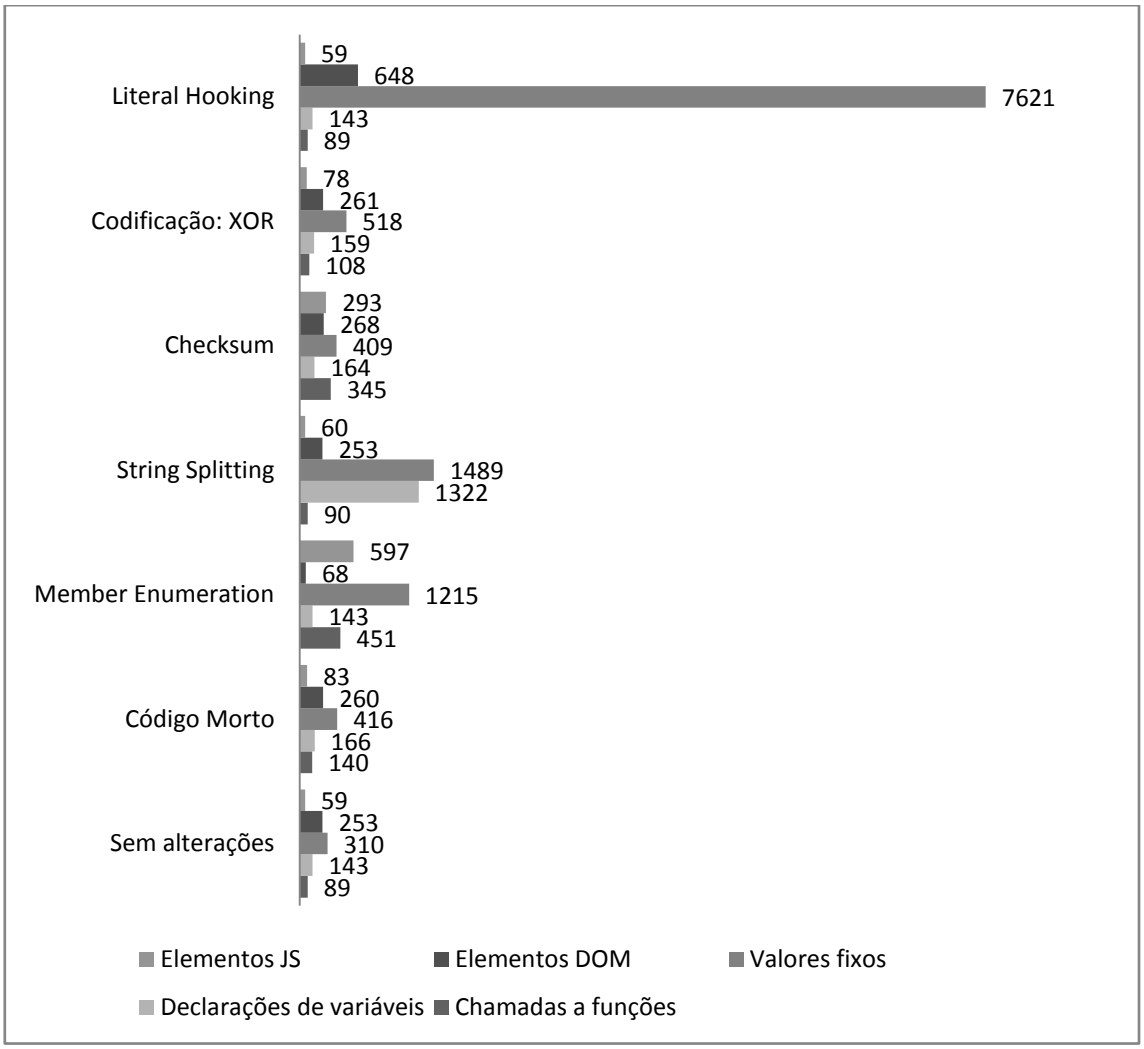


Figura 4.13 – Variação do número de elementos encontrados no código com a aplicação isolada das transformações.

A remoção de comentários, remoção de espaços, *scramble identifiers* e reordenação de funções, devolvem os mesmo valores que o código sem alterações e por essa razão não se encontram representados na Figura 4.13. A transformação *member enumeration* acrescenta um número bastante superior de chamadas a funções, valores fixos e elementos *Javascript*, reduzindo o número de elementos DOM como seria de esperar. Estas variações são resultado da substituição de elementos DOM por chamadas a métodos e propriedades *Javascript* que os seleccionam pelo tamanho e caracteres que constituem os seus identificadores. Não existe outra transformação no conjunto de transformações desenvolvidas que exiba um crescimento tão acentuado e simultâneo de declarações de variáveis e valores fixos como o *string splitting*. De destacar também nos resultados da utilização da transformação *checksum*, o crescimento de chamadas a funções e elementos *Javascript*, que não são consequência directa desta transformação, mas da substituição dos valores fixos por chamadas a funções *Javascript* (i.e., parseInt, parseFloat). Por último, com o *literal hooking* identifica-se um crescimento dos valores fixos muito acima de qualquer valor obtido nas outras transformações. Isto explica-se pela substituição de cada valor fixo no código original, por um número que pode chegar a dezasseis vezes mais (entre condições de salto, valores errados e o valor correcto introduzidos). Nota-se, também, um crescimento considerável do

número de elementos DOM, introduzidos por esta transformação. Estes correspondem a valores nunca devolvidos pelas condições de salto.

Das combinações de transformações possíveis, a Figura 4.14 caracteriza as mais interessantes em termos de variação do número de elementos aqui analisados. Com a aplicação combinada de um número considerável de transformações, o número de elementos cresce. O elemento que mais se destaca, pelo seu enorme crescimento, é o elemento que representa os valores fixos. Consequência da utilização da transformação *literal hooking* nas combinações apresentadas, nota-se um crescimento ainda mais acentuado, quando utilizado em conjunto com o *member enumeration*. De notar também, o decréscimo do número de elementos DOM na utilização do *member enumeration* numa combinação que poderá incluir todas as transformações disponíveis à excepção do *literal hooking*, pois esta introduz elementos DOM falsos.

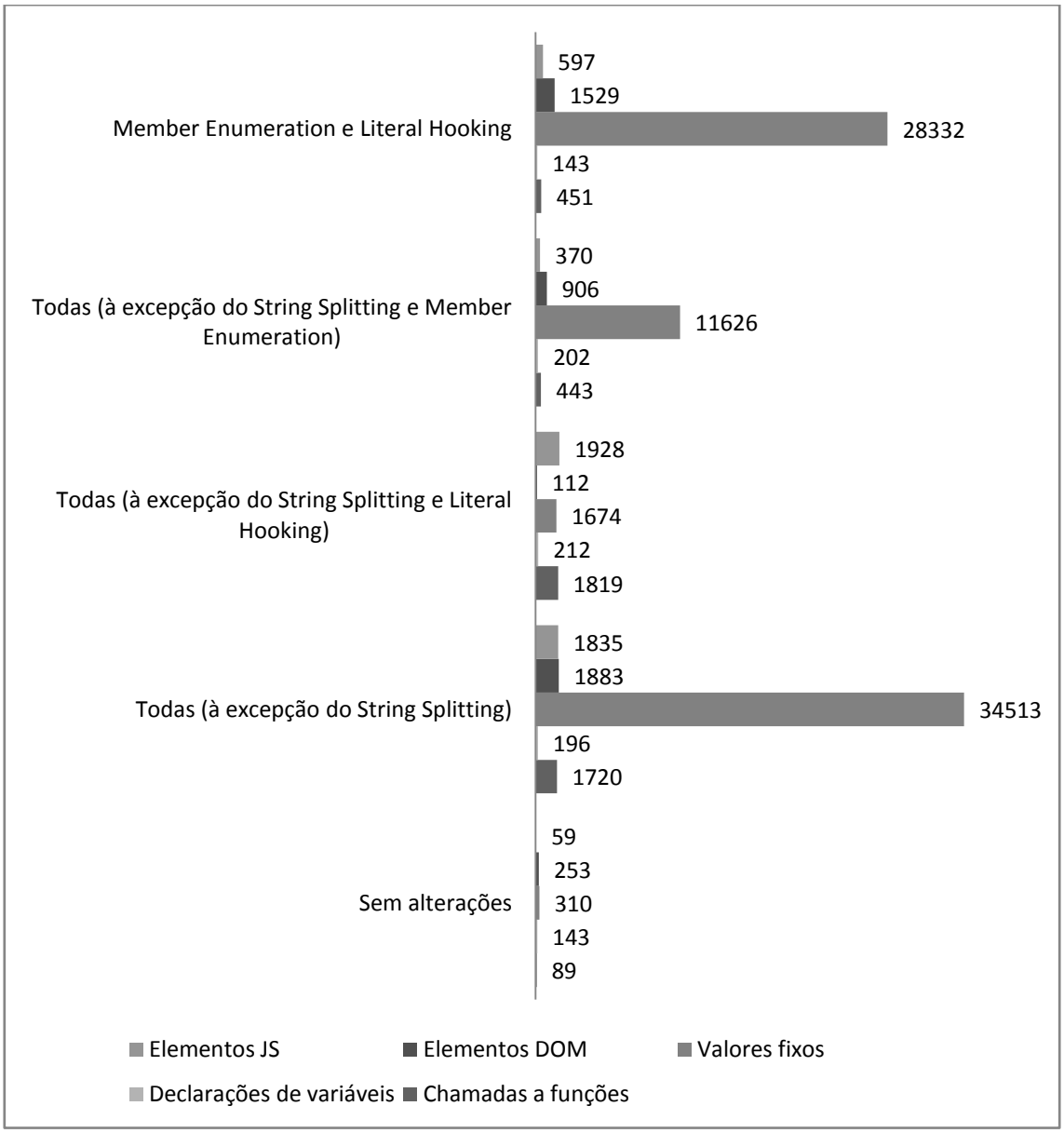


Figura 4.14 – Variação do número de elementos encontrados no código com a aplicação combinada das transformações.

4.2. Testes de Performance: *JSFromHell*

O ficheiro utilizado para a segunda ronda de testes é constituído por uma compilação de funções obtidas no repositório de *Javascript JSFromHell.com* [3]. É um ficheiro de teste computacionalmente mais exigente que o protótipo JIC. Das funções seleccionadas podemos encontrar, por exemplo, reordenação de vectores, desenho de gráficos, manipulação de cadeias de caracteres e cálculos matemáticos mais exigentes. Os mesmos testes efectuados ao protótipo do JIC são feitos novamente à excepção daqueles que testem transformações ou combinações de transformações que já não tenham interesse nesta segunda fase de testes (e.g. *string splitting*).

4.2.1. Navegadores

Os valores obtidos para os diferentes navegadores (ver Figura 4.15) denunciam que este ficheiro de teste é computacionalmente mais exigente que o primeiro (protótipo JIC).

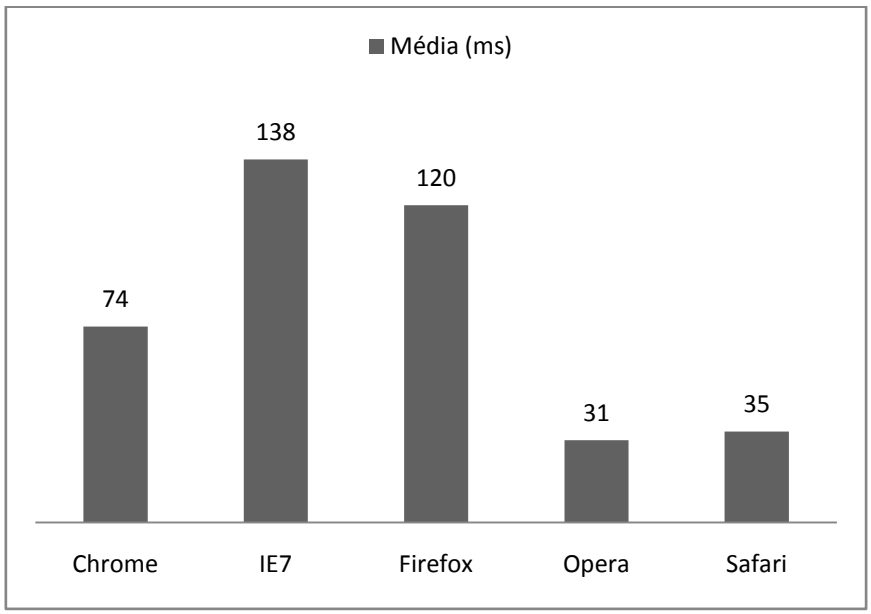


Figura 4.15 – Valores obtidos na execução do ficheiro de teste JSFromHell em diferentes navegadores.

Comparando os desempenhos dos navegadores pode verificar-se que o *Internet Explorer 7* continua a ser o navegador mais lento, não sendo seguido desta vez pelo *Opera*, mas sim, pelo *Mozilla Firefox*. Ao aplicar todas as transformações à excepção do *string splitting,* espera-se que com o aumento do custo o *Safari* e o *Chrome* mostrassem melhor desempenho, mas isso acaba por não acontecer. A relação entre os tempos de execução dos navegadores não é a mesma da obtida com a execução do protótipo JIC, deixando o *Mozilla Firefox* de ser o pior caso possível dos três (ver Figura 4.16).

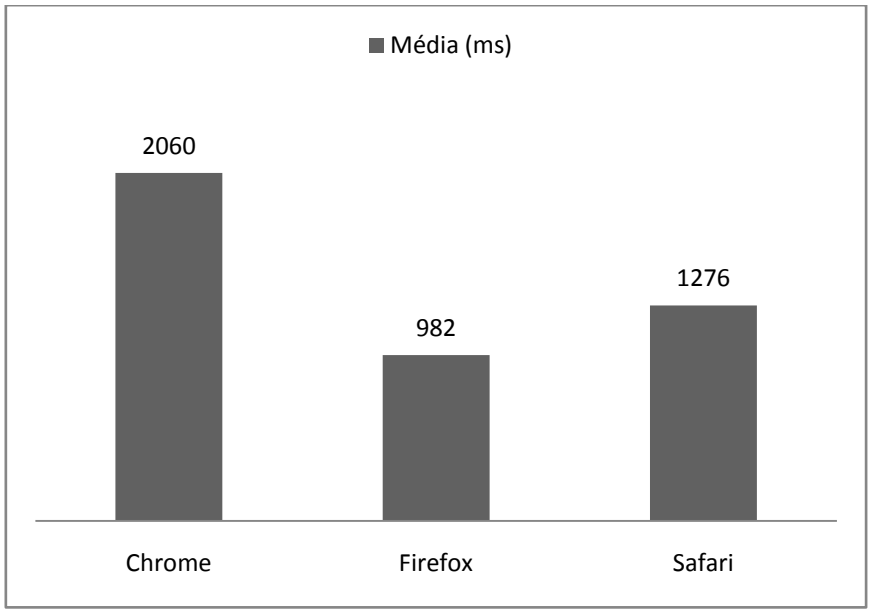


Figura 4.16 – Valores obtidos na execução do ficheiro de teste JSFromHell ofuscado em diferentes navegadores. Aplicação de todas as transformações implementadas à excepção *string splitting*.

Ao retirar-se a transformação *member enumeration,* o *Mozilla Firefox* continua a ser o melhor caso possível (ver Figura 4.17).

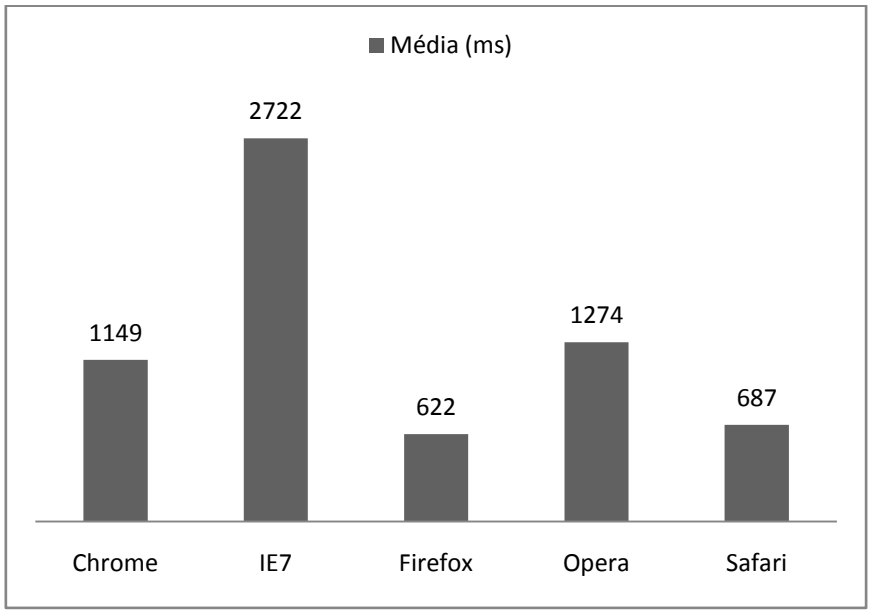


Figura 4.17 – Valores obtidos na execução do ficheiro de teste JSFromHell ofuscado em diferentes navegadores. Aplicação de todas as transformações implementadas à excepção *string splitting* e *member enumeration*.

Depois de testar outras combinações, verifica-se que com a aplicação de todas as transformações à excepção do *checksum* (ou qualquer outra combinação que não inclua o *checksum*) o *Mozilla Firefox* passa a ser novamente, o que tem o tempo de execução mais lento dos três (Figura 4.18). A utilização do *checksum* parece assim não afectar tão negativamente o custo de execução no *Mozilla Firefox* ao contrário do que acontece com outros navegadores. Já tinha sido anteriormente verificado que este navegador aplica um grande número de optimizações ao código que é

devolvido pela chamada do arguments.callee. Esta camada é introduzida pela transformação *checksum* e é a única explicação para que a execução neste navegador não tenha sido tão afectada como nos outros.

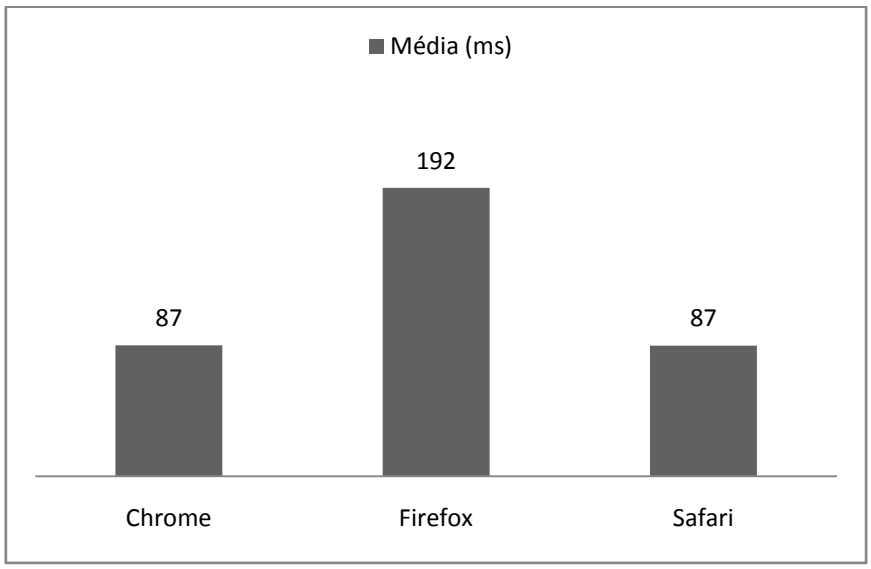


Figura 4.18 – Valores obtidos na execução do ficheiro de teste JSFromHell ofuscado em diferentes navegadores. Aplicação de todas as transformações implementadas à excepção *string splitting* e *checksum*.

Não sendo estritamente necessária a descoberta da razão que leva o *Mozilla Firefox* a não ser tão afectado por esta transformação - pois o que interessará é o estudo dos piores casos possíveis de agravamento do custo adicionado pela ofuscação - e tendo em conta que este já foi utilizado anteriormente como navegador de teste como também, só deixa de ser o caso com o tempo de execução mais lento aquando da utilização da transformação *checksum*, decidiu-se utilizá-lo novamente, juntamente com o *Internet Explorer 7* (o mais lento dos cinco), para que a comparação entre resultados dos diferentes ficheiros de teste, fosse feita com os mesmos navegadores.

4.2.2. Tempo de Execução no Navegador

Na Figura 4.19 podemos encontrar os resultados médios das execuções do código ofuscado com a utilização isolada das transformações. Este ficheiro de teste apresenta um tempo de execução muito superior ao protótipo JIC. O valor médio de *120ms* obtido no teste de execução do ficheiro sem alterações está acima até, de qualquer resultado da aplicação isolada de qualquer transformação sobre o primeiro ficheiro de teste. À semelhança dos testes efectuados ao protótipo JIC, verifica-se uma grande subida do tempo de execução aquando da aplicação do *checksum*. Já utilização do *member enumeration* não aumenta significativamente o tempo de execução neste caso. Isto deve-se ao facto de o número de chamadas *DOM* ser consideravelmente mais baixo neste ficheiro de teste.

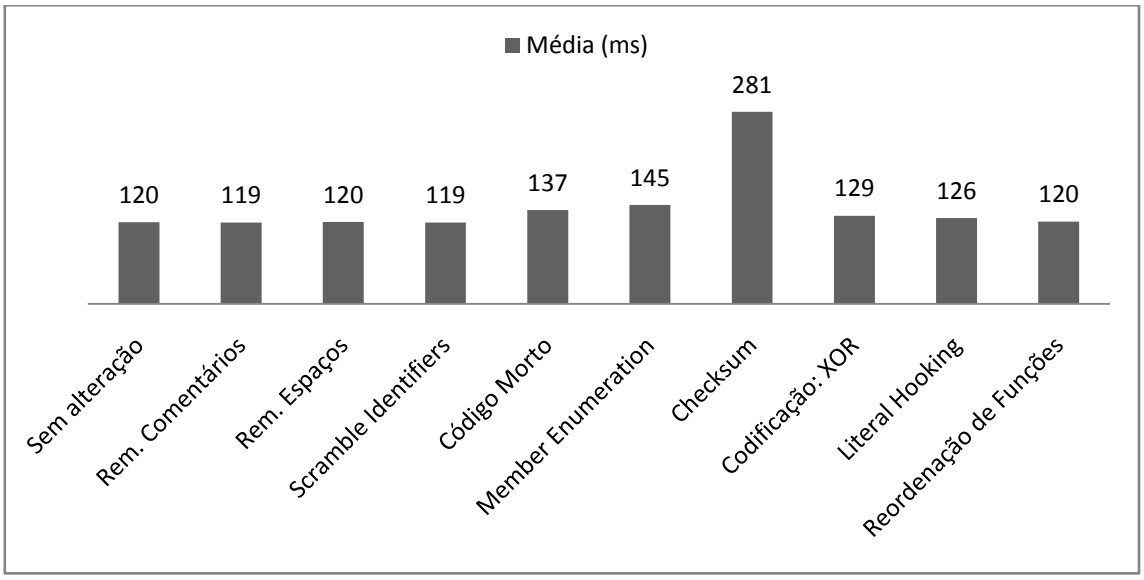


Figura 4.19 – Valores obtidos na execução do ficheiro de teste *JSFromHell* com a aplicação isolada de cada uma das transformações implementadas utilizando o *Mozzila Firefox*.

Os tempos de execução para as combinações de transformações mais interessantes são apresentados na Figura 4.20. Verifica-se que o valor obtido para a combinação de todas as transformações, à excepção do *string splitting* e *member enumeration* neste ficheiro, é superior ao valor obtido no mesmo teste efectuado ao protótipo JIC. Existem à primeira vista, duas possíveis razões. A primeira razão está relacionada com o facto do número de dependências entre funções neste ficheiro ser superior ao ficheiro de teste anterior, o que agrava o custo de execução do programa, pois serão efectuadas mais verificações de integridade. A segunda está relacionada com o facto do número de chamadas *DOM* ser pequeno neste ficheiro, o resultado da sua remoção não afecta tanto as verificações de integridade pela transformação *checksum* como acontecia no protótipo JIC.

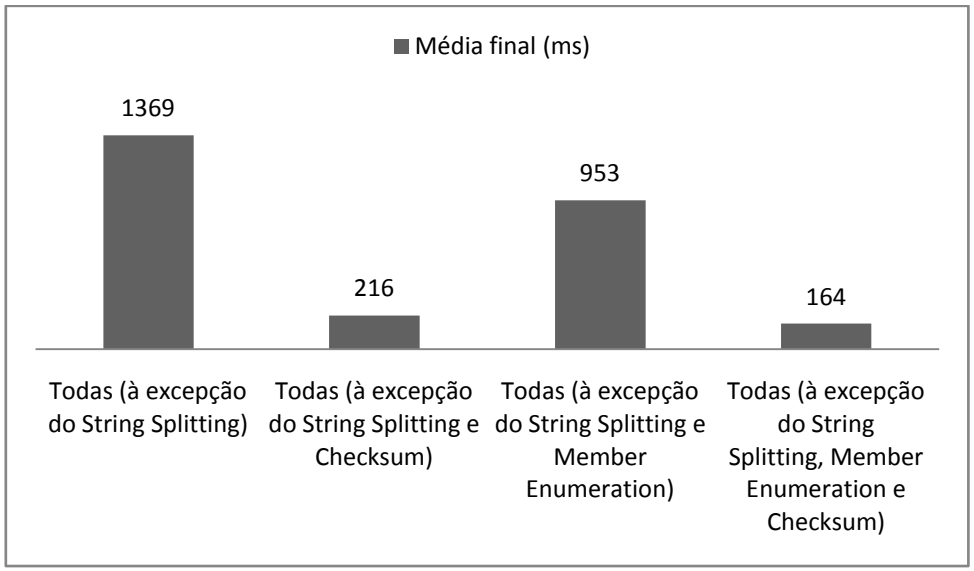


Figura 4.20 – Valores obtidos na execução do ficheiro de teste *JSFromHell* ofuscado com a aplicação de diferentes combinações de transformações.

Com as combinações que não utilizam o *member enumeration* foram obtidos para o *Internet Explorer 7* os resultados apresentados na Figura 4.21.

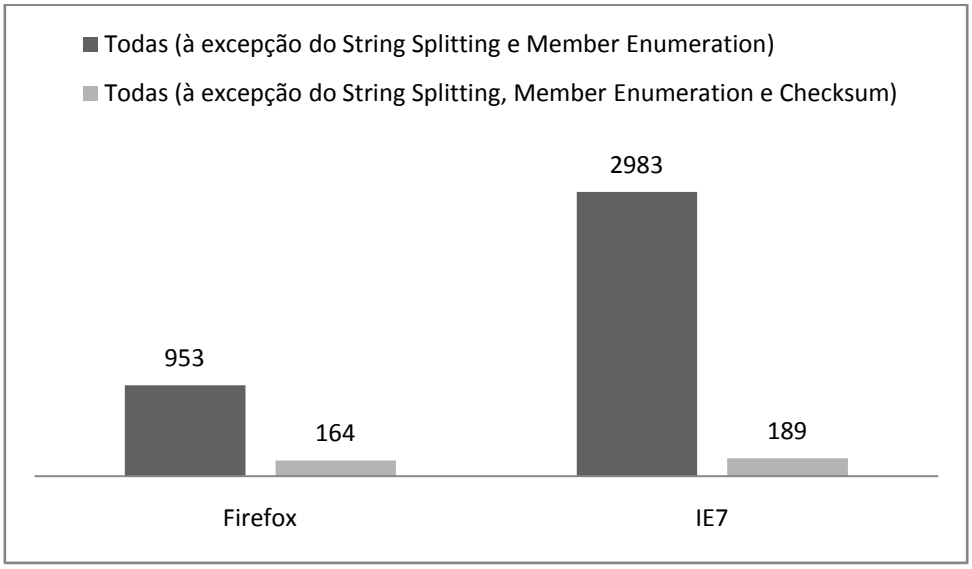


Figura 4.21 – Valores dos tempos obtidos na execução do ficheiro de teste JSFromHell ofuscado nos navegadores *Mozilla Firefox* e *Internet Explorer 7*.

Entre os navegadores apresentados na Figura 4.21, verifica-se que o *Internet Explorer 7* é novamente o mais lento, com uma execução 1,15 vezes mais lenta para a combinação mais pequena de transformações e 3,1 vezes mais lenta para a outra combinação. Como se pode verificar, os valores obtidos para a menor combinação de transformações (em ambos os navegadores) são perfeitamente praticáveis. Na outra combinação, já existe uma diferença de valores que ultrapassa os valores obtidos na mesma combinação aplicada ao protótipo JIC. O agravamento do tempo de execução é aproximadamente três vezes superior (entre *Mozilla Firefox* e *Internet Explorer 7*) quando no primeiro caso de teste era aproximadamente duas vezes superior. Esta discrepância entre os dois piores casos parece indicar novamente que o melhor desempenho do *Mozilla Firefox* aquando da utilização da chamada de arguments.callee é a justificação, e não um possível agravamento da interpretação do código pelo *Internet Explorer 7,* neste caso em particular. De qualquer forma, o valor obtido no pior caso ultrapassa a marca do segundo (1s) – custo que dificilmente seria aceite – descartando a utilização do ficheiro ofuscado com a combinação de transformações mais exigentes.

4.2.3. Tempo de Transformação na Ferramenta

A Figura 4.22 apresenta os resultados do teste ao tempo de transformação na ferramenta na aplicação isolada das técnicas de ofuscação. A Figura 4.23 apresenta os resultados do teste ao tempo de transformação na ferramenta na aplicação combinada das técnicas de ofuscação.

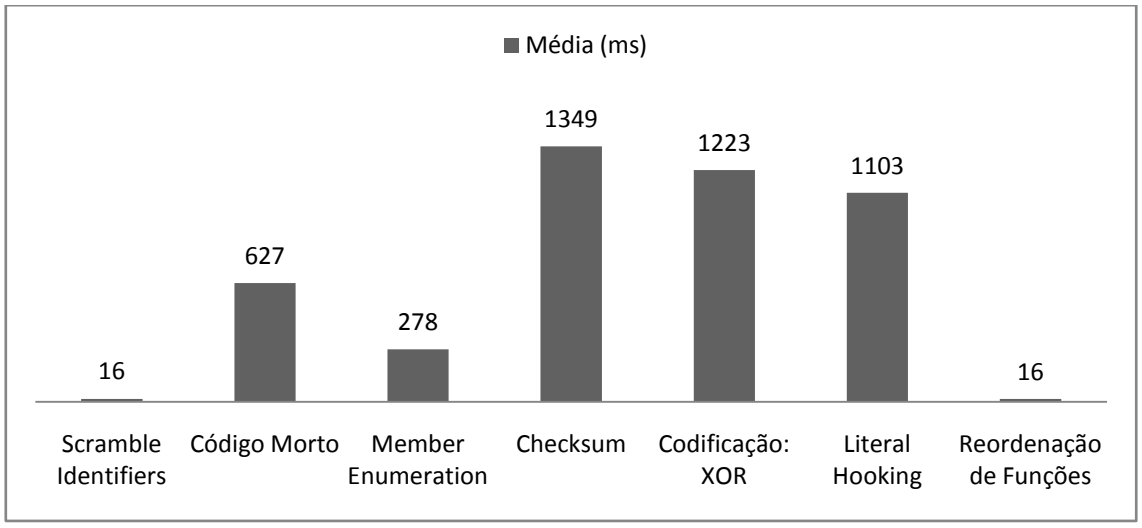
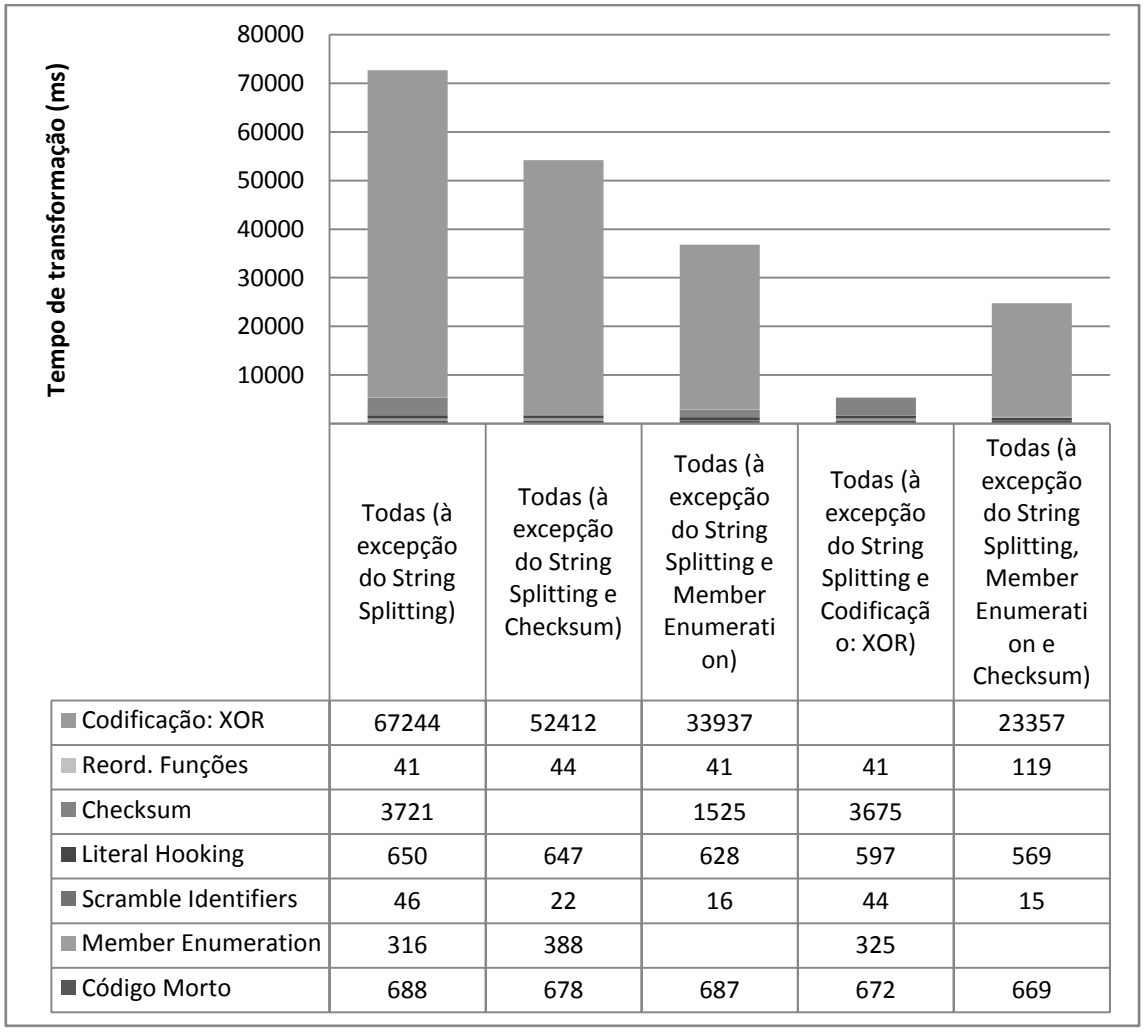


Figura 4.22 – Valores do tempo de transformação obtidos na aplicação isolada de transformações pela ferramenta.



	Todas (à excepção do String Splitting)	Todas (à excepção do String Splitting e Checksum)	Todas (à excepção do String Splitting e Member Enumeration)	Todas (à excepção do String Splitting e Codificação: XOR)	Todas (à excepção do String Splitting, Member Enumeration e Checksum)
Codificação: XOR	67244	52412	33937		23357
Reord. Funções	41	44	41	41	119
Checksum	3721		1525	3675	
Literal Hooking	650	647	628	597	569
Scramble Identifiers	46	22	16	44	15
Member Enumeration	316	388		325	
Código Morto	688	678	687	672	669

Figura 4.23 – Valores do tempo de transformação obtidos na aplicação combinada de transformações pela ferramenta.

Na Figura 4.23 verifica-se uma descida dos tempos de transformação na ferramenta com um padrão muito semelhante ao do ficheiro de teste anterior. Verifica-se novamente que, quanto menor o tamanho do código ofuscado – consequência da não utilização de transformações que introduzam muito código (e.g., *member enumeration*) a ser codificado, mais rápida é a criação do novo programa ofuscado.

De notar também que os valores, comparáveis com as mesma combinações do ficheiro de teste anterior, devolvem médias de tempo de transformação consideravelmente mais baixas, apesar de este ficheiro de teste ser maior que o anterior. A razão é novamente o número inferior de chamadas DOM que leva a que o código não cresça tão abruptamente, e a que não sejam disponibilizados tantos valores fixos para serem ofuscados pelo *literal hooking* (outra transformação que acrescenta muito código ao programa).

4.2.4. Tamanho, Nós da Árvore Sintáctica e Outros Elementos

Não existem observações relevantes que se possam fazer comparando os resultados obtidos neste ficheiro de teste com o anterior. Verifica-se novamente a diferença no número de chamadas DOM existente entre os ficheiros de teste, o que reforça as afirmações anteriores. As tabelas com os dados obtidos podem ser consultadas no Anexo A.

4.3. Testes de Compatibilidade com Navegadores

Ao longo do relatório foram apresentadas as transformações implementadas que quebram a funcionalidade do código aquando da utilização de diferentes navegadores (e respectivas soluções). Serve a presente secção para apresentar um resumo do estado de compatibilidade das transformações implementadas com os navegadores escolhidos para os testes (ver Tabela 4.2).

	Navegadores				
Transformações	Google Chrome	IE7	Mozilla Firefox	Opera	Safari
Checksum	Sim	Sim	Sim	Sim	Sim
Checksum (sem correcções)	Sim	Sim	Não	Sim	Sim
Member Enumeration	Sim	Não	Sim	Não	Sim
Restantes[13]	Sim	Sim	Sim	Sim	Sim

Tabela 4.2 – Compatibilidade das transformações implementadas com os diferentes navegadores seleccionados para teste.

As correcções efectuadas à transformação *checksum* visavam resolver o problema resultante da chamada do arguments.callee.toString() no *Mozilla Firefox* aplicar optimizações ao código antes de o devolver e por isso criarem-se *checksums* diferentes em tempo de transformação e em tempo de execução, quebrando a funcionalidade do código.

[13] Todas as implementadas (ver Tabela 3.3) à excepção das transformações *checksum* e *member enumeration*.

4.4. Conclusões

Neste capítulo foram apresentados os resultados experimentais obtidos nos testes de performance efectuados ao protótipo JIC – razão da implementação da ferramenta –, e a um segundo ficheiro (*JSFromHell*) que serviu para comparar resultados entre os diferentes ficheiros de teste e como exemplo de demonstração da viabilidade da utilização da ferramenta com outros ficheiros de código *Javascript*. Os testes de performance incluíram testes aos desempenhos dos navegadores escolhidos, tempos de execução do código nos navegadores, tempos de transformação na ferramenta, crescimento do tamanho do ficheiro, número de nós da árvore sintáctica e outros elementos no código.

Os resultados obtidos mostram que a combinação de transformações que não inclui o *member enumeration* e *string splitting* é a solução funcional (para o protótipo JIC) que oferece maior ofuscação e capacidade de anti-depuração, com um tempo de transformação baixo e sem ultrapassar a marca do segundo no tempo de execução (seja qual for o navegador utilizado). Quanto ao segundo ficheiro de teste, este já não poderia ser ofuscado com a mesma combinação se o limite do custo de tempo de execução fosse inferior a três segundos. Com os testes efectuados, verificou-se que a maior combinação possível de transformações – mantendo a funcionalidade em todos os navegadores – com o menor custo de tempo de execução, é a combinação que não inclui o *string splitting*, *member enumeration* e o *checksum*. Idealmente, aplicar-se-iam todas as transformações, mas para que isso seja possível para qualquer ficheiro a ofuscar, será necessário adicionar opções extra que possibilitem ao utilizador reduzir a potência ou abrangência de algumas transformações. O jogo entre potência e custo deverá ser feito, preferencialmente, alterando o comportamento das transformações, evitando assim, a escolha de umas transformações em detrimento de outras.

5. Conclusões

A ofuscação de código *Javascript* tenta contrariar a facilidade com que alguém consegue analisar e até reutilizar o código exposto no navegador. Neste âmbito, é apresentada neste relatório uma ferramenta que inclui um conjunto de técnicas de ofuscação e anti-depuração para automatizar o processo de ofuscação. Um processo automático de ofuscação é extremamente importante para que não sejam criados atrasos comprometedores no desenvolvimento de aplicações *Web*. Os resultados experimentais e os testes efectuados até ao momento revelam a robustez e eficiência da ferramenta desenvolvida.

Dificilmente se conseguiria aplicar uma transformação que dificultasse a compreensão do código e/ou resistência a inversão automática, sem aumentar o custo de execução e tamanho do código. Contudo, o aumento do custo pelas transformações implementadas e de um modo geral, quando aplicadas isoladamente, é praticamente indetectável. O problema da relação entre custo e eficácia de ofuscação surge, quando estas são usadas em conjunto, com diferentes ordens de aplicação ou abrangência. Idealmente, aplicar-se-iam todas as transformações, mas para que isso seja possível para qualquer programa a ofuscar, será necessário reduzir a potência ou abrangência de algumas transformações. O compromisso entre potência e custo deverá ser feito, preferencialmente, alterando o comportamento das transformações, evitando assim, a escolha de uma transformação em detrimento de outra. Outra condicionante é o diferente desempenho/comportamento na execução do mesmo programa ofuscado em diferentes navegadores. Com um limite de tempo de execução (custo) mais exigente, mais facilmente aparecerão navegadores que apresentarão valores acima desse limite, o que condiciona a utilização de algumas combinações e obriga a uma diminuição da potência de ofuscação. Existem também, algumas transformações que apresentam diferentes comportamentos quando utilizadas em diferentes navegadores, quebrando por vezes a funcionalidade do programa. Quando isto acontece é imperativo fazerem-se alterações aos algoritmos idealizados inicialmente, para possibilitar o seu funcionamento em todos os navegadores. Muitas vezes isso poderá ser bastante trabalhoso e reduzir consideravelmente o interesse por uma transformação.

A introdução de funções de *hash* na construção de predicados opacos é bastante promissora para dificultar a análise estática do programa. Contudo, é difícil a sua inserção no programa sem denunciar que uma transformação de ofuscação foi aplicada. A inserção de código morto utiliza condições de salto que contêm estes predicados opacos e por essa razão será interessante explorar a possibilidade de aplicar estas condições de salto a todo o programa, aumentando assim, os alvos a analisar e dificultando a inversão da transformação.

Foi constatado que transformações que codifiquem ou cifrem código (neste caso em particular *Javascript*) apresentam uma vulnerabilidade quando alvos de análise dinâmica. Com o crescente número de *plugins* para navegadores, apareceram alguns, que oferecem funcionalidades de depuração bastante úteis e que, por exemplo, devolvem o conteúdo de variáveis em tempo de execução. Por mais complexa que seja a rotina de descodificação ou de decifra, algures no programa, o código pronto a executar (decifrado) será devolvido e consequentemente apresentado de forma não intrusiva pelo *plugin*. Este facto conduz a que a necessidade de utilização de ferramentas de depuração para analisar o código, quando se sabe que este não é malicioso – o que é o caso do programa que se pretende ofuscar neste projecto –, seja cada vez menos necessária, perdendo-se assim, a capacidade de detecção de um ataque à ofuscação. Possíveis técnicas de anti-depuração idealizadas para detecção de alterações no código, detecção de alterações no tempo de execução ou detecção da utilização de ferramentas de depuração, são praticamente inúteis neste caso, pois deixa de existir a necessidade de compreender ou alterar o código ofuscado para se obter o código descodificado. Assim, numa tentativa de análise, examina-se o programa codificado como se de uma caixa negra se tratasse, não interessando o que ele faz, mas simplesmente o que ele devolve.

Como trabalho futuro pretende-se continuar a estudar a evolução das transformações de ofuscação, tanto na área de ofuscação de código malicioso (capacidade de não detecção), como na ofuscação como meio de protecção intelectual. Pretende-se também criar novas transformações e combinações de transformações mais potentes, sempre com as preocupações na relação entre custo e eficácia e com o objectivo de dificultar a análise estática e dinâmica ao código. Em termos de implementação, pretende-se aumentar o leque de transformações de ofuscação e anti-depuração e oferecer um maior número de opções que possibilitem o utilizador controlar a abrangência, localização e potência na aplicação das transformações. Para consolidar os resultados experimentais, aconselha-se a realização de testes à potência e resistência de inversão automática de ofuscação, completando assim a avaliação à qualidade de ofuscação. Para medir a dificuldade de compreensão e custo de inversão, poder-se-ia utilizar um grupo de pessoas com conhecimentos nas áreas necessárias à depuração e análise de código *Javascript* e medir o esforço necessário para inverter a ofuscação do código previamente ofuscado.

Bibliografia

[1]. **Visser, E., Mens, T. e Wallace, M.** Program Transformation. *Program-Transformation.Org.* [Online] 7 de Maio de 2004. [Citação: 20 de Abril de 2009.] http://www.program-transformation.org/Transform/ProgramTransformation.

[2]. **AuditMark Lda.** Home. *AuditMark.* [Online] http://www.auditmark.com/.

[3]. **Silva, J. e Rodrigues, C.** JSFromHell. *JSFromHell.com.* [Online] [Citação: 27 de Maio de 2009.] http://jsfromhell.com/.

[4]. **Collberg, C., Thomborson, C. e Low, D.** *A Taxonomy of Obfuscating Transformations.* Department of Computer Science, University of Auckland. New Zealand, 1997. pp. 1-36, Technical Report # 148.

[5]. *Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs.* **Collberg, C., Thomborson, C. e Low, D.** San Diego, California, United States, 1998. Proceedings of the 25th ACM SIGPLAN-SIGACT symposium on Principles of programming languages. pp. 184-196.

[6]. *A Qualitative Analysis of Java Obfuscation.* **Karnick, M., et al.** Dallas, TX, USA, 2006. Proceedings of 10th IASTED International Conference on Software Engineering and Applications. pp. 166-171.

[7]. *A complexity measure.* **McCabe, T.** 1976, IEEE Transactions on Software Engineering, Vol. 2(4), pp. 308-320.

[8]. *A complexity measure based on nesting level.* **Harrison, W. e Magel, K.** 1981, ACM SIGPLAN Notices, Vol. 16(3), pp. 63-74.

[9]. *Measurement of data structure complexity.* **Munson, J. e Kohshgoftaar, T.** 1993, Journal of Systems Software, Vol. 20, pp. 217-225.

[10]. **Hallstead, M.** *Elements of Software Science.* Elsevier North-Holland, 1997.

[11]. *A metrics suite for object oriented design.* **Chidamber, S. e Kemerer, C.** 1994, IEEE Transactions on Software Engineering, Vol. 20(6), pp. 476-493.

[12]. *Static analysis of executables to detect malicious patterns.* **Christodorescu, M. e Jha, S.** Washington, D.C., USA, 2003. In Proceedings of the 12th Usenix Security Symposium - Security'03. pp. 169-186.

[13]. *On the possibility of practically obfuscating programs towards a unified perspective of code protection.* **Beaucamps, P. e Filiol, E.** 2007, Journal in Computer Virology, Vol. 3, pp. 3-21.

[14]. *Program Fragmentation as a Metamorphic Software Protection.* **Birrer, B., et al.** Manchester, United Kingdom, 2007. Third International Symposium on Information Assurance and Security. pp. 369-374.

[15]. *Java Obfuscation with a Theoretical Basis for Building Secure Mobile Agents.* **Sakabe, Y., Soshi, M. e Miyaji, A.** Springer Berlin / Heidelberg, 2003, Lecture Notes in Computer Science, Vol. 2828, pp. 89-103.

[16]. *Obfuscation of Design Intent in Object-Oriented Applications.* **Sosonkin, M., Naumovich, G. e Memon, N.** Washington, D.C., USA, 2003. ACM Workshop On Digital Rights Management: Proceedings of the 3rd ACM workshop on Digital rights management, Software and Systems. pp. 142-153.

[17]. *Obfuscating Java: the most pain for the least gain.* **Batchelder, M. e Hendren, L.** Braga, Portugal, 2007. International Conference on Compiler Construction. Vol. 4420, pp. 96-110.

[18]. *Stealth Breakpoints.* **Vasudevan, A. e Yerraballi, R.** Tucson, Arizona, USA, 2005. Proceedings of the 21st Annual Computer Security Applications Conference. pp. 381-392.

[19]. **Microsoft.** IsDebuggerPresent Function. *Microsoft Developer Network.* [Online] 5 de Março de 2009. [Citação: 11 de Março de 2009.] http://msdn.microsoft.com/en-us/library/ms680345(VS.85).aspx.

[20]. **Microsoft.** NtQueryInformationProcess Function. *Microsoft Developer Network.* [Online] 5 de Fevereiro de 2009. [Citação: 11 de Março de 2009.] http://msdn.microsoft.com/en-us/library/ms684280(VS.85).aspx.

[21]. **Lawson, N.** Building a mesh versus a chain. *Root labs rdist.* [Online] 26 de Março de 2007. [Citação: 8 de Março de 2009.] http://rdist.root.org/2007/03/26/building-a-mesh-versus-a-chain.

[22]. **Lawson, N.** Mesh design pattern: hash-and-decrypt. *Root labs rdist.* [Online] 9 de Abril de 2007. [Citação: 8 de Março de 2009.] http://rdist.root.org/2007/04/09/mesh-design-pattern-hash-and-decrypt.

[23]. **Microsoft.** SetUnhandledExceptionFilter Function. *Microsoft Developer Network.* [Online] 5 de Março de 2009. [Citação: 12 de Março de 2009.] http://msdn.microsoft.com/en-us/library/ms680634.aspx.

[24]. **Falliere, N.** Windows Anti-Debug Reference. *Security Focus.* [Online] 12 de Setembro de 2007. [Citação: 12 de Março de 2009.] http://www.securityfocus.com/infocus/1893.

[25]. **Lawson, N.** Anti-debugging: using up a resource versus checking it. *Root labs rdist.* [Online] 21 de Maio de 2008. [Citação: 8 de Março de 2009.] http://rdist.root.org/2008/05/21/anti-debugging-using-up-a-resource-versus-checking-it.

[26]. *The Ghost In The Browser Analysis of Web-based Malware.* **Provos, N., et al.** Cambridge, Massachusetts, USA, 2007. Proceedings of the first conference on First Workshop on Hot Topics in Understanding Botnets. pp. 4-4.

[27]. *A Taxonomy of JavaScript Redirection Spam.* **Chellapilla, K. e Maykov, A.** Banff, Alberta, Canada, 2007. Proceedings of the 3rd international workshop on Adversarial information retrieval on the Web. Vol. 215, pp. 81-88.

[28]. **Flannagan, David.** *JavaScript: The Definitive Guide.* O'Reilly, 2006.

[29]. **Kolisar.** WhiteSpace: A Different Approach to JavaScript Obfuscation. *Hakim.* [Online] 2008. [Citação: 8 de Março de 2009.] http://www.hakim.ws/DEFCON16/Kolisar/defcon-16-kolisar.pdf.

[30]. **Wesemann, D.** Javascript decoding round-up. *Internet Storm Center.* [Online] 17 de Fevereiro de 2007. [Citação: 18 de Março de 2009.] http://isc.sans.org/diary.html?storyid=2268.

[31]. **Musciano, Chuck e Kennedy, Bill.** *HTML & XHTML: The Definitive Guide.* O'Reilly, 2002.

[32]. **Mozilla.** Rhino: JavaScript for Java. *mozilla.org.* [Online] 30 de Agosto de 2007. [Citação: 01 de Abril de 2009.] http://www.mozilla.org/rhino/.

[33]. **Mozilla.** The Mozilla Foundation. *mozilla.org.* [Online] 13 de Agosto de 2008. [Citação: 01 de Abril de 2009.] http://www.mozilla.org/foundation/.

[34]. **Mozilla.** SpiderMonkey (JavaScript-C) Engine. *mozilla.org.* [Online] 26 de Outubro de 2007. [Citação: 01 de Abril de 2009.] http://www.mozilla.org/js/spidermonkey/.

[35]. **Sun Microsystems, Inc.** Java SE 6. *Sun Developer Network.* [Online] 12 de Agosto de 2008. [Citação: 22 de Junho de 2009.] http://java.sun.com/javase/6/.

[36]. **The Flex Project.** flex: The Fast Lexical Analyzer. *flex: The Fast Lexical Analyzer.* [Online] [Citação: 1 de Março de 2009.] http://flex.sourceforge.net/.

[37]. **Free Software Foundation, Inc.** Bison - GNU parser generator. *GNU Operating System.* [Online] [Citação: 1 de Março de 2009.] http://www.gnu.org/software/bison/.

[38]. **Parr, Terence.** ANTLRv3. *ANTLR Parser Generator.* [Online] [Citação: 1 de Março de 2009.] http://www.antlr.org/.

[39]. **Sun Microsystems, Inc.** java.net The Source for Java Technology Collaboration. *javacc: Project home.* [Online] [Citação: 1 de Março de 2009.] https://javacc.dev.java.net/.

[40]. **Zdrnja, B.** Advanced JavaScript Obfuscation (or why signature scanning is a failure). *Internet Storm Center.* [Online] 7 de Abril de 2009. [Citação: 11 de Maio de 2009.] http://isc.sans.org/diary.html?storyid=6142&rss.

[41]. *Advanced Metamorphic Techniques in Computer Viruses.* **Beaucamps, P.** Venice, Italy, 2007. International Conference on Computer, Electrical, and Systems Science, and Engineering - CESSE'07. Vols. Inria-00338066, version 1.

[42]. *Watermarking, Tamper-Proofing, and Obfuscation - Tools for Software Protection.* **Collberg, C. e Thomborson, C.** 2002, IEEE Transactions on Software Engineering, Vol. 28(8), pp. 735-746.

[43]. **Zdrnja, B.** JavaScript traps for analysts. *Internet Storm Center.* [Online] 5 de Março de 2007. [Citação: 18 de Março de 2009.] http://isc2.sans.org/diary.html?storyid=2358.

[44]. *On the (Im) possibility of Obfuscating Programs.* **Barak, B., et al.** Santa Barbara, California, USA, 2001. Advances in Cryptology – CRYPTO'01. Vol. 2139, pp. 1-18.

[45]. **Lawson, N.** Mesh design pattern: error correction. *Root labs rdist.* [Online] 21 de Agosto de 2007. [Citação: 8 de Março de 2009.] http://rdist.root.org/2007/08/21/mesh-design-pattern-error-correction.

Anexo A

Os testes apresentados a seguir foram realizados em ambiente *Windows XP* 64bit, com um processador *Intel Core Quad* CPU *2.40 Ghz* e *2.00 Gb* de memória RAM.

A.1. Testes de Performance: Protótipo JIC

Navegadores

Navegadores	Tempo de execução (ms)										Média (ms)
Chrome	6	8	4	6	6	4	3	4	11	6	6
IE7	16	16	16	16	16	15	15	16	16	16	16
Firefox	7	7	7	7	7	8	7	7	8	8	7
Opera	16	15	15	16	16	16	15	16	16	16	16
Safari	10	5	5	5	5	5	5	5	5	4	5

Tabela A.1 – Valores obtidos na execução do protótipo JIC em diferentes navegadores.

Navegadores	Tempo de execução (ms)										Média (ms)
Chrome	319	304	306	305	310	304	315	306	307	308	308
IE7	812	781	781	781	782	781	782	766	782	781	783
Firefox	408	418	405	411	408	409	409	404	409	412	409
Opera	375	375	391	391	375	375	375	375	375	375	378
Safari	211	210	210	214	213	211	211	212	213	212	212

Tabela A.2 – Valores obtidos na execução do protótipo JIC ofuscado em diferentes navegadores. Aplicação de todas as transformações implementadas à excepção do *string splitting* e *member enumeration*.

Navegadores	Tempo de execução (ms)										Média (ms)
Chrome	1131	1133	1131	1141	1137	1138	1133	1140	1135	1134	1135
IE7	Não funciona										
Firefox	1426	1443	1495	1497	1482	1426	1509	1479	1483	1437	1468
Opera	Não funciona										
Safari	805	735	734	734	734	734	741	753	737	734	744

Tabela A.3 – Valores obtidos na execução do protótipo JIC ofuscado em diferentes navegadores. Aplicação de todas as transformações implementadas à excepção do *string splitting*.

Tempo de Execução no Navegador

Transformação	Tempo de execução (ms)										Média (ms)
Sem alteração	7	7	7	7	7	8	7	7	8	8	7
Rem. Comentários	8	7	8	7	7	7	7	7	7	8	7
Rem. Espaços	7	7	7	7	7	7	7	7	7	8	7
Scramble Identifiers	7	7	7	7	7	7	7	7	7	7	7
Código Morto	33	32	32	34	32	33	33	33	32	33	33
Member Enumeration	66	66	67	68	67	67	67	67	68	67	67
String Splitting	8	8	8	8	8	8	8	8	8	8	8
Checksum	40	41	40	40	41	41	40	40	41	41	41
Codificação: XOR	11	11	11	10	11	10	11	11	10	11	11
Literal Hooking	13	13	14	14	14	13	14	14	13	13	14
Reordenação de Funções	6	7	7	6	7	7	7	7	7	7	7

Tabela A.4 – Valores obtidos na execução do protótipo JIC com a aplicação isolada de cada uma das transformações implementadas.

Transformação	Tempo de execução (ms)										Média (ms)
Sem alteração	16	16	16	16	16	16	15	16	16	16	16
Rem. Comentários	16	16	16	16	16	16	16	16	16	16	16
Rem. Espaços	16	16	16	16	16	16	16	16	16	16	16
Scramble Identifiers	16	16	16	16	15	16	16	15	16	16	16
Código Morto	63	63	62	47	62	63	63	63	47	62	60
String Splitting	16	16	16	16	16	16	16	16	16	16	16
Checksum	141	141	141	125	125	140	125	125	140	125	133
Codificação: XOR	16	16	16	16	16	16	16	16	15	16	16
Literal Hooking	16	16	16	16	16	16	16	16	16	16	16
Reordenação de Funções	16	16	16	16	15	16	16	16	16	16	16

Tabela A.5 – Valores obtidos na execução do protótipo JIC com a aplicação isolada de cada uma das transformações implementadas IE7.

Criação nº	Tempo de execução (ms)										Média (ms)
1	1458	1436	1486	1440	1433	1433	1468	1462	1455	1441	1451
2	1424	1454	1440	1436	1436	1435	1425	1453	1426	1426	1436
3	1903	1906	1944	1942	1930	1883	1874	1980	2009	1946	1932
4	1456	1435	1412	1398	1449	1400	1414	1412	1466	1408	1425
5	1364	1420	1370	1370	1401	1356	1368	1358	1339	1368	1371
											Média final (ms)
											1523

Tabela A.6 – Valores obtidos na execução do protótipo JIC ofuscado com a aplicação de todas as transformações (à excepção do *string splitting*).

Criação nº	Tempo de execução (ms)										Média (ms)
1	792	788	797	799	788	796	795	795	794	794	794
2	927	931	929	929	930	930	926	928	928	925	928
3	1021	1012	1017	1020	1023	1025	1021	1022	1022	1017	1020
4	1031	1031	1027	1033	1030	1029	1031	1028	1031	1028	1030
5	897	898	895	897	901	893	896	899	899	898	897
											Média final (ms)
											934

Tabela A.7 – Valores obtidos na execução do protótipo JIC ofuscado com a aplicação de todas as transformações (à excepção do *string splitting* e codificação: XOR).

Criação nº	Tempo de execução (ms)										Média (ms)
1	339	338	348	345	343	348	336	341	344	337	342
2	421	418	419	418	417	427	416	419	415	419	419
3	319	318	500	315	318	316	324	321	317	321	337
4	352	351	348	373	353	352	356	352	354	365	356
5	237	245	238	243	246	238	247	237	254	240	243
											Média final (ms)
											339

Tabela A.8 – Valores obtidos na execução do protótipo JIC ofuscado com a aplicação de todas as transformações (à excepção do *string splitting* e *member enumeration*).

Criação nº	Tempo de execução (ms)										Média (ms)
1	625	625	625	625	625	625	625	640	640	640	630
2	765	750	766	765	766	765	766	765	766	766	764
3	578	563	562	563	563	562	563	562	562	562	564
4	625	641	641	640	625	625	640	625	625	625	631
5	437	437	422	422	437	422	438	422	437	422	430
											Média final (ms)
											604

Tabela A.9 – Valores obtidos na execução do protótipo JIC ofuscado com a aplicação de todas as transformações (à excepção do *string splitting* e *member enumeration*) *IE7*.

Criação nº	Tempo de execução (ms)										Média (ms)
1	200	198	198	198	199	197	197	200	199	197	198
2	225	224	225	225	224	223	224	227	225	227	225
3	144	145	145	145	144	145	144	143	143	144	144
4	206	206	203	203	205	204	205	204	204	205	205
5	155	155	156	154	155	156	157	155	157	158	156
											Média final (ms)
											186

Tabela A.10 – Valores obtidos na execução do protótipo JIC ofuscado com a aplicação de todas as transformações (à excepção do *string splitting* e *checksum*).

Criação nº	Tempo de execução (ms)										Média (ms)
1	82	82	83	83	82	82	82	82	83	83	82
2	123	123	123	122	123	124	124	123	123	124	123
3	67	69	68	68	69	68	67	67	68	68	68
4	52	52	51	52	52	52	52	51	52	52	52
5	58	58	57	57	58	57	56	57	57	58	57
											Média final (ms)
											77

Tabela A.11 – Valores obtidos na execução do protótipo JIC ofuscado com a aplicação de todas as transformações (à excepção do *string splitting*, *member enumeration* e *checksum*).

Criação nº	Tempo de execução (ms)										Média (ms)
1	110	109	110	110	109	109	110	125	109	125	113
2	172	156	172	172	156	156	156	156	172	157	163
3	94	94	94	94	94	94	94	94	94	93	94
4	63	62	63	62	63	78	63	63	78	79	67
5	78	78	79	78	62	63	62	78	63	63	70
											Média final (ms)
											101

Tabela A.12 – Valores obtidos na execução do protótipo JIC ofuscado com a aplicação de todas as transformações (à excepção do *string splitting, member enumeration* e *checksum*) *IE7*.

Criação nº	Tempo de execução (ms)										Média (ms)
1	7732	7766	7796	7778	7744	7719	7768	7810	7733	7719	7757

Tabela A.13 – Valores obtidos na execução do protótipo JIC ofuscado com a aplicação de todas as transformações (à excepção do *checksum* e codificação: XOR).

Tamanho dos Ficheiros

Transformação	Tamanho (kb)										Média (kb)
Sem alteração	14.2	(*)	(*)	(*)	(*)	(*)	(*)	(*)	(*)	(*)	14.2
Rem. Comentários	10.1	(*)	(*)	(*)	(*)	(*)	(*)	(*)	(*)	(*)	10.1
Rem. Espaços	13.7	(*)	(*)	(*)	(*)	(*)	(*)	(*)	(*)	(*)	13.7
Scramble Identifiers	16	16	16.4	16.1	16.2	16	16.2	16.4	15.9	16.3	16.2
Código Morto	15.1	30.1	17.2	20.2	18.1	17.6	19.7	19.7	18.2	17.7	19.4
Member Enumeration	40.9	40.8	40.6	40.5	40.4	40.7	40.9	40.2	40.9	40.8	40.7
String Splitting	39.5	39.3	39.7	39.3	39.5	39.5	39	39.2	39.6	39.6	39.4
Checksum	22.2	22.7	22.8	22.7	21.9	22.6	22.3	22.6	22.3	22.1	22.4
Codificação: XOR	15.6	15.6	15.7	15.6	15.6	15.6	15.6	15.7	15.6	15.7	15.6
Literal Hooking	52.3	53.7	52.5	52.3	52.7	52.8	53	51.5	51.8	51.6	52.4
Reord. de Funções	14.2	(*)	(*)	(*)	(*)	(*)	(*)	(*)	(*)	(*)	14.2

(*) Sem variação nas várias cópias criadas

Tabela A.14 – Valores dos tamanhos dos ficheiros resultantes da aplicação isolada das transformações.

	Tamanho (kb)					
Combinação de Transformações	**Criação nº1**	**Criação nº2**	**Criação nº3**	**Criação nº4**	**Criação nº5**	**Média (kb)**
Todas (sem String Splitting)	218	243	241	225	198	225
Todas (sem String Splitting e Checksum)	235	198	219	207	229	217.6
Todas (sem String Splitting e Member Enumeration)	68	87	63	64	74	71.2
Todas (sem String Splitting e Codificação: XOR)	191	246	227	238	218	224
Todas (sem String Splitting, Member Enumeration e Checksum)	78	81	67	63	63	70.4
Todas (sem Checksum e Codificação: XOR)	759	761	714	689	692	723

Tabela A.15 – Valores dos tamanhos dos ficheiros resultantes da combinação de transformações.

Tempo de Transformação na Ferramenta

Transformação	Tempo de transformação (ms)										Média (ms)
Rem. Comentários	(*)	(*)	(*)	(*)	(*)	(*)	(*)	(*)	(*)	(*)	
Rem. Espaços	(**)	(**)	(**)	(**)	(**)	(**)	(**)	(**)	(**)	(**)	
Scramble Identifiers	16	1	1	1	1	1	1	1	1	15	4
Código Morto	703	704	719	687	782	687	703	578	688	703	695
Member Enumeration	594	594	609	594	625	625	609	610	610	609	608
String Splitting	672	641	641	657	657	640	657	657	656	656	653
Checksum	1844	1828	1844	1813	1875	1656	1844	1875	1859	1828	1827
Codificação: XOR	1219	1250	1250	1250	1235	1250	1218	1250	1235	1235	1239
Literal Hooking	1688	1781	1703	1735	1719	1688	1766	1781	1719	1765	1735
Reord. de Funções	1	1	1	1	1	1	1	1	1	1	1

(*) Removido em tempo de análise sintáctica
(**) Não consome tempo de transformação

Tabela A.16 – Valores obtidos na aplicação isolada de transformações pela ferramenta.

	Tempo de transformação (ms)					
Transformação	**Criação nº1**	**Criação nº2**	**Criação nº3**	**Criação nº4**	**Criação nº5**	**Média (ms)**
Rem. Comentários	(*)	(*)	(*)	(*)	(*)	
Rem. Espaços	(*)	(*)	(*)	(*)	(*)	
Código Morto	672	672	735	735	687	700
Member Enumeration	1297	1281	1313	1250	1281	1284
Scramble Identifiers	31	47	47	63	47	47
Literal Hooking	1031	1140	1188	1172	1203	1147
Checksum	3532	3828	4484	4484	4078	4081
Reord. Funções	47	62	47	47	47	50
Codificação: XOR	202969	216265	202875	208641	213438	208838
String Splitting	-	-	-	-	-	-

(*) Irrelevante

Tabela A.17 – Valores do tempo de transformação obtidos na aplicação combinada de todas as transformações (à excepção do *string splitting*) pela ferramenta.

	Tempo de transformação (ms)					
Transformação	**Criação nº1**	**Criação nº2**	**Criação nº3**	**Criação nº4**	**Criação nº5**	**Média (ms)**
Rem. Comentários	(*)	(*)	(*)	(*)	(*)	
Rem. Espaços	(*)	(*)	(*)	(*)	(*)	
Código Morto	735	641	703	719	750	710
Member Enumeration	1265	1281	1219	1235	1359	1272
Scramble Identifiers	47	31	46	62	47	47
Literal Hooking	1188	1125	1156	1125	1172	1153
Checksum	6781	3890	3532	5750	3812	4753
Reord. Funções	47	47	47	47	47	47
Codificação: XOR	-	-	-	-	-	-
String Splitting	-	-	-	-	-	-

(*) Irrelevante

Tabela A.18 – Valores do tempo de transformação obtidos na aplicação combinada de todas as transformações (à excepção do *string splitting* e codificação: XOR) pela ferramenta.

	Tempo de transformação (ms)					
Transformação	**Criação nº1**	**Criação nº2**	**Criação nº3**	**Criação nº4**	**Criação nº5**	**Média (ms)**
Rem. Comentários	(*)	(*)	(*)	(*)	(*)	
Rem. Espaços	(*)	(*)	(*)	(*)	(*)	
Código Morto	687	719	750	750	687	719
Member Enumeration	-	-	-	-	-	-
Scramble Identifiers	16	15	15	16	15	15
Literal Hooking	1297	1328	1234	1297	1234	1278
Checksum	703	687	1000	765	547	740
Reord. Funções	47	47	47	47	47	47
Codificação: XOR	19656	37907	15531	19922	30391	24681
String Splitting	-	-	-	-	-	-

(*) Irrelevante

Tabela A.19 – Valores do tempo de transformação obtidos na aplicação combinada de todas as transformações (à excepção do *string splitting* e *member enumeration*) pela ferramenta.

	Tempo de transformação (ms)					
Transformação	**Criação nº1**	**Criação nº2**	**Criação nº3**	**Criação nº4**	**Criação nº5**	**Média (ms)**
Rem. Comentários	(*)	(*)	(*)	(*)	(*)	
Rem. Espaços	(*)	(*)	(*)	(*)	(*)	
Código Morto	688	687	625	562	734	659
Member Enumeration	1344	1265	1359	1422	1281	1334
Scramble Identifiers	31	32	47	31	47	38
Literal Hooking	1203	1203	1219	1219	1188	1206
Checksum	-	-	-	-	-	-
Reord. Funções	47	47	47	47	47	47
Codificação: XOR	211563	147891	183171	161328	200125	180816
String Splitting	-	-	-	-	-	-

(*) Irrelevante

Tabela A.20 – Valores do tempo de transformação obtidos na aplicação combinada de todas as transformações (à excepção do *string splitting* e *checksum*) pela ferramenta.

	Tempo de transformação (ms)					
Transformação	**Criação nº1**	**Criação nº2**	**Criação nº3**	**Criação nº4**	**Criação nº5**	**Média (ms)**
Rem. Comentários	(*)	(*)	(*)	(*)	(*)	
Rem. Espaços	(*)	(*)	(*)	(*)	(*)	
Código Morto	718	687	672	766	672	703
Member Enumeration	-	-	-	-	-	-
Scramble Identifiers	16	15	15	16	16	16
Literal Hooking	1406	1328	1360	1375	1359	1366
Checksum	-	-	-	-	-	-
Reord. Funções	47	47	47	47	47	47
Codificação: XOR	19359	28844	24562	23390	21016	23434
String Splitting	-	-	-	-	-	-

(*) Irrelevante

Tabela A.21 – Valores do tempo de transformação obtidos na aplicação combinada de todas as transformações (à excepção do *string splitting*, *member enumeration* e *checksum*) pela ferramenta.

Nós da Árvore Sintáctica

Transformação	Número de nós										Nº médio de nós
Sem alteração	4769	(*)	(*)	(*)	(*)	(*)	(*)	(*)	(*)	(*)	4769
Rem. Comentários	4769	(*)	(*)	(*)	(*)	(*)	(*)	(*)	(*)	(*)	4769
Rem. Espaços	4769	(*)	(*)	(*)	(*)	(*)	(*)	(*)	(*)	(*)	4769
Scramble Identifiers	4769	(*)	(*)	(*)	(*)	(*)	(*)	(*)	(*)	(*)	4769
Código Morto	6335	6489	6779	7353	7617	7428	7131	6498	7601	7119	7035
Member Enumeration	16971	(*)	(*)	(*)	(*)	(*)	(*)	(*)	(*)	(*)	16971
String Splitting	9593	9481	9609	9465	9457	9393	9537	9481	9401	9489	9491
Checksum	7712	(*)	(*)	(*)	(*)	(*)	(*)	(*)	(*)	(*)	7712
Codificação: XOR	779	(*)	(*)	(*)	(*)	(*)	(*)	(*)	(*)	(*)	779
Literal Hooking	36180	39404	39456	38884	38754	37714	38052	40730	37584	39326	38608
Reordenação de Funções	4769	(*)	(*)	(*)	(*)	(*)	(*)	(*)	(*)	(*)	4769

(*) Valores não variam

Tabela A.22 – Número de nós após a aplicação isolada das transformações.

	Número de nós					
Combinação de Transformações	**Criação nº1**	**Criação nº2**	**Criação nº3**	**Criação nº4**	**Criação nº5**	**Nº médio**
Todas (sem String Splitting)	779	779	779	779	779	779
Todas (sem String Splitting e Checksum)	779	779	779	779	779	779
Todas (sem String Splitting e Member Enumeration)	779	779	779	779	779	779
Todas (sem String Splitting e Codificação: XOR)	171903	186668	185635	180756	180867	181165
Todas (sem String Splitting, Member Enumeration e Checksum)	779	779	779	779	779	779

Tabela A.23 – Número de nós após a aplicação combinada de transformações.

Outros Elementos

Transformação	Declarações de funções	Chamadas a funções	Declarações de variáveis	Valores fixos	Elementos DOM	Elementos JS
Sem alterações	8	89	143	310	253	59
Remoção de comentários	(*)	(*)	(*)	(*)	(*)	(*)
Remoção de espaços	(*)	(*)	(*)	(*)	(*)	(*)
Scramble Identifiers	(*)	(*)	(*)	(*)	(*)	(*)
Código Morto	17	140	166	416	260	83
Member Enumeration	8	451	143	1215	68	597
String Splitting	8	90	1322	1489	253	60
Checksum	17	345	164	409	268	293
Codificação: XOR	10	108	159	518	261	78
Literal Hooking	8	89	143	7621	648	59
Reordenação de funções	(*)	(*)	(*)	(*)	(*)	(*)

(*) Valores semelhantes ao ficheiro não alterado

Tabela A.24 – Variação do número de elementos encontrados no código com a aplicação isolada das transformações.

Criação nº	Declarações de funções	Chamadas a funções	Declarações de variáveis	Valores fixos	Elementos DOM	Elementos JS
1	19	1740	198	35427	1976	1847
2	(*)	1752	205	35243	2023	1866
3	(*)	1671	183	33398	1848	1788
4	(*)	1683	192	33435	1734	1805
5	(*)	1752	202	35061	1834	1867
Média	19	1720	196	34513	1883	1835

(*) Não varia com diferentes cópias

Tabela A.25 – Variação do número de elementos encontrados no código com a aplicação combinada de todas as transformações (à excepção do *string splitting*).

Criação nº	Declarações de funções	Chamadas a funções	Declarações de variáveis	Valores fixos	Elementos DOM	Elementos JS
1	19	1828	204	1673	111	1931
2	(*)	1781	210	1649	110	1897
3	(*)	1758	206	1635	111	1877
4	(*)	1929	230	1764	123	2022
5	(*)	1801	210	1651	104	1914
Média	19	1819	212	1674	112	1928

(*) Não varia com diferentes cópias

Tabela A.26 – Variação do número de elementos encontrados no código com a aplicação combinada de todas as transformações (à excepção do *string splitting* e *literal hooking*).

Criação nº	Declarações de funções	Chamadas a funções	Declarações de variáveis	Valores fixos	Elementos DOM	Elementos JS
1	17	1695	174	34450	1885	1805
2	(*)	1715	179	34972	1988	1819
3	(*)	1836	201	37245	2015	1931
4	(*)	1678	173	34517	1893	1794
5	(*)	1678	173	34172	1836	1794
Média	17	1720	180	35071	1923	1829

(*) Não varia com diferentes cópias

Tabela A.27 – Variação do número de elementos encontrados no código com a aplicação combinada de todas as transformações (à excepção do *string splitting* e codificação: XOR).

Criação nº	Declarações de funções	Chamadas a funções	Declarações de variáveis	Valores fixos	Elementos DOM	Elementos JS
1	19	437	199	11551	931	367
2	(*)	427	185	10963	846	367
3	(*)	467	213	12180	941	384
4	(*)	452	208	11892	910	369
5	(*)	432	204	11544	904	365
Média	19	443	202	11626	906	370

(*) Não varia com diferentes cópias

Tabela A.28 – Variação do número de elementos encontrados no código com a aplicação combinada de todas as transformações (à excepção do *string splitting* e *member enumeration*).

Criação nº	Declarações de funções	Chamadas a funções	Declarações de variáveis	Valores fixos	Elementos DOM	Elementos JS
1	19	558	202	34143	1825	680
2	(*)	574	211	35513	1970	687
3	(*)	547	193	34963	1915	667
4	(*)	594	211	36224	1976	708
5	(*)	544	202	35009	1948	669
Média	19	563	204	35170	1927	682

(*) Não varia com diferentes cópias

Tabela A.29 – Variação do número de elementos encontrados no código com a aplicação combinada de todas as transformações (à excepção do *string splitting* e *checksum*).

Criação nº	Declarações de funções	Chamadas a funções	Declarações de variáveis	Valores fixos	Elementos DOM	Elementos JS
1	19	172	206	10939	906	108
2	(*)	165	200	10928	833	104
3	(*)	197	195	12086	914	125
4	(*)	207	210	12808	1000	130
5	(*)	209	202	12499	944	139
Média	19	190	203	11852	919	121

(*) Não varia com diferentes cópias

Tabela A.30 – Variação do número de elementos encontrados no código com a aplicação combinada de todas as transformações (à excepção do *string splitting*, *member enumeration* e *checksum*).

Criação nº	Declarações de funções	Chamadas a funções	Declarações de variáveis	Valores fixos	Elementos DOM	Elementos JS
1	17	1698	171	33992	1856	1813
2	(*)	1730	180	34219	1946	1847
3	(*)	1704	179	34116	1877	1819
4	(*)	1710	180	34771	1913	1830
5	(*)	1712	177	34594	1763	1814
Média	17	1711	177	34338	1871	1825

(*) Não varia com diferentes cópias

Tabela A.31 – Variação do número de elementos encontrados no código com a aplicação combinada de todas as transformações (à excepção do *checksum* e codificação: XOR).

Criação nº	Declarações de funções	Chamadas a funções	Declarações de variáveis	Valores fixos	Elementos DOM	Elementos JS
1	8	451	143	26971	1348	597
2	(*)	(*)	(*)	29660	1654	(*)
3	(*)	(*)	(*)	26965	1659	(*)
4	(*)	(*)	(*)	26857	1248	(*)
5	(*)	(*)	(*)	31209	1736	(*)
Média	8	451	143	28332	1529	597

(*) Não varia com diferentes cópias

Tabela A.32 – Variação do número de elementos encontrados no código com a aplicação combinada de todas as transformações (à excepção do *member enumeration* e *literal hooking*).

A.2. Testes de Performance: *JSFromHell*

Navegadores

Navegadores	Tempo de execução (ms)										Média (ms)
Chrome	68	69	71	69	73	113	69	68	70	72	74
IE7	141	140	140	141	140	141	141	125	125	141	138
Firefox	117	122	123	120	116	117	121	123	122	121	120
Opera	31	31	31	31	31	32	32	31	31	31	31
Safari	35	34	34	35	35	35	34	34	35	34	35

Tabela A.33 – Valores obtidos na execução do ficheiro de teste *JSFromHell* em diferentes navegadores.

Navegadores	Tempo de execução (ms)										Média (ms)
Chrome	1151	1141	1149	1149	1151	1149	1152	1149	1147	1147	1149
IE7	2179	2734	2719	2719	2735	2719	2719	2719	2719	2719	2722
Firefox	608	614	618	625	629	614	634	617	630	627	622
Opera	1281	1282	1281	1281	1266	1281	1250	1266	1266	1282	1274
Safari	689	682	686	686	685	691	694	684	688	684	687

Tabela A.34 – Valores obtidos na execução do ficheiro de teste *JSFromHell* ofuscado em diferentes navegadores. Aplicação de todas as transformações implementadas à excepção *string splitting* e *member enumeration*.

Navegadores	Tempo de execução (ms)										Média (ms)
Chrome	2067	2053	2082	2055	2052	2052	2067	2057	2067	2050	2060
IE7	Não funciona										
Firefox	987	973	973	983	969	982	999	992	975	986	982
Opera	Não funciona										
Safari	1272	1274	1272	1294	1279	1260	1270	1282	1274	1282	1276

Tabela A.35 – Valores obtidos na execução do ficheiro de teste *JSFromHell* ofuscado em diferentes navegadores. Aplicação de todas as transformações implementadas à excepção *string splitting*.

Navegadores	Tempo de execução (ms)										Média (ms)
Chrome	91	91	84	95	86	84	84	85	85	85	87
IE7	Não funciona										
Firefox	188	189	190	189	189	205	195	189	193	191	192
Opera	Não funciona										
Safari	87	89	89	87	89	87	86	85	85	85	87

Tabela A.36 – Valores obtidos na execução do ficheiro de teste *JSFromHell* ofuscado em diferentes navegadores. Aplicação de todas as transformações implementadas à excepção *string splitting e checksum*.

Tempo de Execução no Navegador

Transformação	Tempo de execução (ms)										Média (ms)
Sem alteração	122	117	121	120	119	117	118	121	121	119	120
Rem. Comentários	117	117	118	117	120	123	117	116	129	117	119
Rem. Espaços	116	121	118	117	118	123	117	119	128	121	120
Scramble Identifiers	117	118	118	119	119	123	122	115	119	121	119
Código Morto	132	134	142	135	138	137	141	141	134	136	137
Member Enumeration	141	150	143	143	142	147	152	145	143	142	145
Checksum	276	275	280	283	284	278	289	283	279	279	281
Codificação: XOR	128	130	130	128	129	129	132	127	129	127	129
Literal Hooking	123	123	126	126	124	125	133	126	123	127	126
Reordenação de Funções	119	120	120	124	124	117	115	118	120	125	120

Tabela A.37 – Valores obtidos na execução do ficheiro de teste *JSFromHell* com a aplicação isolada de cada uma das transformações implementadas.

Criação nº	Tempo de execução (ms)										Média (ms)
1	2103	2088	2126	2095	2094	2160	2173	2087	2225	2137	2129
2	1553	1558	1537	1530	1530	1520	1551	1548	1538	1516	1538
3	807	807	802	822	808	830	807	787	782	784	804
4	1167	1186	1178	1188	1183	1230	1259	1247	1254	1210	1210
5	1181	1173	1169	1173	1114	1152	1187	1146	1179	1185	1166

Média final (ms)
1369

Tabela A.38 – Valores obtidos na execução do ficheiro de teste *JSFromHell* ofuscado com a aplicação de todas as transformações (à excepção do *string splitting*).

Criação nº	Tempo de execução (ms)										Média (ms)
1	1839	1880	1897	1888	1859	1889	1903	1907	1867	1871	1880
2	774	784	769	765	766	797	780	770	767	779	775
3	830	830	820	829	846	833	834	849	826	827	832
4	538	556	536	538	545	542	537	532	528	547	540
5	724	732	730	739	745	742	751	755	728	738	738

Média final (ms)
953

Tabela A.39 – Valores obtidos na execução do ficheiro de teste *JSFromHell* ofuscado com a aplicação de todas as transformações (à excepção do *string splitting* e *member enumeration*).

Criação nº	Tempo de execução (ms)										Média (ms)
1	3359	3344	3344	3359	3312	3328	3344	3328	3312	3328	3336
2	2782	2766	2766	2755	2766	2797	2766	2766	2766	2781	2771
3	3312	3313	3312	3281	3313	3297	3312	3313	3328	3297	3308
4	2813	2844	2828	2844	2812	2829	2828	2828	2828	2828	2828
5	2656	2672	2672	2688	2688	2672	2672	2672	2672	2672	2674

Média final (ms)
2983

Tabela A.40 – Valores obtidos na execução do ficheiro de teste *JSFromHell* ofuscado com a aplicação de todas as transformações (à excepção do *string splitting* e *member enumeration*) *IE7*.

Criação nº	Tempo de execução (ms)										Média (ms)
1	179	200	199	205	207	203	204	201	207	200	201
2	248	252	250	246	252	252	249	247	250	248	249
3	194	193	195	198	193	196	199	194	191	190	194
4	223	226	223	230	222	229	225	228	229	222	226
5	205	208	210	212	206	207	209	212	211	206	209
											Média final (ms)
											216

Tabela A.41 – Valores obtidos na execução do ficheiro de teste *JSFromHell* ofuscado com a aplicação de todas as transformações (à excepção do *string splitting* e *checksum*).

Criação nº	Tempo de execução (ms)										Média (ms)
1	155	167	161	159	157	162	157	156	157	159	159
2	150	152	155	154	152	151	156	153	150	155	153
3	167	169	169	167	173	167	170	168	174	167	169
4	175	177	183	180	177	178	180	180	176	178	178
5	155	155	157	163	159	162	165	165	156	159	160
											Média final (ms)
											164

Tabela A.42 – Valores obtidos na execução do ficheiro de teste *JSFromHell* ofuscado com a aplicação de todas as transformações (à excepção do *string splitting*, *member enumeration* e *checksum*).

Criação nº	Tempo de execução (ms)										Média (ms)
1	171	172	172	172	188	172	187	172	172	172	175
2	171	172	172	172	172	172	171	172	172	172	172
3	203	203	188	204	203	203	203	203	203	204	202
4	218	218	203	203	219	219	219	203	203	203	211
5	187	188	187	172	188	187	188	187	172	187	184
											Média final (ms)
											189

Tabela A.43 – Valores obtidos na execução do ficheiro de teste *JSFromHell* ofuscado com a aplicação de todas as transformações (à excepção do *string splitting, member enumeration* e *checksum*) *IE7*.

Tamanho dos Ficheiros

Transformação	Tamanho (kb)										Média (kb)
Sem alteração	16.4	(*)	(*)	(*)	(*)	(*)	(*)	(*)	(*)	(*)	16.4
Rem. Comentários	10.3	(*)	(*)	(*)	(*)	(*)	(*)	(*)	(*)	(*)	10.3
Rem. Espaços	16.1	(*)	(*)	(*)	(*)	(*)	(*)	(*)	(*)	(*)	16.1
Scramble Identifiers	21.4	21.2	20.9	21.5	21.1	21.4	21.6	20.9	21.7	21.2	21.3
Código Morto	21.7	20.9	18.8	19.4	19.4	20.5	18.9	19.4	19.4	18.9	19.7
Member Enumeration	29.1	28.4	28.4	29.1	29	29.3	29.1	29	29.5	29.2	29.0
Checksum	28.6	28.8	28.5	28.1	28.8	28.3	28.9	28.6	28.6	28.7	28.6
Codificação: XOR	18.1	18.1	18.2	18.1	18.1	18.1	18.2	18.2	18.2	18.2	18.2
Literal Hooking	52.9	65.3	53.5	52	55.2	60.3	55.7	53.5	70.9	68	58.7
Reordenação de Funções	16.4	(*)	(*)	(*)	(*)	(*)	(*)	(*)	(*)	(*)	16.4

(*) Sem variação nas várias cópias criadas

Tabela A.44 – Valores dos tamanhos dos ficheiros resultantes da aplicação isolada das transformações.

	Tamanho (kb)					
Combinação de Transformações	**Criação nº1**	**Criação nº2**	**Criação nº3**	**Criação nº4**	**Criação nº5**	**Média (kb)**
Todas (sem String Splitting)	133	133	132	134	141	134.6
Todas (sem String Splitting e Checksum)	115	132	102	107	133	117.8
Todas (sem String Splitting e Member Enumeration)	92	92	88	87	78	87.4
Todas (sem String Splitting, Member Enumeration e Checksum)	77	82	72	85	78	78.8

Tabela A.45 – Valores dos tamanhos dos ficheiros resultantes da combinação de transformações.

Tempo de Transformação na Ferramenta

Transformação	**Tempo de transformação (ms)**										**Média (ms)**
Rem. Comentários	(*)	(*)	(*)	(*)	(*)	(*)	(*)	(*)	(*)	(*)	
Rem. Espaços	(**)	(**)	(**)	(**)	(**)	(**)	(**)	(**)	(**)	(**)	
Scramble Identifiers	16	16	16	16	16	16	16	16	16	16	16
Código Morto	688	687	703	687	672	63	688	703	688	688	627
Member Enumeration	266	281	265	281	281	265	281	281	297	281	278
Checksum	1344	1359	1375	1344	1360	1328	1343	1329	1360	1344	1349
Codificação: XOR	1203	1204	1218	1234	1234	1219	1219	1234	1234	1235	1223
Literal Hooking	1125	1047	1141	1172	1125	1062	1110	1015	1094	1141	1103
Reordenação de Funções	16	15	16	16	16	16	16	16	16	16	16

(*) Removido em tempo de análise sintáctica
(**) Não consome tempo de transformação

Tabela A.46 – Valores obtidos na aplicação isolada de transformações pela ferramenta.

	Tempo de transformação (ms)					
Transformação	**Criação nº1**	**Criação nº2**	**Criação nº3**	**Criação nº4**	**Criação nº5**	**Média (ms)**
Rem. Comentários	(*)	(*)	(*)	(*)	(*)	
Rem. Espaços	(*)	(*)	(*)	(*)	(*)	
Código Morto	703	688	672	672	704	688
Member Enumeration	328	344	360	204	344	316
Scramble Identifiers	172	15	15	15	15	46
Literal Hooking	672	641	672	625	641	650
Checksum	2875	4015	4984	2437	4296	3721
Reord. Funções	31	31	32	47	63	41
Codificação: XOR	68094	71797	80313	46656	69360	67244
String Splitting	-	-	-	-	-	-

(*) Irrelevante

Tabela A.47 – Valores do tempo de transformação obtidos na aplicação combinada de todas as transformações (à excepção do *string splitting*) pela ferramenta.

	Tempo de transformação (ms)					
Transformação	**Criação nº1**	**Criação nº2**	**Criação nº3**	**Criação nº4**	**Criação nº5**	**Média (ms)**
Rem. Comentários	(*)	(*)	(*)	(*)	(*)	
Rem. Espaços	(*)	(*)	(*)	(*)	(*)	
Código Morto	688	688	688	703	594	672
Member Enumeration	313	297	328	281	406	325
Scramble Identifiers	31	15	16	141	16	44
Literal Hooking	640	594	578	609	562	597
Checksum	3422	3531	3390	4500	3531	3675
Reord. Funções	47	47	32	31	47	41
Codificação: XOR	-	-	-	-	-	-
String Splitting	-	-	-	-	-	-

(*) Irrelevante

Tabela A.48 – Valores do tempo de transformação obtidos na aplicação combinada de todas as transformações (à excepção do *string splitting* e codificação: XOR) pela ferramenta.

	Tempo de transformação (ms)					
Transformação	**Criação nº1**	**Criação nº2**	**Criação nº3**	**Criação nº4**	**Criação nº5**	**Média (ms)**
Rem. Comentários	(*)	(*)	(*)	(*)	(*)	
Rem. Espaços	(*)	(*)	(*)	(*)	(*)	
Código Morto	750	687	656	672	672	687
Member Enumeration	-	-	-	-	-	-
Scramble Identifiers	16	15	16	16	16	16
Literal Hooking	625	703	625	610	578	628
Checksum	1766	1312	1562	1718	1265	1525
Reord. Funções	47	32	47	47	32	41
Codificação: XOR	38938	35312	35875	33140	26422	33937
String Splitting	-	-	-	-	-	-

(*) Irrelevante

Tabela A.49 – Valores do tempo de transformação obtidos na aplicação combinada de todas as transformações (à excepção do *string splitting* e *member enumeration*) pela ferramenta.

	Tempo de transformação (ms)					
Transformação	**Criação nº1**	**Criação nº2**	**Criação nº3**	**Criação nº4**	**Criação nº5**	**Média (ms)**
Rem. Comentários	(*)	(*)	(*)	(*)	(*)	
Rem. Espaços	(*)	(*)	(*)	(*)	(*)	
Código Morto	672	719	656	671	671	678
Member Enumeration	313	391	313	312	612	388
Scramble Identifiers	15	16	16	31	31	22
Literal Hooking	641	672	578	579	765	647
Checksum	-	-	-	-	-	-
Reord. Funções	31	62	47	46	32	44
Codificação: XOR	51094	58000	40000	44515	68453	52412
String Splitting	-	-	-	-	-	-

(*) Irrelevante

Tabela A.50 – Valores do tempo de transformação obtidos na aplicação combinada de todas as transformações (à excepção do *string splitting* e *checksum*) pela ferramenta.

	Tempo de transformação (ms)					
Transformação	**Criação nº1**	**Criação nº2**	**Criação nº3**	**Criação nº4**	**Criação nº5**	**Média (ms)**
Rem. Comentários	(*)	(*)	(*)	(*)	(*)	
Rem. Espaços	(*)	(*)	(*)	(*)	(*)	
Código Morto	656	672	671	672	672	669
Member Enumeration	-	-	-	-	-	-
Scramble Identifiers	15	16	15	16	15	15
Literal Hooking	547	625	547	593	532	569
Checksum	-	-	-	-	-	-
Reord. Funções	125	125	94	141	109	119
Codificação: XOR	23969	27596	19438	27219	18563	23357
String Splitting	-	-	-	-	-	-

(*) Irrelevante

Tabela A.51 – Valores do tempo de transformação obtidos na aplicação combinada de todas as transformações (à excepção do *string splitting*, *member enumeration* e *checksum*) pela ferramenta.

Nós da Árvore Sintáctica

Transformação	**Número de nós**										**Nº médio de nós**
Sem alteração	6159	(*)	(*)	(*)	(*)	(*)	(*)	(*)	(*)	(*)	6159
Rem. Comentários	6159	(*)	(*)	(*)	(*)	(*)	(*)	(*)	(*)	(*)	6159
Rem. Espaços	6159	(*)	(*)	(*)	(*)	(*)	(*)	(*)	(*)	(*)	6159
Scramble Identifiers	6159	(*)	(*)	(*)	(*)	(*)	(*)	(*)	(*)	(*)	6159
Código Morto	7963	8168	8314	7879	8085	8252	8687	8315	8457	8956	8308
Member Enumeration	9984	(*)	(*)	(*)	(*)	(*)	(*)	(*)	(*)	(*)	9984
Checksum	9614	(*)	(*)	(*)	(*)	(*)	(*)	(*)	(*)	(*)	9614
Codificação: XOR	779	(*)	(*)	(*)	(*)	(*)	(*)	(*)	(*)	(*)	779
Literal Hooking	53257	40127	60407	43741	45431	42155	44963	52347	42415	51047	47589
Reordenação de Funções	6159	(*)	(*)	(*)	(*)	(*)	(*)	(*)	(*)	(*)	6159

(*) Valores não variam

Tabela A.52 – Número de nós após a aplicação isolada das transformações.

	Número de nós					
Combinação de Transformações	**Criação nº1**	**Criação nº2**	**Criação nº3**	**Criação nº4**	**Criação nº5**	**Nº médio**
Todas (sem String Splitting)	779	779	779	779	779	779
Todas (sem String Splitting e Checksum)	779	779	779	779	779	779
Todas (sem String Splitting e Member Enumeration)	779	779	779	779	779	779
Todas (sem String Splitting e Codificação: XOR)	114011	105953	117743	86611	111047	107073
Todas (sem String Splitting, Member Enumeration e Checksum)	779	779	779	779	779	779

Tabela A.53 – Número de nós após a aplicação combinada de transformações.

Outros Elementos

Transformação	Declarações de funções	Chamadas a funções	Declarações de variáveis	Valores fixos	Elementos DOM	Elementos JS
Sem alterações	6	182	148	378	139	154
Remoção de comentários	(*)	(*)	(*)	(*)	(*)	(*)
Remoção de espaços	(*)	(*)	(*)	(*)	(*)	(*)
Scramble Identifiers	(*)	(*)	(*)	(*)	(*)	(*)
Código Morto	15	228	171	469	146	184
Member Enumeration	6	282	148	628	89	304
Checksum	15	510	169	493	162	453
Codificação: XOR	8	201	164	586	147	173
Literal Hooking	6	182	140	8158	576	154
Reordenação de funções	(*)	(*)	(*)	(*)	(*)	(*)

(*) Valores semelhantes ao ficheiro não alterado

Tabela A.54 – Variação do número de elementos encontrados no código com a aplicação isolada das transformações.

Criação nº	Declarações de funções	Chamadas a funções	Declarações de variáveis	Valores fixos	Elementos DOM	Elementos JS
1	17	905	191	20411	1085	897
2	(*)	1018	210	19129	1220	997
3	(*)	969	200	19696	1147	952
4	(*)	928	195	21378	1454	917
5	(*)	969	200	20030	946	952
Média	17	958	199	20129	1170	943

(*) Não varia com diferentes cópias

Tabela A.55 – Variação do número de elementos encontrados no código com a aplicação combinada de todas as transformações (à excepção do *string splitting*).

Criação nº	Declarações de funções	Chamadas a funções	Declarações de variáveis	Valores fixos	Elementos DOM	Elementos JS
1	17	992	204	1046	134	972
2	(*)	992	202	1045	135	974
3	(*)	902	187	964	120	894
4	(*)	928	195	990	124	917
5	(*)	925	189	977	120	916
Média	17	948	195	1004	127	935

(*) Não varia com diferentes cópias

Tabela A.56 – Variação do número de elementos encontrados no código com a aplicação combinada de todas as transformações (à excepção do *string splitting* e *literal hooking*).

Criação nº	Declarações de funções	Chamadas a funções	Declarações de variáveis	Valores fixos	Elementos DOM	Elementos JS
1	17	586	204	13268	986	508
2	(*)	565	203	13700	820	499
3	(*)	541	191	13192	751	481
4	(*)	550	195	13095	825	485
5	(*)	596	199	10760	751	529
Média	17	568	198	12803	827	500

(*) Não varia com diferentes cópias

Tabela A.57 – Variação do número de elementos encontrados no código com a aplicação combinada de todas as transformações (à excepção do *string splitting* e *member enumeration*).

Criação nº	Declarações de funções	Chamadas a funções	Declarações de variáveis	Valores fixos	Elementos DOM	Elementos JS
1	17	239	191	11840	743	193
2	(*)	239	191	12802	731	193
3	(*)	262	189	10426	756	217
4	(*)	246	197	13136	844	197
5	(*)	253	194	10075	563	197
Média	17	248	192	11656	727	199

(*) Não varia com diferentes cópias

Tabela A.58 – Variação do número de elementos encontrados no código com a aplicação combinada de todas as transformações (à excepção do *string splitting*, *member enumeration* e *checksum*).

Criação nº	Declarações de funções	Chamadas a funções	Declarações de variáveis	Valores fixos	Elementos DOM	Elementos JS
1	6	282	148	15574	870	304
2	(*)	(*)	(*)	17515	899	(*)
3	(*)	(*)	(*)	14461	875	(*)
4	(*)	(*)	(*)	16719	958	(*)
5	(*)	(*)	(*)	16115	982	(*)
Média	6	282	148	16077	917	304

(*) Não varia com diferentes cópias

Tabela A.59 – Variação do número de elementos encontrados no código com a aplicação combinada de todas as transformações (à excepção do *member enumeration* e *literal hooking*).

Índice Remissivo

Q

R

S

T

U

V

X