FACULDADE DE ENGENHARIA DA UNIVERSIDADE DO PORTO

Solução de criptografia de caixa branca para aplicações JavaScript

Luís Filipe Ferreira Araújo

Mestrado Integrado em Engenharia Informática e Computação

Orientador: José Manuel de Magalhães Cruz (Professor Doutor)

Junho de 2016

Solução de criptografia de caixa branca para aplicações JavaScript

Luís Filipe Ferreira Araújo

Mestrado Integrado em Engenharia Informática e Computação

Aprovado em provas públicas pelo Júri:

Presidente: Nome do Presidente (Título)

Vogal Externo: Nome do Arguente (Título)

Orientador: José Manuel de Magalhães Cruz (Professor Doutor)

---

27 de Junho de 2016

Resumo

Atualmente, a linguagem de programação JavaScript é uma das mais utilizadas. Por todo o mundo são produzidas e distribuídas inúmeras aplicações JavaScript. Assim sendo, estas têm de ser protegidas contra roubos que podem violar a propriedade intelectual da aplicação e contra adulterações maliciosas que podem modificar o seu comportamento.

Para proteger estas aplicações, um dos possíveis caminhos é usar encriptação. No entanto, a encriptação, mesmo forte, tem um ponto de falha: a chave criptográfica. Caso esta seja comprometida, a aplicação ficará facilmente acessível para potenciais atacantes.

Os algoritmos de criptografia de caixa branca tentam proteger a chave para que um atacante não lhe tenha acesso e, consequentemente, a aplicação tenha um maior nível de segurança.

Neste documento é apresentada uma primeira solução de criptografia de caixa branca para aplicações JavaScript. Esta solução é totalmente adaptada ao contexto em que se insere, sendo tão importante o acesso ao código fonte da aplicação como à chave criptográfica. Além disso, é uma solução que combate as principais formas de ataque proporcionadas pela própria linguagem.

A solução de criptografia de caixa branca será resistente a modificações por parte de um atacante. Caso este modifique o código fonte, a aplicação deixará de funcionar.

A solução proposta aumenta o nível de segurança apesar de ter algumas limitações. Um atacante se tiver um bom nível de conhecimento sobre JavaScript e criatividade, poderá conseguir com dificuldade ultrapassar os mecanismos de defesa que são aplicados nesta solução e, eventualmente atingir os seus objetivos finais.

Nos testes de desempenho, verificou-se que, com esta solução, uma função ficou 11 a 14 vezes mais lenta. Não existe nenhuma forma de estimar como será o desempenho de uma função transformada em relação à original, dependendo apenas da estrutura da função.

Abstract

Nowadays, the JavaScript programming language is one of the most used. JavaScript applications are produced and distributed in all over the world. Therefore, this kind of applications has to be protected against thefts which can break the intellectual property and against malicious tampering.

In order to protect JavaScript applications, one of the possible ways is to use encryption. However, the encryption, even strong, has a point of failure: the encryption key. If this key is compromised, the application will become easily accessible for potential attackers.

The white-box cryptography solutions try to protect the encryption key so that an attacker cannot have access to it and, thereafter, the application will have a higher security level.

In this document it is presented a first white-box cryptography solution for JavaScript applications. This solution fits well in its environment and the access to the source code of the application has the same relevance as the access to the cryptographic key. Besides that, it's a solution that fights the majors threats to the JavaScript programming language.

The white-box cryptography solution will have an anti-tampering mechanism. If an attacker modifies the application, it will stop working.

The proposed solution increases the security level despite some limitations. An attacker with a good level of JavaScript and creativity could overcome with difficulty the defense mechanisms that are applied in this solution and eventually achieves its goals.

In the performance tests with this solution, it was found that a function was 11 to 14 times slower. There is no way to estimate how a transformed function will perform over the original function. It depends on the function structure.

Agradecimentos

Ao meu orientador, José Manuel de Magalhães Cruz, por toda a orientação, disponibilidade e conselhos que me deu durante toda a realização desta tese.

À Jscrambler S.A., com um maior enfâse ao Pedro Fortuna, por todo o conhecimento e ajuda que se mostraram capazes de me transmitir, mesmo em alturas mais complicadas.

Aos meus amigos, por todos os momentos de descontração e brincadeira que indiretamente fizeram com que esta tese se tenha tornado realidade.

Aos meus pais, por toda a ajuda que sempre me prestaram quando eu precisei, empurrando sempre para que esta tese seguisse em frente.

À Helena, por todos os momentos muito bem passados e por se ter tornado um génio do JavaScript e da criptografia de caixa branca ao rever esta tese.

Luís Filipe Ferreira Araújo

Conteúdo

Lista de Figuras

Lista de Tabelas

Abreviaturas e Símbolos

AES	Advanced Encryption Standard
AST	Árvore sintática abstrata
CBC	Cipher Block Chaining
C-WBC AES	Modelo de criptografia proposto por Chow et al. por adaptação do algoritmo AES
C-WBC DES	Modelo de criptografia proposto por Chow et al. por adaptação do algoritmo DES
DES	Data Encryption Standard
DRM	Digital Rights Management (Gestão de direitos digitais)
ECB	Electronic CodeBook
GF(2)	Corpo de Galois de 2 elementos
HTML5	HyperText Markup Language 5
IIFE	Expressão de função imediatamente invocada
OFB	Output FeedBack
PCBC	Propagating Cipher Block Chaining
PRNG	Pseudo Random Number Generator
RAM	Random Access Memory
SHA-256	Secure Hash Algorithm de 256 bits
SMS4	Algoritmo criptográfico de chave simétrica SMS4
WBC	Criptografia de caixa branca
WBC AES	Criptografia de caixa branca por adaptação do algoritmo AES
XOR	Ou exclusivo

Capítulo 1

Introdução

1.1 Enquadramento

Este tema enquadra-se na gestão de direitos digitais de *software* JavaScript. A gestão de direitos digitais tenta proteger os desenvolvedores, tentando evitar cópia, adulteração ou uso abusivo de *software*. Neste caso, as aplicações a proteger serão em JavaScript, de forma a impedir a adulteração e roubo por parte de qualquer potencial atacante.

1.2 Objectivos e contribuições

O objectivo desta dissertação é explorar soluções de criptografia de "caixa branca" (white-box cryptography) de forma a alcançar uma solução que proteja aplicações JavaScript em ambientes não confiáveis. Esta solução não deve permitir acesso e modificação ao código fonte das aplicações JavaScript a proteger.

A solução apresentada tem de estar adaptada às caraterísticas da linguagem de programação JavaScript de forma a oferecer um bom nível de segurança. Para isso, é necessário ter em conta as principais fraquezas e funcionalidades que esta linguagem de programação tem.

Como resultado final deste documento, obtém-se técnica e *software* que proteja um código fonte JavaScript, permitindo a sua utilização em ambiente Web, mas impedindo o seu conhecimento e utilização abusiva.

1.3 Caraterização do problema

Uma aplicação em JavaScript pode ser executada num local que não é confiável, onde um atacante poderá tentar adulterar ou roubar essa mesma aplicação. Este tipo de ambiente é

conhecido por ser "não confiável" e nele um atacante poderá usar ferramentas como depuradores de *software*, acedendo assim a informação privilegiada.

Para fazer face a este tipo de ambientes, em criptografia, criou-se o conceito de criptografia de caixa branca. As soluções de criptografia de caixa branca deverão resistir à extração de qualquer tipo de informação considerada sigilosa, como por exemplo de chaves criptográficas.

Desde 2003, surgiram várias soluções de criptografia de caixa branca que consistiam na adaptação de algoritmos criptográficos de chave simétrica. As alterações tinham como objetivo dificultar o acesso à chave criptográfica dentro do mecanismo de cifra e fazer com que cada utilização do algoritmo criptográfico de caixa branca não fosse de resultado único, tendo dados de saída diferentes para os mesmos dados de entrada e chave.

Existem algoritmos de criptografia de caixa branca para diferentes algoritmos como AES ou DES. As primeiras soluções foram propostas por *Chow et al.* [1], [2] para cada um destes algoritmos.

Atualmente não é conhecida nenhuma solução de criptografia de caixa branca que seja adaptada às caraterísticas da linguagem de programação JavaScript e que proteja aplicações em JavaScript, sendo que esta necessidade prevalece.

JavaScript é uma linguagem de programação altamente flexível, sendo interpretada e fracamente tipada. Tem especificidades que fazem com que uma solução de criptografia de caixa branca possa ter muitos detalhes e funcionalidades, enriquecendo a solução. Alguns exemplos destes detalhes são expressões de função imediatamente invocadas, *prototype poisoning* e *closures*.

Neste documento, é apresentada uma primeira solução de criptografia de caixa branca para aplicações JavaScript, que impede o acesso e adulteração indevido por parte de um atacante com controlo total do dispositivo que contém a aplicação e o código fonte em JavaScript. Nesta solução estarão presentes mecanismos de proteção de *software*, que fazem com que o *software* deixe de funcionar caso seja objeto de algum tipo de modificação.

1.4 Organização do documento

Este documento encontra-se dividido em 7 capítulos e os 4 primeiros capítulos são de introdução e estado da arte: (1) Introdução, (2) Ofuscação, (3) White-box Cryptography, (4) JavaScript, (5) Descrição da solução, (6) Implementação e Validação, (7) Conclusões e Trabalho Futuro.

Capítulo 2

Ofuscação

2.1 Introdução

A proteção de *software* tem vindo a aumentar a sua importância nas mais diversas indústrias e, assim sendo, as ferramentas e técnicas nesta área têm vindo a aumentar de uso e complexidade.

A ofuscação é uma das técnicas mais utilizadas e também uma das mais controversas a nível científico. Existem artigos científicos que provam a impossibilidade de criar um ofuscador genérico para determinado tipo de problema [3].

A ofuscação é necessária, visto que, utilizando desassembladores e descompiladores de aplicações, consegue-se fazer engenharia reversa a *software*.

"A ideia básica é a Alice correr a sua aplicação num ofuscador, um programa que transforma a aplicação numa que é funcionalmente idêntica face à original mas que é muito mais difícil para o Bob entender. É nossa convicção que a ofuscação é uma técnica viável para proteger segredos comerciais de *software*, que ainda não recebeu a atenção que merece." [4].

Passados muitos anos desde que [4] foi publicado, a ofuscação cresceu a nível comercial e científico, tanto que existem as mais variadas utilizações desta técnica, tal como a criptografia de caixa branca.

Neste documento será dada maior importância à ofuscação de dados, visto que é esse o tipo de ofuscação utilizado em soluções de criptografia de caixa branca, como poderemos ver com mais detalhe mais à frente, no capítulo 3.

2.2 Métricas de ofuscação

A ofuscação é determinada por várias métricas: (1) potência (ou grau de obscuridade), (2) resistência, (3) custo [4] e (4) discrição [5], [6], que serão explicadas de forma sucinta de seguida. Neste conjunto de métricas, terá de ser encontrado um valor em que a ofuscação seja suficientemente forte para que o atacante não consiga fazer engenharia reversa em tempo útil, mesmo usando uma ferramenta de desofuscação, mas em que o desempenho da aplicação não seja prejudicado de forma significativa.

2.2.1 Potência

A potência da ofuscação do *software* define-se pela dificuldade em entender e aplicar engenharia reversa a um programa. Esta métrica deverá ser sempre maximizada, para que alguém mal-intencionado tenha de perder muito mais tempo a entender a funcionalidade do *software*.

2.2.2 Resistência

A resistência de cada transformação durante a ofuscação deverá ser também maximizada. Esta caraterística é definida pela dificuldade em criar uma ferramenta automática de desofuscação e pela execução dessa ferramenta ser demasiado demorada.

2.2.3 Custo

O custo é definido pelo aumento de tempo de execução ou aumento da quantidade de memória utilizada durante a execução de um programa ofuscado em relação ao original. Este valor será sempre minimizado para que o utilizador não note nenhuma diferença de desempenho derivada da ofuscação.

2.2.4 Discrição

A discrição é a capacidade do resultado de uma transformação passar despercebido a um humano. “Por exemplo, se uma transformação introduzir novo código que difere muito do programa original, será fácil localizá-la para aplicar engenharia reversa.” [5]

Esta métrica também deve ser maximizada.

2.3 Transformações

Para se conseguir ofuscar *software* é necessário recorrer a um vasto conjunto de transformações. Transformações essas que fazem com que o código fonte original fique incompreensível. Estas transformações são categorizadas em (1) ofuscação de dados, (2) ofuscação de controlo, (3) ofuscação da apresentação e (4) transformações preventivas [4]. Devido às soluções de criptografia de caixa branca ofuscarem apenas dados, só abordaremos esse subtópico das transformações de ofuscação.

2.3.1 Ofuscação de dados

De forma a ofuscar os dados, poderão ser utilizados vários métodos como divisão de variáveis, conversão de dados estáticos em procedimentais, mudança do tempo de vida de variáveis, promoção de escalares em objectos e modificação da codificação. É neste último ponto que se baseia grande parte das soluções de criptografia de caixa branca, como poderemos ver no capítulo seguinte. A modificação da codificação de variáveis permite que o conteúdo que o programador necessita esteja guardado, sem que o atacante perceba facilmente qual é esse conteúdo. Na Figura 1 podemos ver uma transformação de ofuscação de dados na variável i, utilizando o método de modificação da codificação.

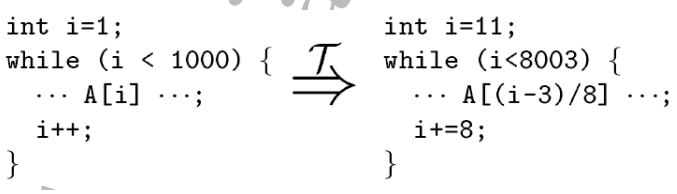

Figura 1 - Transformação de ofuscação de dados na variável i, utilizando o método de modificação da codificação. i' = 8*i + 3, considerando i a variável antes da transformação e i' a variável depois da transformação. [4]

2.4 Conclusão

A ofuscação protege *software* através do conceito de "segurança através da obscuridade" [6], fazendo código e dados obscuros para todos os atacantes, dificultando-lhes ao máximo o processo de engenharia reversa.

A ofuscação divide-se em vários tipos de transformações do código fonte original em código ofuscado, sendo que neste trabalho o mais importante a realçar, é a ofuscação de dados

através da modificação da codificação que lhes é aplicada. Desta forma, é possível ofuscar chaves utilizadas em algoritmos criptográficos, como se irá analisar mais à frente.

Existem algumas métricas que classificam a ofuscação quanto ao seu poder e desempenho, sendo estas utilizadas por ofuscadores automáticos.

Apesar de a ofuscação ser uma solução comum na proteção de propriedade intelectual, apresenta também algumas lacunas. “Com tempo, esforço e determinação, um programador competente conseguirá estar sempre apto a fazer engenharia reversa a uma aplicação.” [4].

Capítulo 3

White-box Cryptography

3.1 Introdução

Tradicionalmente, em criptografia são usados modelos onde se supõe que o atacante só conseguirá ter acesso a dados de entrada e saída, sendo denominados modelos de "caixa preta". Nestas condições, estes modelos são adequados à proteção dos sistemas. Mas, neste trabalho, pretende-se proteger aplicações de ambientes não confiáveis e a criptografia de caixa preta não é adequada. Nestes ambientes, o atacante consegue observar o código, tanto em execução como no estado armazenado, a chave criptográfica e todas as variáveis intermédias de execução desse código. É neste tipo de ambientes que aparece a criptografia de "caixa branca" (*White-Box Cryptography*, WBC). Neste modelo criptográfico, as soluções são idealizadas tendo em conta que o ambiente é totalmente observável e modificável pelo atacante, através de, por exemplo, depuradores de código. Assim sendo, é preciso com estas condicionantes minimizar a probabilidade de a segurança ser comprometida, dificultando ao máximo ao atacante aceder a informação de forma não autorizada ou interferir com o normal funcionamento da aplicação.

Uma implementação de criptografia de caixa branca é feita com recurso exclusivo a *software*, não se podendo guardar chaves em *hardware*, visto que o próprio modelo assume que um atacante consegue aceder e observar tudo. Os valores guardados em *hardware* só servem para dificultar esse acesso e observação, não fazendo sentido em modelos criptográficos de caixa branca. Uma implementação é tipicamente efetuada por adaptação de algoritmos criptográficos de chave simétrica, como Advanced Encryption Standard (AES) [7], Data Encryption Standard (DES) [1] e SMS4 [8]. Uma das razões possíveis para a não utilização de algoritmos criptográficos de chave assimétrica é o desempenho, visto que estes têm fases de cifragem e decifragem muito mais morosas que os algoritmos criptográficos de chave simétrica.

As implementações de criptografia de caixa branca baseiam-se na ofuscação de algoritmos criptográficos de chave simétrica, acrescentando dados aleatórios em vários passos da cifragem

e decifragem. Desta forma tentam impossibilitar o acesso à chave utilizada, ao texto decifrado, assim como a valores intermédios de execução do algoritmo criptográfico. Estes dados aleatórios são adicionados durante a fase de compilação, e não em execução, para que o atacante não os consiga distinguir dos dados reais. Os dados aleatórios são adicionados recorrendo a métodos de ofuscação, como codificação de variáveis e, desta forma, o atacante não consegue extrair informação relativa, por exemplo, à chave criptográfica utilizada.

Em todos os algoritmos de chave simétrica, existe uma operação de cálculo envolvendo o estado atual da cifra e a chave criptográfica. Esta operação, numa implementação de criptografia de caixa branca, é efetuada com recurso a tabelas de consulta, estando a chave embutida nestas tabelas e sendo o dado de entrada destas o estado do algoritmo criptográfico. Para ser possível utilizar tabelas de consulta com os recursos computacionais atuais, estas são partidas em pequenas tabelas, visto que uma tabela de consulta completa necessitaria de uma exagerada quantidade de memória [7].

A chave utilizada na cifragem e decifragem pode ser fixa ou dinâmica, ou seja, existem implementações que especificam a obrigatoriedade de manter sempre a mesma chave enquanto outras permitem a atualização da chave a qualquer momento.

3.2 Criptografia simétrica

A criptografia de chave simétrica é geralmente composta por vários tipos de operações: (1) operações de substituição, aplicadas através de caixas-S, (2) operações de permutação, aplicadas através de caixas-P e (3) operações de transformação, aplicadas através de caixas-T. Estes tipos de operações são aplicados nos mais variados algoritmos criptográficos de chave simétrica conhecidos e são estes 3 tipos que fazem parte do algoritmo criptográfico de chave simétrica que é atualmente o padrão, o AES, tendo este sido precedido pelo DES.

Estes algoritmos têm várias rondas semelhantes e aplicadas de forma consecutiva, sendo que cada ronda é composta por várias operações dentro dos 3 tipos de operações anteriormente mencionados.

Como o AES é atualmente o padrão deste tipo de algoritmos será este o algoritmo que será aprofundado de seguida.

AES é um algoritmo de chave simétrica, usualmente com 10 rondas, sendo que o número de rondas depende do tamanho da chave a utilizar, podendo ser 10, 12 ou 14; quanto mais rondas tiver, maior terá de ser o tamanho da chave. Em cada uma das rondas, é utilizada uma sub-chave diferente que é gerada a partir da chave completa, através de métodos que não serão aqui abordados. Na Figura 2 temos uma visão geral sobre este algoritmo.

Os dados de entrada e saída deste algoritmo têm tamanho igual e fixo de 16 bytes.

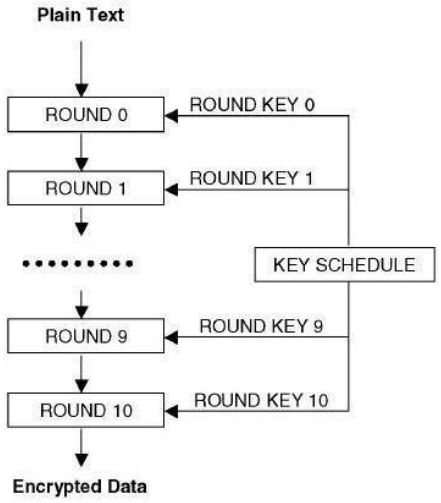

Figura 2 - Visão geral do algoritmo AES, com 10 rondas. Em cada ronda é utilizada uma sub-chave diferente. [19]

Em cada uma das rondas, são utilizadas 4 operações (Figura 3): (1) *Subbytes*, onde são substituídos bytes por outros numa transformação não linear; (2) *ShiftRows*, onde cada *byte* de uma linha do estado é deslocado exatamente o número de vezes correspondente ao número da linha a que pertence; (3) *MixColumns*, onde os *bytes* de cada coluna são combinados numa transformação linear invertível; (4) *AddRoundKey*, onde a sub-chave é combinada com o estado da cifra através de um XOR.

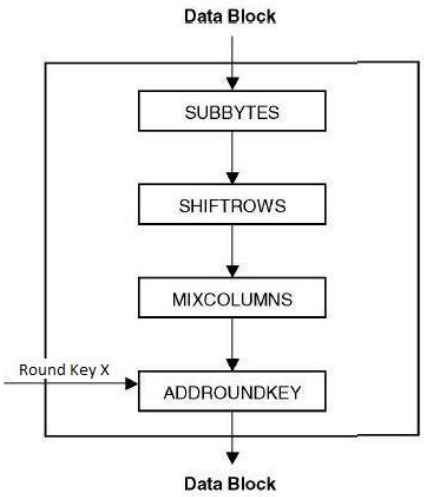

Figura 3 - Exemplo de uma ronda do algoritmo AES, utilizando a sua sub-chave respetiva. [19]

3.3 Modificações de criptografia de caixa branca

Em primeiro lugar, será necessário recordar que as tabelas de consulta são utilizadas nas operações que envolvem a chave criptográfica, não sendo esta um dado de entrada em nenhuma operação.

Como em ambientes não confiáveis o atacante tem total acesso aos recursos do dispositivo eletrónico, facilmente conseguiria aceder às tabelas de consulta e retirar informação relativa à chave. Para impedirmos isso de acontecer, são utilizadas codificações internas e *mixing bijections* [1], [2], [7], que serão explicados com maior pormenor mais à frente. A Figura 4 mostra os locais onde são aplicadas as intervenções de proteção.

Um atacante, em contextos de criptografia de caixa branca, também consegue aceder ao código fonte e modificá-lo facilmente para usar, por exemplo, uma função de decifração que poria em causa todos os dados encriptados que queremos proteger. Para salvaguardar esta situação, a implementação de criptografia de caixa branca tem como dados de entrada e saída, texto cifrado e decifrado, mas codificados. Assim sendo, em vez de transformar texto cifrado em texto decifrado, transforma texto cifrado codificado em texto decifrado codificado. A transformação posterior de texto decifrado codificado em texto decifrado descodificado acontece por um sistema externo à implementação de criptografia de caixa branca e só deverá ser efetuado no momento em que é necessária. Para a implementação de criptografia de caixa branca fazer a transformação de texto cifrado codificado em texto decifrado codificado, e vice-versa, são utilizadas codificações externas, que serão clarificadas numa secção posterior.

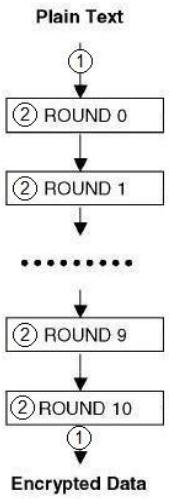

Figura 4 - Visão geral das modificações e do local onde são aplicadas (AES). Comparando com a Figura 3, note-se a inexistência de chaves como dados de entrada de rondas. 1 – Zona de aplicação das codificações externas. 2 – Zonas de aplicação das codificações internas e mixing bijections. (Adaptado de [19])

3.3.1 Codificações internas

Codificações internas são aplicadas em cada ronda dos algoritmos criptográficos de chave simétrica, na tabela de consulta correspondente à operação que envolve o estado e a sub-chave. É através das codificações internas que podemos alcançar a confusão, como definido por Shannon [7], [9]. As codificações nos dados de entrada e nos dados de saída são denominados *input encodings* e *output encodings*, respetivamente. A codificação nos dados de saída de uma ronda é inversa à codificação nos dados de entrada da ronda seguinte. Desta forma, as codificações ficam todas ligadas entre rondas, sendo cada *output encoding* sempre anulado na ronda seguinte pelo *input encoding*. (Ver Figura 5). As codificações internas são apenas substituição de bytes.

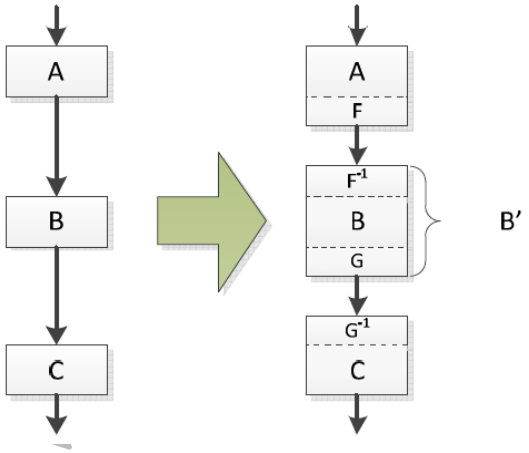

Figura 5 - Transformação de tabelas de consulta em tabelas de consulta com codificações internas. A, B, C representam as tabelas de consulta com a chave embutida sem qualquer proteção. F, G são operações de codificação que são juntas às tabelas de consulta e F^{-1}, G^{-1} as operações inversas, ou seja, as operações de descodificação. [20]

3.3.2 Mixing Bijections

Para a implementação de criptografia de caixa branca alcançar a difusão, como definido por Shannon [9], utiliza-se *mixing bijections* [7]. *Mixing bijections* são tranformações lineares invertíveis que, numa implementação de criptografia de caixa branca podem ser utilizadas de várias formas. Estas transformações ocorrem anterior e posteriormente às tabelas de consulta que envolvem a chave criptográfica usada. (Ver Figura 6).

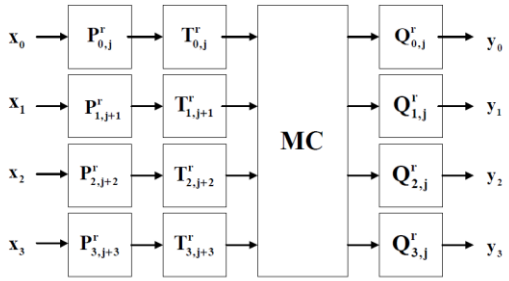

Figura 6 - Mixing bijections (P e Q), MixColumns (MC) e caixas-T com a chave embutida da solução C-WBC AES [2].

3.3.3 Codificações externas

Codificações externas são aplicadas no início e fim do algoritmo criptográfico, para que este transforme texto cifrado codificado em texto decifrado codificado na fase de decifragem e texto decifrado codificado em texto cifrado codificado na fase de cifragem. Isto delega a descodificação para uma fase posterior. *Muir* [7], em soluções de criptografia de caixa branca para Digital Rights Management (DRM), sugere que o conteúdo que é decifrado seja descodificado, ou seja, transformado de texto decifrado codificado em texto decifrado descodificado, apenas quando for necessário ser reproduzido [7]. Estas codificações são aplicadas no exterior do algoritmo criptográfico, ou seja, antes da primeira ronda e no fim da última.

3.4 Soluções de criptografia de caixa branca

Chow et al. [1] propôs a primeira solução de criptografia de caixa branca usando o algoritmo criptográfico de chave simétrica DES (C-WBC DES) em 2003 [1]. Na solução proposta, utiliza uma chave criptográfica fixa e define muitos conceitos dentro desta área. Apesar de apresentar duas soluções que apenas diferem na utilização de codificações externas, Chow et al. recomenda a variante em que as codificações externas são utilizadas, sendo esta a variante a que não aponta nenhum ataque possível. Referem também que o objetivo do seu trabalho é apenas dificultar a extração da chave utilizada para encriptação e que este poderá ser facilmente extrapolado para o uso do algoritmo criptográfico 3DES.

Os mesmos autores, em [2], propõem uma solução utilizando o algoritmo criptográfico AES (C-WBC AES) e já só consideram uma implementação que utilize codificações externas. Sugerem que num trabalho futuro seja possível mudar a chave a qualquer momento, ou seja, utilizar uma chave dinâmica na solução.

Link et al. encontrou um possível ataque à solução C-WBC DES que compromete a chave utilizada, baseada no método de *statistical bucketing* que já tinha sido um método utilizado para descobrir a chave criptográfica na implementação C-WBC DES [10]. O ataque detetado, só era possível devido ao tamanho do bloco ser de 4 X 4, como fora recomendado por Chow et al. Assim sendo, sugeriu alterações no tamanho bloco, passando de 4 X 4 para 8 X 4, de forma a impossibilitar o ataque que tinha descoberto. Ainda para prevenir o ataque previamente apontado, utilizando também o método *statistical bucketing*, sugeriu várias modificações nas caixas-S e caixas-T que fazem com que seja possível associar cada caixas-S com as caixas-T correspondentes.

Bringer et al. propôs uma implementação WBC AES diferente da C-WBC AES que, ao contrário da dos outros autores, não utiliza caixas-S [11]. Termos específicos são adicionados na primeira ronda e são anulados apenas na última, perturbando todos os valores das rondas intermédias. Os resultados finais da rotina de decifragem nem sempre estão corretos, por isso é necessário efetuar múltiplas execuções desta rotina, procedendo-se no fim à decisão através do voto sobre todas as execuções. Cada ronda conjuga um sistema da cifra original, sistema de polinómios e sistema aleatório de equações polinomiais, como podemos ver nos 3 blocos da figura 7, por esta mesma ordem. Os valores intermédios das rondas são falsos com uma grande probabilidade, ou seja, não se encontram num estado semelhante à da cifra original utilizada AES, visto terem alguns dados aleatórios que foram propositadamente introduzidos.

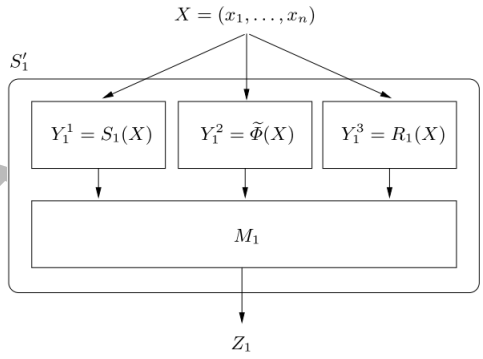

Figura 7 - Primeira ronda da solução WBC AES apresentada em [11].

Xiao et al. propôs uma melhoria ao nível da segurança da solução C-WBC AES, impossibilitando que *B-Attack* seja possível [12]. Para impossibilitá-lo modificou o tamanho das matrizes de ofuscação existentes antes das caixas-T e depois da etapa *MixColumns*, passando as primeiras (16 X 16) a terem metade do tamanho das segundas (32 X 32). Desta forma, qualquer potencial atacante não consegue relacionar diretamente as matrizes L e R da figura 8, como acontece nas matrizes P e Q da figura 6, sendo que era esta relação que possibilitava o *B-Attack*.

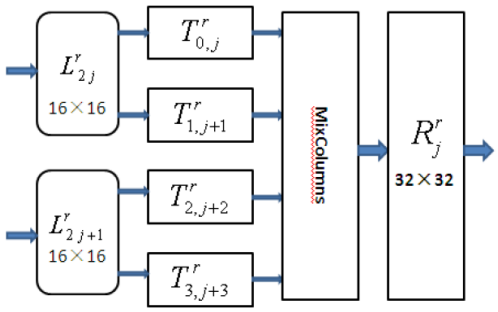

Figura 8 - *Mixing bijections* (L e R), *MixColumns* (MC) e caixas-T com a chave embutida da solução [12].

Park et al. aponta dois problemas às implementações de criptografia de caixa branca: 1 - Impossibilidade de atualização da chave criptográfica; 2 - Inadequabilidade a conteúdos muito grandes, visto que estas implementações são bastante mais lentas que os algoritmos criptográficos originais, sendo que a implementação C-WBC AES apresenta um desempenho 55 vezes pior em relação ao algoritmo original AES [13].

Para ultrapassar o primeiro problema, *Park et al.* sugere que o servidor tenha um gerador de tabelas protegidas que inclui um *Pseudo Random Number Generator* (PRNG). Este número é utilizado para fazer todas as *random bijections* das tabelas de consulta utilizadas na implementação WBC. O dado de entrada deste gerador é uma chave criptográfica nova e um valor que identifica o cliente, como, por exemplo, o seu nome. Com estes dados, o gerador de tabelas fornece todas as tabelas de consulta que têm de ser atualizadas na implementação WBC, ou seja, todas as tabelas de consulta que têm a chave criptográfica embutida.

Para aumentar o desempenho em conteúdos muito grandes, *Park et al.* recomenda o uso do modo de operação *Propagating Cipher Block Chaining* (PCBC) com alguns módulos a utilizar o algoritmo AES original e outros a utilizar a implementação WBC AES. Os módulos AES utilizam uma chave diferente do WBC AES devido à inexistência de qualquer proteção na chave. (Ver Figura 9).

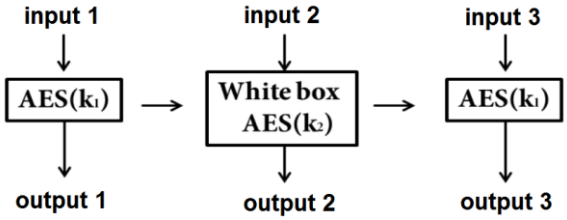

Figura 9 - Conjugação de módulos AES com módulos WBC AES, utilizando chaves diferentes em cada um destes [13].

Yoo et al. aponta os mesmos problemas a implementações WBC que *Park et al.* e soluciona o problema do desempenho de uma forma similar, embora utilize um modo de operação diferente que diz levar a um melhor desempenho das fases de cifragem e decifragem [14]. Também combina módulos de WBC AES e AES, utilizando uma chave diferente para cada um deste tipo de módulos. Afirma-se que na primeira operação do modo de operação que utiliza devemos ter um dado de entrada aleatório que leva a um dado de saída aleatório e, a partir desse momento, podemos utilizar os módulos AES. Sendo que a primeira operação é sempre efetuada utilizando WBC AES.

Os modos de operação sugeridos são o *Cipher Block Chaining* (CBC) e *Output FeedBack* (OFB). Compara o seu trabalho com o de *Park et al.*, apresentando um desempenho superior [14].

Shi et al. mostra uma implementação WBC utilizando o algoritmo criptográfico SMS4, mas segue os mesmos conceitos que a implementação WBC de *Chow et al.* [8]. O principal objetivo desta solução foi o seu desempenho, assim como uma menor quantidade de memória necessária pelas tabelas de consulta, visto que esta solução pretendia ser aplicada em contextos com capacidades computacionais muito menores do que um computador atual. Compara 3 versões, sendo uma baseada na solução C-WBC AES. As outras duas tentam melhorar o desempenho e aumentar a segurança da primeira.

3.4.1 Resumo comparativo

A Tabela 1 compara os principais aspetos de cada uma das implementações WBC referidas anteriormente.

Autores	**Ano**	**Algoritmo base**	**Chave**	**Modo de operação**	**Tamanho total tabelas de consulta**	**Resistente a *statistical bucketing attack***	**Resistente a *B-attack***
Chow et al. [1]	2002	DES	Fixa	–	2.3 MB	Não	Sim
Chow et al. [2]	2003	AES	Fixa	–	752 KB	Sim	Não
Link et al. [10]	2004	DES	Fixa	–	2.3 MB	Sim	Sim
Bringer et al. [11]	2006	AES	Fixa	–	–	Sim	Sim
Xiao et al. [12]	2009	AES	Fixa	–	20.5 MB	Sim	Sim
Park et al. [13]	2010	AES	Dinâmica	PCBC	–	Sim	Sim
Yoo et al. [14]	2012	AES	Dinâmica	CBC/OFB	–	Sim	Sim
Shi et al. [8]	2015	SMS4	Fixa	–	144.1 KB	Sim	Sim

Tabela 1 - Comparação entre várias soluções WBC. – representa um campo que não faz sentido para essa solução.

A tabela descreve algumas potenciais fragilidades a nível de segurança das soluções nas últimas duas colunas e o espaço utilizado pelas tabelas de consulta em cada solução. O ano da publicação é também referido, assim como o tipo de chave utilizado: chave dinâmica ou fixa. O modo de operação para conjugar a cifra é avançado em apenas dois trabalhos, sendo que é indicado na tabela quais os modos de operação utilizados em cada um deles. AES é atualmente o algoritmo padrão de criptografia de chave simétrica, nunca tendo sido alvo de nenhum ataque conhecido apesar de inúmeras tentativas devido à sua popularidade e tendo sido analisado, aprovado e definido como padrão por especialistas há menos tempo que outros (DES, por exemplo, que atualmente se encontra desatualizado). A atualização da chave é uma caraterística bastante importante, visto que esta poderá ficar comprometida e, com a sua atualização, conseguimos, pelo menos, adiar algum potencial roubo de informação ou código. Assim sendo, o uso de chave dinâmica será fundamental.

Apesar de muitas das soluções referidas se mostrarem resistentes ao *B-attack*, posteriormente o ataque foi melhorado. Atualmente, estas soluções são vulneráveis à extração da chave criptográfica devido ao *B-attack* [15].

3.5 Conclusão

Nos últimos anos existiram várias contribuições que tornaram as soluções WBC mais mais seguras, mais rápidas e mais flexíveis. No entanto, continuam a aplicar os princípios que *Chow et al.* apresentou nos seus primeiros trabalhos, tais como codificações internas, *mixing bijections* e codificações externas. Estes são os princípios das mais variadas implementações WBC que tentam sempre melhorar o trabalho feito até então, seja a corrigir algumas falhas de segurança que possam existir, como fez *Xiao et al.*, seja para tornar estas soluções mais práticas, aumentando o seu desempenho nas situações em que isso se demonstra necessário, como fez *Park et al.*.

Os três princípios são aplicados em diferentes fases da cifra. As codificações internas, em conjunto com as *mixing bijections*, são aplicadas em cada ronda da cifra e confundem o atacante, com informação que não corresponde ao esperado, visto que numa cifra, espera-se que a chave, assim que observada, comprometa a segurança do sistema. Aqui, a chave encontra-se disfarçada, através destes dois princípios, e, por isso mesmo, o atacante enfrenta outro tipo de dificuldades. As codificações externas são aplicadas antes de a cifra começar e após o término desta.

As soluções de criptografia de caixa branca delegam responsabilidades de descodificação para sistemas externos à técnica devido às codificações externas, para que estes se possam adaptar da forma mais segura possível dentro do seu domínio. Assim sendo, estes sistemas poderão ter um grande impacto no nível de segurança do *software*, tendo uma importância similar à que têm na solução de criptografia de caixa branca. Isto porque um atacante deteta as debilidades e se essas se encontram no sistema externo à solução WBC, será nele que o ataque irá incidir e, quem sabe, aceder a propriedade intelectual de forma maliciosa.

Capítulo 4

JavaScript

4.1 Introdução

JavaScript é a linguagem do navegador de Internet e isso fá-la uma das linguagens de programação mais populares do mundo [16].

Esta linguagem de programação é de alto nível, sem tipos estáticos e é interpretada e não compilada.

Apesar de suportar novas definições de objetos por parte do programador, não é tipada, devido à não obrigatoriedade de uma variável ficar, durante todo o seu tempo de vida, com objetos do mesmo tipo. Assim sendo, durante a execução de uma aplicação, uma variável pode mudar o tipo de objeto que representa.

A caraterística do JavaScript mais relevante para este trabalho é o facto de ser uma linguagem interpretada e não compilada, o que permite que, facilmente, o código fonte do *software* seja alterado. Assim sendo, esta linguagem de programação é bastante dinâmica, podendo até mudar o seu próprio código fonte durante a sua execução.

Com toda esta flexibilidade e dinamismo, torna-se mais fácil para um utilizador mal intencionado, aceder a dados privados ou alterar o código fonte de uma aplicação que se requer que seja totalmente inacessível nestes pontos.

Neste capítulo, serão apresentadas algumas técnicas que são utilizadas por atacantes que aproveitam as funcionalidades e dinamismo desta linguagem de programação.

4.2 Técnicas de ataque

Em JavaScript existem várias técnicas que permitem alterar o funcionamento normal de uma aplicação, aceder a dados ou modificar esses dados. Algumas técnicas utilizadas são: (1)

Shadowing; (2) *Emulation*; (3) *Hooking*; (4) *Prototype poisoning* [17]. Estas técnicas serão clarificadas de seguida.

4.2.1 Shadowing

É possível substituir a utilização de objetos nativos do JavaScript, criando variáveis globais com o mesmo nome desses objetos. Assim sendo, um código fonte em vez de utilizar os objetos nativos, utiliza objetos que podem ser adulterados e que têm propriedades ou valores diferentes do objeto nativo que se pretendia aceder.

4.2.2 Emulation

Quando se acede a uma variável de um objeto, caso tenha sido definida uma função que retorne o valor dessa variável, o que é obtido é o valor que essa função retornou. Assim sendo, não se acede ao estado interno do objeto e o comportamento deste pode ser simulado mais facilmente.

4.2.3 Hooking

Hooking é um conceito de programação que também se aplica em JavaScript. Este conceito manipula chamadas de funções, eventos ou mensagens. Em JavaScript, permite aceder a objetos assim como modificar funções, adicionando outras antes ou depois das que se pretende modificar.

Este conceito é particularmente importante na idealização da solução, visto que um atacante poderá fazer *hooking* do objeto que contém todo o código fonte a ser executado e, desta forma, derrotar todos os mecanismos de defesa utilizados.

4.2.4 Prototype poisoning

Com o uso desta técnica, é possível alterar o comportamento dos objetos nativos, como *strings* ou funções. Para isso basta redefinir determinada propriedade ou função que, a partir desse momento, o comportamento nativo do objeto terá.

4.3 Técnicas de defesa

Não existem formas diretas de impedir um atacante de utilizar as técnicas mencionadas anteriormente. Assim sendo, o máximo que pode ser feito é aplicar alguns mecanismos que dificultem a sua tarefa.

Esses mecanismos podem ser as próprias técnicas de ataque da linguagem. Desta forma um atacante não consegue entender facilmente o que está a acontecer e é enganado pelos mecanismos de defesa que são utilizados. Por exemplo, *prototype poisoning* pode ser utilizado para retornar valores errados a um atacante, deixando mais longe de ser bem sucedido.

Além deste tipo de mecanismos, podem existir outros que impedem a adulteração e depuração do código fonte.

Um conjunto de vários mecanismos diferentes será, sem dúvida, um desafio muito grande que um atacante terá de ultrapassar.

4.4 Funções

Em JavaScript existem várias formas de definir funções. As duas formas mais comuns de o fazer são através de expressões de função e declarações de função. Apesar de poder parecer que esta é uma questão meramente sintática, existem outras ligeiras alterações que devem ser referidas.

A diferença com maior relevância para este documento prende-se com o momento em que estas podem ser convertidas para texto.

As declarações de função têm uma sintaxe semelhante a funções de outras linguagens de programação como, por exemplo, o Java. Já as expressões de função atribuem um valor a uma variável que será de seguida o nome da função que se definiu.

Devido a esta atribuição de uma função a uma variável, a sua representação em texto só poderá ser obtida depois da atribuição. Já numa declaração de função, a sua representação em texto poderá ser obtida mesmo antes do local onde se encontra definida.

4.5 Expressão de função imediatamente invocada (IIFE)

Este tópico é uma especificidade desta linguagem, não sendo uma funcionalidade da maioria das linguagens de programação.

Uma expressão de função imediatamente invocada é uma função que é imediatamente executada com os parâmetros que lhe são atribuídos. Não existe nenhuma referência posterior para esta função. Embora possa existir apenas para o valor de retorno desta.

4.6 Strict mode

Em JavaScript, existe um modo que restringe algumas funcionalidades da linguagem. Este modo é designado por *strict mode* e é utilizado com a diretiva *"use strict"*. Neste modo não é possível declarar variáveis dentro da função de avaliação de código (eval), apagar variáveis ou

funções, fazer *prototype poisoning* da função de avaliação de código (eval), entre outros pontos importantes mas com menor relevância neste documento.

4.7 Conclusão

Existem várias técnicas que podem comprometer dados sensíveis, em JavaScript. Dados esses que, na nossa solução, poderão ser o código fonte de uma aplicação que queremos proteger.

Assim sendo, é de extrema importância a prevenção destes possíveis ataques, assim como outros, que possam derrotar todos os mecanismos de segurança da solução que será proposta.

Algumas propriedades do JavaScript podem permitir a um atacante ter um maior número de ferramentas que o deixam mais próximo dos seus objetivos. Mas essas mesmas propriedades também podem permitir mais e melhores mecanismos de defesa e que terão de ser exploradas ao máximo.

Capítulo 5

Descrição da solução

5.1 Visão geral

Para uma aplicação JavaScript ser protegida deve ser efetuada a sua transformação, encriptando uma parte do seu código fonte. Desta forma, a aplicação, antes de ser distribuída, passará pela ferramenta de transformação de código fonte, onde os ficheiros JavaScript serão modificados.

Nessa transformação, todas as funções ficam com o seu código fonte encriptado, sendo a sua decifração efetuada no cliente durante a fase de interpretação do código. A interpretação é feita uma única vez, no início da execução da aplicação.

A transformação que protege a aplicação JavaScript apenas protege funções, sendo por isso necessário que todo o código esteja dentro de funções para ser protegido.

5.2 Utilização da solução

Para utilizar a solução de criptografia de caixa branca para aplicações JavaScript, um desenvolvedor da aplicação necessita de fornecer o código fonte da sua aplicação. No servidor, a chave e as tabelas derivadas da chave são geradas aleatoriamente por cada ficheiro que o desenvolvedor envia para o servidor. Depois dessa geração, cada ficheiro é transformado, encriptando algumas partes do corpo da função e adicionando o código de decifração e de inicialização das tabelas de consulta.

A decifração do corpo das funções é efetuada pelo próprio código fonte quando é interpretado, utilizando uma expressão de função imediatamente invocada. Deste modo, um programador apenas necessita de passar a sua aplicação pela ferramenta de transformação de

código fonte, sendo todo o processo de decifração feito de forma completamente automática pelo próprio código fonte que foi transformado (Ver Figura 10).

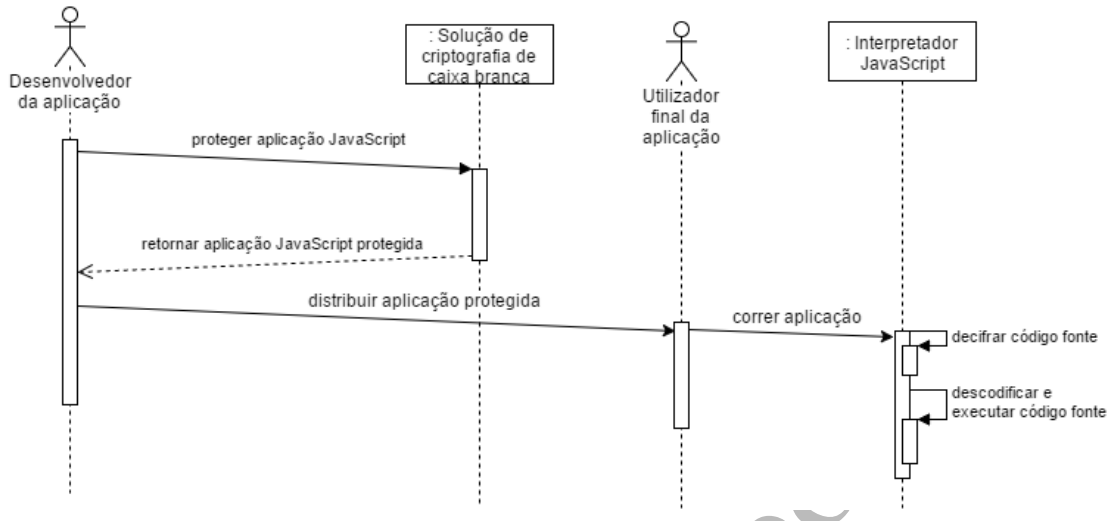

Figura 10 - Diagrama de sequência da utilização da solução de caixa branca para aplicações JavaScript

O código transformado torna-se inescrutável. Um atacante não conseguirá entender e modificar o que uma função fará, pelo menos de forma estática sem modificar e perceber a estrutura da solução.

5.3 Arquitetura da solução

A solução encontra-se organizada em três módulos principais: (1) Transformador da AST, (2) Algoritmo de criptografia de caixa branca, (3) Gerador dinâmico de tabelas.

O módulo transformador da AST é responsável por modificar as funções e adicionar a expressão de função imediatamente invocada. Para isso, necessita de utilizar os outros dois módulos que fornecem algumas ferramentas essenciais para a solução.

O módulo do algoritmo de criptografia de caixa branca fornece-se o *software* para cifrar e decifrar, assim como as tabelas de consulta que são utilizadas nessas duas operações.

O módulo gerador dinâmico de tabelas, fornece também *software* mas com um propósito diferente. Este módulo modifica as tabelas de acordo com um valor de *hash* e providencia *software* para a reconstrução correta das tabelas originais a partir desse valor de *hash*.

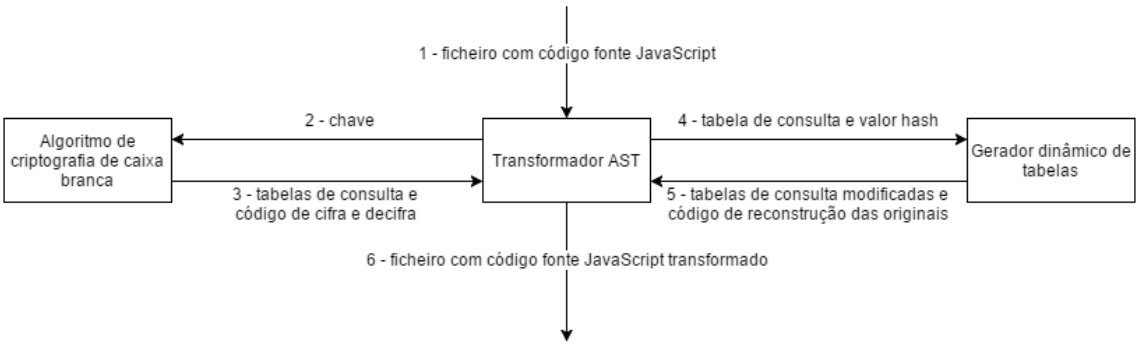

Figura 11 – Arquitetura da solução com três módulos principais.

Na Figura 11 pode-se observar os principais módulos da solução com os seus dados de entrada e saída. Cada dado de entrada e saída contém um número que se refere à ordem pela qual as interações entre os módulos acontecem. A chave é o único dado que é gerado aleatoriamente.

5.4 Caso de utilização

A solução proposta, apesar de poder ser utilizada em todos os ambientes em que o JavaScript se insere, fará ainda mais sentido em aplicações móveis híbridas. Estas aplicações são desenvolvidas como qualquer outra aplicação web, utilizando tecnologias como HTML5 e JavaScript. Entre algumas ferramentas para desenvolver este tipo de aplicações estão Apache Cordova e PhoneGap.

Nestas aplicações o código fonte JavaScript é distribuído e guardado em todos os dispositivos móveis que instalarem a aplicação. Esse factor acentua a necessidade das aplicações serem protegidas visto que estão acessíveis em inúmeros dispositivos.

5.5 Algoritmo de criptografia de caixa branca

O algoritmo de criptografia de caixa branca adotado segue os mesmos conceitos que a maioria das soluções no estado da arte: codificações internas e externas e *mixing bijections*, sendo uma adaptação do algoritmo AES.

As codificações internas e *mixing bijections* utilizadas são semelhantes às descritas por *Chow et al.* [2], assim como a maioria dos algoritmos de criptografia de caixa branca, e as codificações externas semelhantes às descritas por *Muir* [7]. A escolha por estas codificações externas deve-se à sua maior simplicidade e à inexistência de alguma evidência que demonstre um nível de segurança diferente em relação às codificações externas descritas por *Chow et al.* [2].

Nesta solução estão presentes 3 tipos de tabelas:

(a) Tipo 1, também designadas por caixas-T, contêm a operação *AddRoundKey*, *SubBytes* e uma parte de *MixColumns*.

(b) Tipo 2, também designadas por tabelas XOR, contêm a operação XOR que faz parte da operação MixColumns e das *mixing bijections*.

(c) Tipo 3, também designadas por tabelas das *mixing bijections*, contêm exclusivamente operações das *mixing bijections*, ou seja, multiplicação de matrizes.

Todas estas tabelas encontram-se codificadas internamente de forma concatenada entre tabelas consecutivas. A primeira e última tabela de consulta não apresentam codificações internas nos dados de entrada e de saída, respetivamente. As codificações internas já se encontram explicadas em capítulos anteriores com detalhe suficiente pelo que não serão esmiuçadas novamente.

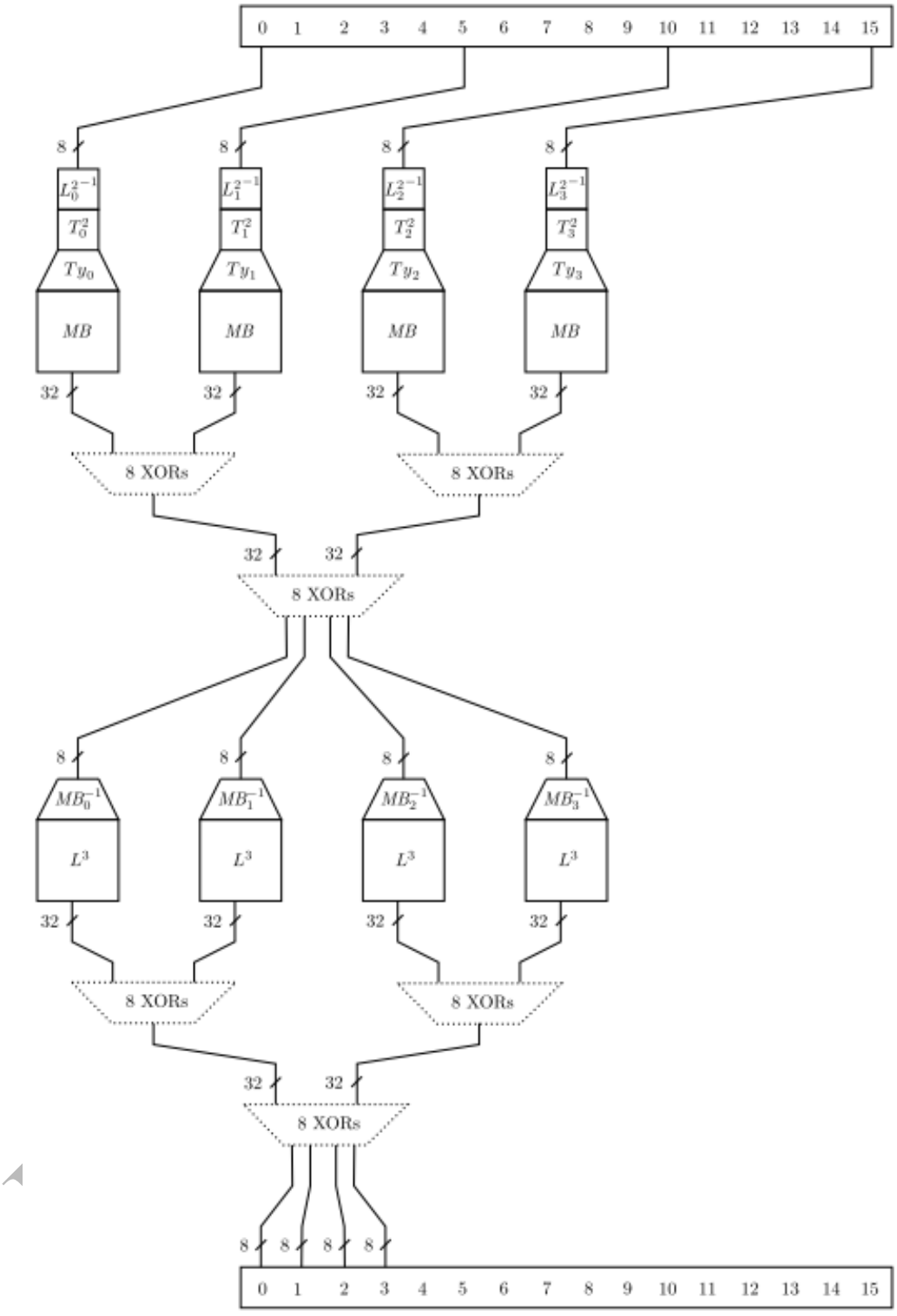

Figura 12 - Esquema de uma das rondas da solução de criptografia branca. Pode-se visualizar o conteúdo de cada uma das tabelas e o fluxo dos bytes 0, 5, 10 e 15 na ronda. [7]

5.5.1 Etapas de cifra e decifra

Para todas as rondas serem semelhantes e a cifração e decifração da solução de criptografia de caixa branca, baseada no algoritmo AES, estarem contidas no mesmo tipo e número de tabelas, é necessária uma ligeira modificação da ordem das operações do AES. Esta alteração não compromete em ponto algum o nível de segurança da solução, pois estas alterações podem também ser utilizadas pelo algoritmo AES sem que exista algum comprometimento do seu nível de segurança. A cifra tem o mesmo resultado final com estas ligeiras alterações que são apenas uma reordenação ligeira das operações (Ver Figura 13 e 14).

```
state <- plaintext
for r = 1 ... 9
        ShiftRows (state)
        AddRoundKey (state, ^k_{r-1})
        SubBytes (state)
        MixColumns (state)
ShiftRows (state)
AddRoundKey (state, ^k9)
SubBytes (state)
AddRoundKey (state, k10)
ciphertext <- state
```

Figura 13 - Cifra no algoritmo de criptografia de caixa branca AES. O acento circunflexo significa que a chave sofreu a operação *ShiftRows*.

```
state <- ciphertext
ShiftRowsInv (state)
AddRoundKey (state, ^k10)
SubBytesInv (state)
AddRoundKey (state, k9)
MixColumnsInv (state)
for r = 8 ... 1
        ShiftRowsInv (state)
        SubBytesInv (state)
        AddRoundKey (state, kr)
        MixColumnsInv (state)
ShiftRowsInv (state)
SubBytesInv (state)
AddRoundKey (state, k0)
plaintext <- state
```

Figura 14 – Decifra no algoritmo de criptografia de caixa branca AES. O acento circunflexo significa que a chave sofreu a operação *ShiftRows*.

5.5.2 Mixing Bijections

Mixing bijections são aplicadas nas tabelas que contêm a operação com a chave criptográfica. Na solução são utilizadas duas *mixing bijections*.

Nas rondas 2 a 10 existem 16 mixing bijections com 8 *bits* de entrada e saída (L^{-1}) e são aplicadas nos dados de entrada das caixas-T. A operação inversa destas *mixing bijections* são efetuadas na ronda anterior, na tabela das *mixing bijections*.

Nas rondas 1 a 9 existem 4 *mixing bijections* com 32 *bits* de entrada e saída (MB) e são aplicadas nos dados de saída das caixas-T. A operação inversa destas *mixing bijections* são efetuadas na mesma ronda, na tabela das *mixing bijections*.

Cada *mixing bijection* é uma matriz invertível em GF(2), sendo gerada aleatoriamente como descrito por *Xiao et al.* [18]. O tamanho dos valores de entrada e saída das *mixing bijections* variam de acordo com o tamanho da matriz correspondente, sendo que existe a multiplicação da matriz com os dados de entrada.

Como as caixas-T têm 8 bits como dado de entrada, L^{-1}, não precisam de nenhum tipo de modificação nos dados de entrada, assim como MB, visto que os dados de saída das caixas-T têm 32 *bits* de tamanho nos dados de saída.

Já nas operações inversas de L^{-1} e MB, a situação é completamente diferente. Na tabela das *mixing bijections*, MB^{-1}, são aplicadas nos dados de entrada, que têm 8 *bits*, e L são aplicadas nos dados de saída, que têm 32 *bits*.

Cada MB^{-1} adiciona 24 *bits* a zero em posições que dependem da posição da tabela das *mixing bijections* (Ver Figura 15).

$$MB^{-1}\begin{bmatrix} z_0 \\ z_1 \\ z_2 \\ z_3 \end{bmatrix} = MB^{-1}\begin{bmatrix} z_0 \\ 0 \\ 0 \\ 0 \end{bmatrix} \oplus MB^{-1}\begin{bmatrix} 0 \\ z_1 \\ 0 \\ 0 \end{bmatrix} \oplus MB^{-1}\begin{bmatrix} 0 \\ 0 \\ z_2 \\ 0 \end{bmatrix} \oplus MB^{-1}\begin{bmatrix} 0 \\ 0 \\ 0 \\ z_3 \end{bmatrix}$$

Figura 15 - Decomposição de uma *mixing bijection* com 4 *bytes* de dado de entrada em quatro *mixing bijections* com 1 *byte* de dado de entrada cada. Aqui podemos encontrar o motivo da existência de tabelas XOR após as tabelas de *mixing bijections*. [7]

Cada L, que tem 32 *bits* de dados de entrada e saída, é formada por concatenação de 4 L^{-1}, que têm 8 *bits* de dados de entrada e de saída (Ver Figura 16). É relevante relembrar que, como estas mixing bijections são invertidas na ronda posterior, a concatenação terá de ser feita tendo em conta a operação *ShiftRows* que é efetuada no início de cada ronda.

$$L^3 = L_0^3 \| L_{13}^3 \| L_{10}^3 \| L_7^3$$

Figura 16 - Concatenação de quatro *mixing bijections* com 1 *byte* de dado de entrada cada numa *mixing bijection* com 4 *bytes* de dado de entrada. [7]

5.5.3 Codificações externas

As codificações externas são utilizadas para os dados de entrada e de saída da cifra serem codificados. Assim é necessária a utilização da cifra da solução e não de outra, mesmo sabendo a chave criptográfica. Isto faz com que esta cifra tenha as suas especificidades, podendo ser única ou, pelo menos, rara. Os dados aleatórios das codificações externas, assim como a chave criptográfica, têm de ser os mesmos para que a cifra seja exatamente igual.

Na solução, as codificações externas são apenas substituição aleatória de *bytes*, integradas na primeira e última tabela de consulta do algoritmo, ou seja, nos dados de entrada e saída da cifra. As codificações externas aplicadas nos dados de saída da cifra são anuladas nos dados de entrada da outra fase da cifra, lembrando que esta é composta por fase de cifra e decifra.

Depois de ser efetuada a decifra é necessária a descodificação para se obter o resultado pretendido.

$$D'_k = G \circ D_k \circ F^{-1}$$

Figura 17 - Inserção de codificações externas (G, F^{-1}) na cifra original (D_k). [7]

5.6 Modo de operação da cifra de bloco

Para combinar a cifra por blocos foi necessária a utilização de um modo de operação. O modo de operação utilizado foi o *Eletronic Codebook* (ECB) visto que a solução não necessita de esconder quaisquer possíveis padrões, mas sim o código fonte (Ver Figura 18). Além disso, qualquer padrão seria escondido pelas codificações externas pois estas são geradas aleatoriamente por cada *byte* do bloco.

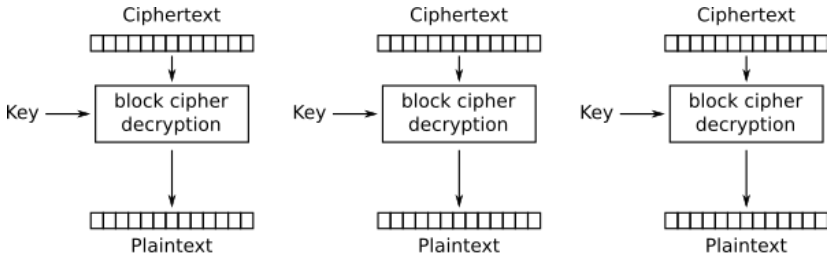

Figura 18 - Modo de operação ECB numa cifra de bloco. [21]

5.7 Anti-adulteração

Na solução existem dois de mecanismos de anti-adulteração que são descritos de seguida.

5.7.1 Mecanismo 1

As tabelas de consulta utilizadas na decifração são geradas a partir do valor de uma *hash* (Ver Figura 19). Esta *hash* é calculada a partir do código fonte das funções protegidas e da expressão de função imediatamente invocada onde este mecanismo é utilizado e onde as tabelas de consulta são geradas. Isto faz com que, caso as funções protegidas sejam adulteradas, a decifração não se concretize como esperado, decifrando o código de forma incorreta. Assim sendo, o resultado da decifração não será algo que seja possível executar, tendo valores completamente inesperados.

```
var EE_GReal;
var MBTablesReal;
var XOR4TablesReal;
var XOR3TablesReal;
var XOR2TablesReal;
var XOR1TablesReal;
var LTablesReal;

(function iife(LTables, XOR1Tables, XOR2Tables, XOR3Tables,
    XOR4Tables, MBTables, EE_G) {
    // calculo valor da hash a partir da representação
    // em texto das funções a proteger
    var crypto = require('crypto');
    const hashSHA256 = crypto.createHash('sha256');
    // nomes das funções a proteger (iife + a)
    var hash = hashSHA256.update('' + a + '' + iife + '');
    hash = hash.digest('hex');

    // inicialização das tabelas de consulta
    var LTablesStrIni = addHashToTables(LTables);
    var XOR1TablesStrIni = addHashToTables(XOR1Tables);
    var XOR2TablesStrIni = addHashToTables(XOR2Tables);
    var XOR3TablesStrIni = addHashToTables(XOR3Tables);
    var XOR4TablesStrIni = addHashToTables(XOR4Tables);
    var MBTablesStrIni = addHashToTables(MBTables);
    var EE_GStrIni = addHashToTables(EE_G);
    LTablesReal = eval(LTablesStrIni);
    XOR1TablesReal = eval(XOR1TablesStrIni);
    XOR2TablesReal = eval(XOR2TablesStrIni);
    XOR3TablesReal = eval(XOR3TablesStrIni);
    XOR4TablesReal = eval(XOR4TablesStrIni);
    MBTablesReal = eval(MBTablesStrIni);
    EE_GReal = eval(EE_GStrIni);
}(..., ..., ..., ..., ..., ..., ...));

function a(c) {
    // código da função
}
```

Figura 19 - Mecanismo 1 de anti-adulteração presente na variável hash.

5.7.2 Mecanismo 2

No Mecanismo 2 ocorre a verificação de possíveis *prototype poisoning* em funções que são utilizadas na decifra ou na execução de código, assim como função eval (Ver Figura 20). Este mecanismo não consegue ser ultrapassado sem que o mecanismo 1 seja evitado, visto que estas verificações são efetuadas numa expressão de função imediatamente invocada que se encontra protegida pelo mecanismo 1.

```
(function iife(LTables, XOR1Tables, XOR2Tables, XOR3Tables, XOR4Tables, MBTables, EE_G) {
    // verificação existência de prototype poisoning
    if(Function.prototype.toString + ' ' != 'function toString() { [native code] } ')
        return;
    if (iife.prototype.toString + ' ' != 'function toString() { [native code] } ')
        return;
    if (eval + ' ' != 'function eval() { [native code] } ')
        return;

    ...
    ...
}(..., ..., ..., ..., ..., ..., ...));
```

Figura 20 - Mecanismo 2 de anti-adulteração. Caso a função eval tenha sido alterada ou a função que converte a função iife para texto também tenha sido alterada, não são criadas as tabelas de consulta da decifração.

5.7.3 Gerador dinâmico de tabelas

Para o Mecanismo 1 de anti-adulteração ser aplicado foi criado um gerador dinâmico de tabelas a partir do valor de uma *hash*.

As tabelas de consulta que são geradas no servidor são vetores multidimensionais com números inteiros entre 0 e 255. O valor de uma *hash* contém carateres que podem ser interpretados como números inteiros entre 0 e 15.

O gerador dinâmico de tabelas, no lado do servidor, subtrai a cada número presente nas tabelas de consulta o valor de um dos carateres da *hash*, sendo estes carateres percorridos de forma sequencial para cada número (Ver Figura 21).

Depois desta operação, as tabelas de consulta modificadas serão incorporadas no ficheiro de código fonte que se está a transformar. Nesse mesmo ficheiro, durante a sua interpretação, as tabelas de consulta originais são reconstruídas, percorrendo pela mesma ordem cada número da tabela de consulta e somando a cada número o valor do carater correspondente da *hash* (Ver Figura 21).

```
var hashValue = "bb1db2b0373ca08cbb06396615f3231d1acf4436898d7257f5c6e7d2fec138d4";
// cada carater é convertido para inteiro como se fosse hexadecimal
var h0 = parseInt(hashValue[0], 16);
var h1 = parseInt(hashValue[1], 16);
var h2 = parseInt(hashValue[2], 16);
var h3 = parseInt(hashValue[3], 16);
var h4 = parseInt(hashValue[4], 16);
var h5 = parseInt(hashValue[5], 16);
var h6 = parseInt(hashValue[6], 16);
var h7 = parseInt(hashValue[7], 16);

// tabela gerada e presente apenas no servidor
var tabelaOriginal = [[[[3, 13, 11, 5, 2, 9, 5, 5], ...], ...], ...];
// tabelaParametroIIFE = tabelaOriginal - hashValue
var tabelaParametroIIFE = [[[[-8, 2, 10, -8, -9, 7, -6, 5], ...], ...], ...];
// string com inicialização da tabela
var stringParaInicializarTabela = "[[[[-8+h0, 2+h1, 10+h2, -8+h3, -9+h4, 7+h5, -6+h6, 5+h7], ..], ..], ..]";
// tabelaInicializadaNaIIFE = tabelaParametroIIFE + hashValue
var tabelaInicializadaNaIIFE = [[[[3, 13, 11, 5, 2, 9, 5, 5], ...], ...], ...];
```

Figura 21 - Construção das tabelas de consulta no cliente e geração de tabelas de consulta modificadas no servidor

5.7.4 Função de hash

No Mecanismo 1 de anti-adulteração a função de *hash* escolhida deve ter o menor número de colisões possível para minimizar a hipótese de funções diferentes, utilizadas como dado de entrada da função de *hash*, tenham o mesmo valor da função de *hash*. O algoritmo de *hash* escolhido foi o SHA-256 por ser uma função de *hash* criptograficamente segura.

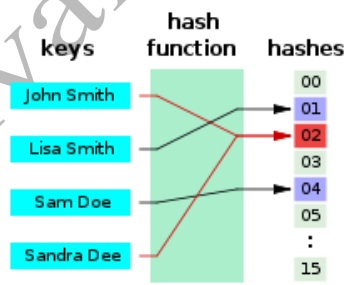

Figura 22 - Representação gráfica do funcionamento de uma função de *hash* [22]. No valor de *hash* 02, assinalado a vermelho, acontece uma colisão.

5.8 *Prototype poisoning* da função de descodificação e avaliação

A avaliação do código fonte é feita por um função de avaliação (eval), mas antes disso é necessário proceder à descodificação. Para disfarçar esta descodificação, faz-se com que a representação da função de descodificação e avaliação (ev) tenha a mesma representação,

quando convertida para texto, que a função de avaliação eval. Desta forma tenta-se enganar o atacante.

Para mudar a representação em texto da função de descodificação e avaliação, recorre-se ao *prototype poisoning* da função que converte essa função para texto (Ver Figura 23).

```
1  ev.toString = eval.toString;
```

Figura 23 - *Prototype poisoning* da função de descodificação e avaliação (ev).

5.9 Transformação da árvore sintática abstrata

A transformação da árvore sintática abstrata (AST) é efetuada em duas componentes.

A primeira componente transforma cada expressão e declaração de função, sendo que o efeito de anti-adulteração só existe nas declarações de função que são definidas no escopo global. A segunda componente realiza a inserção de uma expressão de função imediatamente invocada (IIFE) no início do ficheiro.

5.9.1 Modificação de expressões e declarações de função

A modificação de cada expressão e declaração de função é efetuada a cada elemento do corpo da função da árvore sintática, sendo que a transformação de um elemento só é efetuada se não existir nenhuma expressão de retorno da função. Isto deve-se à impossibilidade da função de avaliação (eval) executar a expressão de retorno quando está presente dentro da função a retornar. Assim sendo, poderão existir casos em que é necessário uma pesquisa em profundidade da árvore sintática, encriptando tudo o que não seja uma expressão de retorno. Caso uma expressão de retorno esteja contida numa ou várias estruturas de controlo de fluxo, estas serão percorridas em profundidade, encriptando cada elemento do corpo dessa estrutura de controlo de fluxo sempre que possível. (Ver Figura 24).

```
// ORIGINAL
function isTheDoubleBiggerThanOne(c) {
    var doubleC = c * 2;
    if (doubleC > 1) {
        return true;
    } else {
        return false;
    }
}

// TRANSFORMADO
function isTheDoubleBiggerThanOne(c) {
    var doubleC = ev(decipherMemoized([228, 43, 113, 4, 92,
        14, 251, 148, 235, 32, 196, 133, 20, 91, 48, 105
    ]));
    if (doubleC > 1) {
        return true;
    } else {
        return false;
    }
}
```

Figura 24 - Exemplo de uma declaração de função que foi transformada. Esta declaração de função tem expressões de retorno (return) dentro de uma estrutura de controlo de fluxo.

5.9.2 Inserção de expressão de função imediatamente invocada

Nesta IIFE é criado o Mecanismo 1 de anti-adulteração e são inicializadas as tabelas de consulta utilizadas na decifração a partir do valor da *hash*. É também nesta IIFE que são inicializadas as funções de decifração e descodificação.

As verificações de *prototype poisoning* na função de avaliação (eval) de código estão contidas nesta função, não sendo construídas as tabelas de consulta da decifração caso a função eval tenha sido modificada.

Por questões de desempenho, esta função executa todas as decifrações que estão presentes no ficheiro, para que o seu resultado fique em memória, não existindo a necessidade de executar a função de decifração mais nenhuma vez durante a execução do ficheiro JavaScript. (Ver Figura 25).

```
(function iife(LTables, XOR1Tables, XOR2Tables, XOR3Tables,
    XOR4Tables, MBTables, EE_G) {
    // calculo valor da hash a partir da representação
    // em texto das funções a proteger
    var crypto = require('crypto');
    const hashSHA256 = crypto.createHash('sha256');
    var hash = hashSHA256.update('' +
        isTheDoubleBiggerThanOne + '' + iife + '');
    hash = hash.digest('hex');
    var h0 = parseInt(hash[0], 16);
    var h1 = parseInt(hash[1], 16);
    ...
    var h63 = parseInt(hash[63], 16);

    // verificação prototype poisoning de algumas funções
    if (eval + ' ' != 'function eval() { [native code] } ')
        return;
    ...

    // inicialização das tabelas de consulta
    var LTablesStrIni = addHashToTables(LTables);
    var XOR1TablesStrIni = addHashToTables(XOR1Tables);
    ...
    LTablesReal = eval(LTablesStrIni);
    XOR1TablesReal = eval(XOR1TablesStrIni);
    ...

    // definição de algumas funções
    // como decifração e descodificação
    ev = eval;
    decipher = function(array) {
        var str = convertArrayToString(array);
        return WBC_AES_EncryptOrDecrypt_ECB(str,
            LTablesReal, XOR1TablesReal, XOR2TablesReal,
            XOR3TablesReal, XOR4TablesReal, MBTablesReal);
    };
    decipherMemoized = memoize(decipher, function(array) {
        return array.toString();
    });
    ...

    // execução de todas decifrações
    // existentes neste código fonte
    decipherMemoized([228, 43, 113, 4, 92, 14, 251, 148,
        235, 32, 196, 133, 20, 91, 48, 105
    ]);
        ...
}([[[[-5, 2, 1, -15, -4, -7, -7, 8], ...], ...], ...]));
```

Figura 25 - Visualização da IIFE com todas as suas principais operações

Capítulo 6

Implementação e Validação

6.1 Tecnologias utilizadas

As tecnologias seguintes foram utilizadas no desenvolvimento da solução de criptografia branca para aplicações JavaScript:

- **JavaScript** - A linguagem de programação utilizada durante todo o desenvolvimento da solução foi o JavaScript.
- **NodeJS** - Interpretador de código fonte JavaScript, normalmente utilizado no servidor. O seu uso foi constante durante todo o desenvolvimento.
- **Npm** - Gestor de pacotes para JavaScript.
- **Crypto** - Biblioteca em JavaScript utilizada para gerar números aleatórios criptograficamente seguros e para a utilização da função de *hash* criptográfica SHA-256.
- **Underscore-node** - Biblioteca em JavaScript para auxiliar operações com vetores, como comparação e cópia de vetores multidimensionais.
- **Lodash.memoize** - Biblioteca em JavaScript utilizada na *memoization* da função de decifra. *Memoization* é explicada no subcapítulo seguinte.
- **Esprima** - Gerador da AST a partir de um código fonte JavaScript.
- **Escodegen** - Gerador de código fonte JavaScript a partir da AST.
- **Git** - Sistema de controlo de versões utilizado no desenvolvimento.
- **Jscrambler engine** - Foi utilizado o motor da Jscrambler para gerar e percorrer a AST e gerar o código fonte. Este motor utiliza o Esprima e o Escodegen sendo uma ferramenta útil por ter um nível mais alto de abstração. Este motor utiliza o padrão de *software visitor* para percorrer os nós da AST, o que faz com que apenas seja necessário o desenvolvimento das funções que visitam e modificam cada nó da AST.

- **Benchmark.js** - Biblioteca em JavaScript para executar testes de desempenho de funções.

6.2 Detalhes da implementação

Para clarificar alguns pontos muito específicos da solução, neste capítulo são descritos de forma mais pormenorizada esses mesmos pontos.

6.2.1 Memoization

O resultado da função de decifração é guardado em memória para um dado de entrada. Desta forma, só existe a necessidade da execução desta função uma vez para cada dado de entrada, havendo um impacto significativo no desempenho da aplicação apenas no início da sua execução.

A *memoization* permite acelerar o desempenho da aplicação depois da interpretação do código fonte, visto que a função de decifração demora várias vezes mais a executar do que quando não tem *memoization*.

O nível de segurança da solução pode diminuir com a utilização de *memoization*, mas este é um ponto essencial para melhorar o desempenho da aplicação. Será difícil de prever a vantagem que um atacante terá com a utilização de *memoization* por parte da solução visto que, depois da decifra, é necessário uma descodificação. Caso o atacante não consiga entender a necessidade da descodificação, este não conseguirá tirar nenhuma vantagem do ataque à *memoization*.

6.2.2 Matrizes GF(2)

Matrizes GF(2) são matrizes num corpo de Galois de dois elementos, também chamado por corpo finito de dois elementos. Estes dois elementos são valores ou números, podendo assumir o valor zero ou um. Isto significa que cada elemento da matriz tem o valor zero ou um.

A soma e multiplicação de elementos do corpo de Galois de dois elementos é efetuada de forma ligeiramente diferente do usual. Estas operações sofrem, no fim, sempre a operação de módulo 2.

Para o uso de *mixing bijections* no algoritmo de criptografia de caixa branca foi necessário o desenvolvimento de vários métodos e operações de matrizes em GF(2). Todos estes métodos foram necessários na geração de matrizes invertíveis aleatórias. Nestes métodos enquadram-se: (a) multiplicação de matrizes, (b) adição de matrizes, (c) inversão de matrizes, (d) redução da matriz à forma canónica através da eliminação Gauss-Jordan. A combinação destes métodos

com o algoritmo de geração aleatória de matrizes invertíveis descrito por *Xiao et al.* [18] leva à geração aleatória de *mixing bijections* de acordo com o recomendado.

6.2.3 Restrição de uso

Atualmente a solução não suporta totalmente o *strict mode* do JavaScript. Esta restrição deve-se apenas à declaração de variáveis dentro da função eval, que neste modo não é suportada. Em JavaScript existe um conceito de escopo um pouco diferente de algumas linguagens. O escopo é limitado ao nível da função ou do objeto. É possível, por exemplo, declarar uma variável dentro de uma estrutura de controlo de fluxo de uma função e modificar ou aceder a essa mesma variável em qualquer local da função, a partir do momento que a variável foi declarada.

Para ultrapassar esta restrição, seria necessário fazer uma pesquisa em profundidade, como a que é efetuada para as expressões de retorno, de forma a não encriptar as declarações de funções (Ver Figura 26).

```
// CASO DE RESTRIÇÃO
"use strict";
function getDoubleIfBiggerThanOne(c) {
    if (c > 1) {
        var doubleC = c * 2;
    } else {
        var doubleC = c;
    }
    return doubleC;
}

// TRANFORMAÇÃO DA FUNÇÃO COM A SOLUÇÃO ATUAL
"use strict";
function getDoubleIfBiggerThanOne(c) {
    ev(decipherMemoized([22, 4, 13, 41, 92, 14, 251, 48,
        235, 32, 196, 133, 20, 91, 48, 105, 1, 41, 130,
        2, 5, 86, 21, 8, 235, 39, 19, 33, 205, 91, 203, 10,
    ]));
    // doubleC não seria "encontrada" pois foi declarada
    // dentro de um eval
    return doubleC;
}

// POSSÍVEL TRANSFORMAÇÃO DA FUNÇÃO COM A SOLUÇÃO SEM RESTRIÇÃO
"use strict";
function isTheDoubleBiggerThanOne(c) {
    if (doubleC > 1) {
        var doubleC = ev(decipherMemoized([220, 1, 3, 150, 86,
            14, 251, 234, 220, 3, 196, 133, 110, 9, 84, 51
        ]));
    } else {
        var doubleC = ev(decipherMemoized([20, 96, 51, 48, 79,
            8, 33, 7, 202, 39, 97, 123, 32, 51, 21, 54
        ]));
    }
    return doubleC;
}
```

Figura 26 - Exemplo de uma função não suportada atualmente devido ao uso de *strict mode*. Para ultrapassar esta dificuldade, é necessário uma transformação como a apresentada.

6.3 Processo de desenvolvimento

Todo o desenvolvimento foi efetuado de forma iterativa, sem ciclos de desenvolvimento definidos, desde o algoritmo de criptografia de caixa branca até à transformação da árvore sintática abstrata, acrescentando sempre valor à solução.

O desenvolvimento do algoritmo de criptografia de caixa branca iniciou-se com o algoritmo AES utilizando também tabelas de consulta. À medida que o desenvolvimento foi avançando, foram adicionadas as especificidades do algoritmo de caixa branca: codificações internas, externas e *mixing bijections*. Nas *mixing bijections* foi necessário desenvolver todas as operações de matrizes visto que não foi encontrada nenhuma biblioteca com as operações de matrizes em corpos de Galois de 2 elementos (GF(2)).

O processo de idealização da transformação da árvore sintática foi efetuado um pouco por tentativa e melhoria. Cada modelo era avaliado em relação a cada potencial ataque e à sua potencial execução e desempenho. Podia-se admitir um modelo final diferente, com maior enfâse em tentar enganar e dificultar a leitura do código fonte de um potencial atacante mas não foi esse o caminho escolhido. Considerou-se este tipo de solução mais complexa para um atacante embora possa ser menos trabalhosa. A tentativa de enganar e dificultar a leitura do código fonte por parte de um atacante poderá ser feita em cima desta solução, sendo inegável o seu valor, enquanto o contrário já não seria efetuado tão facilmente.

6.4 Validação

Não será efetuada nenhuma análise do algoritmo de criptografia de caixa branca utilizado pois não é esse o objetivo deste documento e pela existência de inúmeros artigos na literatura sobre este tema.

Em relação ao nível de segurança da solução, pode-se afirmar que esta solução o aumenta.

O tempo que um possível atacante demorará a obter o código fonte original de um ficheiro dependerá muito do seu conhecimento em JavaScript visto que as barreiras poderão ser mais rapidamente ultrapassadas na área da linguagem de programação utilizada do que na criptografia de caixa branca.

O algoritmo de criptografia de caixa branca poderá não ser óbvio para o atacante, sendo o exercício da sua descoberta difícil. A chave criptográfica utilizada e as codificações externas terão de ser descobertas executando e modificando o código fonte JavaScript, o que consiste em mais uma dificuldade no ataque do algoritmo de criptografia de caixa branca e leva a que o contacto com esta linguagem de programação seja necessário.

A quantificação do nível de segurança da solução não será possível de ser feita, estando esta sujeita a testes de utilização onde cada "atacante" contratado poderá apresentar a vulnerabilidade que encontrou, melhorando posteriormente o modelo sempre que possível.

Apesar da impossibilidade de quantificação do nível de segurança da solução, existem alguns fatores que irão ditar o sucesso de um atacante, ordenados por diminuição de impacto: (a) familiriadade com a linguagem de programação JavaScript, (b) conhecimento das técnicas de ataque a aplicações JavaScript, referidas no capítulo 4 e (c) nível de criatividade, para ultrapassar os mecanismos de defesa existentes.

Apesar da importância diferente, todos estes fatores são determinantes para que um atacante consiga ser bem sucedido.

6.4.1 Anti-adulteração em funcionamento

É possível prever qual o caminho mais curto para que um atacante atinja os seus objetivos. Neste tópico tenta-se fazer essa mesma previsão e demonstrar que a solução oferece um nível de segurança maior em relação à não utilização de qualquer tipo de mecanismo de segurança.

Para começar, poder-se-á pensar que um atacante fará *prototype poisoning* da função de avaliação (eval) de código fonte de forma a mostrar qual o seu dado de entrada antes de executar. Para perceber mais facilmente qual o resultado que pertence a cada vetor passado à função de decifração, poderá mostrar também qual o dado de entrada desta função. Logo, fará também *prototype poisoning* da função de decifração para mostrar o seu dado de entrada (Ver Figura 27).

```
1  var referenciaEval = eval;
2  eval = (function (arg) {
3          console.log("argument eval:" + arg);
4          return referenciaEval.call(this, arg);
5  });
6
7  var referenciaDecifracaoMemoized = decifracaoMemoized;
8  decifracaoMemoized = (function (arg) {
9          console.log("argument decifracao:" + arg);
10         return referenciaDecifracaoMemoized(arg);
11 });
```

Figura 27 - Possível tentativa para obter o código fonte original

Como existe uma verificação para detetar a existência de alterações na função eval (Ver Figura 20 e 25), as tabelas de consulta usadas na decifração não serão inicializadas e a decifração falhará. A aplicação nem sequer será executada. De seguida, um atacante tentará eliminar essa verificação, modificando a IIFE. Como as tabelas de consulta são geradas a partir do valor da função de *hash* que recebe como argumento o código fonte da IIFE (Ver Figura 21 e 25), as tabelas serão, com elevado grau de certeza devido à ausência de colisões na função de *hash*, criadas erradamente e a decifra será erradamente efetuada. A aplicação não executará novamente.

6.4.2 Modificação do código fonte original

Apesar destes mecanismos elevarem o nível de segurança das aplicações JavaScript, é possível ultrapassá-los, embora um atacante tenha de perceber quais os mecanismos existentes e de que forma estes estão ligados. Neste caso, para ser bem sucedido, o mais fácil será calcular de alguma forma o valor da função de *hash* utilizado. Para isso será necessário copiar a representação em texto das funções passadas como argumento. Mas, em execução, esta só é possível de se obter dentro da IIFE, sendo que a IIFE se encontra protegida contra adulterações. A única forma de ultrapassar este problema, será copiar estaticamente a representação em texto das funções necessárias e, de seguida, calcular o valor da *hash*. Desta forma já poderá inicializar o valor correto da *hash*, deixando esta de ser calculado em tempo de execução.

A partir desse momento, todas as alterações serão possíveis, o que deixará o atacante com muito mais flexibilidade.

6.5 Desempenho

Neste capítulo é comparado o desempenho das funções originais em relação às funções transformadas. Além disso, é medido o aumento de tempo de interpretação de um código fonte devido à execução da IIFE que é sempre acrescentada. Por fim, mostra-se o aumento de tamanho de um código fonte depois da sua transformação.

Para os testes de desempenho foi utilizado um computador com as seguintes caraterísticas: (a) Processador: Intel® Core ™ i7-4510U @ 2.00 GHz 2.60 GHz, (b) RAM: 8 GB, (c) Ubuntu 15.10 64-bit (AMD64).

6.5.1 Tempo de execução IIFE

Função	Média (s)	Desvio padrão (s)	Margem de erro relativa (%)	Nº de execuções
IIFE	1.43	0.43	± 31	6

Tabela 2 - Estatísticas do teste de desempenho da IIFE

6.5.2 Tempo de execução de uma função

Aqui serão efetuadas comparações dos tempos de execução de funções antes e depois da sua transformação. As funções a comparar serão duas: (a) cálculo dos primeiros 20000 números de *Fibonacci* e (b) cálculo dos números primos entre 1 e 20000.

```
function fibonacciNumbers(limit) {
    var i;
    var fib = [];
    fib[0] = 0;
    fib[1] = 1;

    for (i = 2; i <= limit; i++) {
        fib[i] = fib[i - 2] + fib[i - 1];
    }

    return fib;
}
```

Figura 28 - Função original de cálculo dos primeiros números de *Fibonacci* (Adaptado de [23]).

```
function fibonacciNumbers1(limit) {
    ev = function(code) {
        var newCode = applyExternalEncodingsInBlockECBMemoized(code, EE_GReal);
        return eval(newCode);
    };
    var i;
    var fib = ev(decipherMemoized([4, 126, 13, 243, 71, 192, 28, 109, 198, 146, 215, 94, 167, 150, 28, 86]));
    ev(decipherMemoized([187, 241, 59, 15, 14, 30, 149, 207, 72, 40, 244, 136, 254, 83, 204, 147]));
    ev(decipherMemoized([103, 72, 43, 133, 16, 176, 5, 254, 145, 71, 186, 117, 154, 23, 141, 6]));
    ev(decipherMemoized([196, 113, 128, 117, 61, 94, 246, 56, 25, 159, 34, 254, 228, 5, 243, 254, 201, 245, 83,
        192, 246, 172, 57, 8, 242, 191, 156, 75, 94, 150, 56, 159, 100, 23, 190, 149, 15, 254, 104, 200, 121, 202,
        30, 72, 184, 220, 17, 148
    ]));
    return fib;
}
```

Figura 29 - Função transformada de cálculo dos primeiros números de *Fibonacci*.

```
function getPrimes(max) {
    var sieve = [], i, j, primes = [];
    for (i = 2; i <= max; ++i) {
        if (!sieve[i]) {
            primes.push(i);
            for (j = i << 1; j <= max; j += i) {
                sieve[j] = true;
            }
        }
    }
    return primes;
}
```

Figura 30 - Função original de cálculo dos números primos até um máximo [24].

```
function getPrimes(max) {
    ev = function(code) {
        var newCode = applyExternalEncodingsInBlockECBMemoized(code, EE_GReal);
        return eval(newCode);
    };
    var sieve = ev(decipherMemoized([80, 96, 64, 89, 175, 187, 242, 81, 100, 248, 20, 78, 218, 64, 170, 218])),
        i, j, primes = ev(decipherMemoized([80, 96, 64, 89, 175, 187, 242, 81, 100, 248, 20, 78, 218, 64, 170, 218]));
    ev(decipherMemoized([190, 62, 147, 202, 96, 190, 125, 2, 205, 111, 233, 99, 242, 100, 47, 68, 102, 255, 27, 173, 193,
        229, 221, 79, 211, 250, 5, 58, 45, 3, 107, 0, 22, 167, 167, 174, 6, 175, 217, 20, 180, 82, 189, 19, 210, 98, 239,
        28, 67, 15, 31, 67, 19, 33, 122, 220, 116, 29, 104, 32, 93, 63, 221, 0, 157, 93, 144, 131, 2, 84, 104, 105, 209,
        10, 189, 61, 148, 16, 196, 146, 40, 222, 168, 72, 57, 242, 182, 33, 87, 134, 49, 46, 231, 33, 22, 79
    ]));
    return primes;
}
```

Figura 31 - Função transformada de cálculo dos números primos até um máximo.

Funções	Média (ms)	Desvio padrão (ms)	Margem de erro relativa (%)	Nº de execuções
Nºs *Fibonacci* (original)	0.16	0.01	± 1	88
Nºs *Fibonacci* (transformada)	1.82	0.10	± 1	85
Nºs primos (original)	0.35	0.03	± 2	88
Nºs primos (transformada)	4.95	0.46	± 2	80

Tabela 3 - Estatísticas do teste de desempenho de 2 funções originais e transformadas.

Analisando os resultados da tabela 3, podemos verificar que a função com o cálculo dos números primos ficou, depois da sua transformação, com um desempenho pior em cerca de 14 vezes. Já a função com o cálculo dos números de *Fibonacci* piorou o seu desempenho em cerca de 11 vezes.

Este desempenho variará de acordo com a função utilizada, podendo ser muito díspar para funções com a mesma funcionalidade mas com uma estrutura diferente do código fonte.

Assim sendo, no futuro, talvez se deva proteger apenas alguns elementos do corpo da função, em vez de todos, como até agora é feito. Isto leva a que o nível de segurança esteja no mesmo patamar mas pode acelerar o desempenho significativamente.

6.5.3 Aumento do tamanho do código fonte

Devido à IIFE que é sempre adicionada em todos os ficheiros e às tabelas de consulta que recebe como argumento, o aumento do tamanho de um ficheiro ficou sempre entre os 2 MB e os 3 MB. As tabelas de consulta serão sempre o maior fator de aumento do tamanho do ficheiro que contém o código fonte, sendo expectável que na transformação ocorra sempre um aumento entre os 2 MB e os 3 MB, independentemente do tamanho do ficheiro original.

6.5.4 Possível futura melhoria

Para aumentar um pouco o desempenho será necessário usar como argumento da função de decifra que se encontra com *memoization*, uma *string*. A biblioteca que faculta a *memoization* utiliza sempre texto como dado de entrada, sendo que nesta solução existe sempre a conversão de um vetor para texto, o que faz com que o desempenho seja pior do que aquilo que poderia ser.

Apesar de ser um pequeno pormenor, poderá trazer melhorias significativas no desempenho da solução.

Outra melhoria possível será combinar elementos consecutivos das funções transformadas que foram encriptados. Dessa forma a decifração, embora com *memoization*, será efetuada num menor número de vezes, acelerando a execução.

Capítulo 7

Conclusões e Trabalho Futuro

7.1 Conclusões

A solução de criptografia de caixa branca para aplicações JavaScript aumenta o nível de proteção das aplicações utilizando mecanismos de anti-adulteração que terão de ser ultrapassados por um atacante para conseguir alcançar o seu objetivo.

Apesar do nível de segurança aumentar, um atacante com um bom nível de conhecimento da linguagem e das técnicas de ataque existentes em JavaScript e um bom nível de criatividade, conseguirá - mas com dificuldade - ultrapassar as barreiras que esta solução coloca.

Esta solução faz com que o tempo de execução das funções aumente significativamente em média. Pelos testes de desempenho efetuados, o desempenho das funções transformadas foi mais de 10 vezes mais lento do que as funções originais, mas estes resultados dependerão sempre da função que se transforma. Alguns melhoramentos no desempenho foram propostos embora seja desconhecida a sua magnitude na melhoria do desempenho.

7.2 Trabalho futuro

A solução poderá ainda ser melhorada, acrescentando mecanismos anti-depuração à solução que impedem o uso de depuradores.

Outro ponto importante que poderá ser aperfeiçoado é limitar ao essencial o escopo das tabelas de consulta da decifra, isto é, restringindo-as ao escopo das funções que são decifradas durante a interpretação do código fonte.

Por fim, toda a solução poderá ser potencializada no sentido de dificultar mais a interpretação do código fonte por parte de um atacante, podendo até usar algumas técnicas de

ataque descritas no capítulo 4 para confundir um atacante. Alguns melhoramentos no desempenho poderão ser feitos de acordo com o enunciado na secção 6.5.4.

Referências

[1] S. Chow, P. Eisen, H. Johnson, and P. C. Van Oorschot, "A white-box DES implementation for DRM applications," *Digit. Rights Manag.*, pp. 1–15, 2003.

[2] S. Chow, P. Eisen, H. Johnson, and P. C. Van Oorschot, "White-Box Cryptography and an AES Implementation," *Sel. Areas Cryptogr.*, pp. 250–270, 2003.

[3] B. Barak, O. Goldreich, and R. Impagliazzo, "On the (im) possibility of obfuscating programs," *Adv. Cryptol. ...*, no. Im, pp. 1–18, 2001.

[4] C. Collberg, C. Thomborson, and D. Low, "A taxonomy of obfuscating transformations," *Tech. Rep. 148 Dep. Comput. Sci. Univ. Auckl. July*, no. 148, p. 36, 1997.

[5] C. Collberg, C. Thomborson, and D. Low, "Manufacturing Cheap, Resilient, and Stealthy Opaque Constructs," *ACM Symp. Princ. Program. Lang.*, no. Figure 1, pp. 184–196, 1998.

[6] C. S. Collberg, C. Thomborson, and S. Member, "Watermarking, Tamper-Proofing, and Obfuscation - Tools for Software Protection," *Computer (Long. Beach. Calif).*, vol. 28, no. 8, pp. 735–746, 2002.

[7] J. A. Muir, "A Tutorial on White-box AES," vol. 18, no. February, pp. 209–229, 2013.

[8] Y. Shi, W. Wei, and Z. He, "A Lightweight White-Box Symmetric Encryption Algorithm against Node Capture for WSNs," *Sensors*, vol. 15, no. 5, pp. 11928–11952, 2015.

[9] C. E. Shannon, "Communication Theory of Secrecy Systems," *Bell Syst. Tech. J.*, vol. 28, no. 4, pp. 656–715, 1949.

[10] H. E. Link and W. D. Neumann, “Clarifying Obfuscation: Improving the Security of White-Box Encoding.,” *IACR Cryptol. ePrint Arch.*, vol. 2004, p. 25, 2004.

[11] J. Bringer, H. Chabanne, and E. Dottax, “White box cryptography: Another attempt,” *located at, last Visit. Jul*, vol. 22, no. 2011, p. 14, 2006.

[12] Y. Xiao and X. Lai, “A Secure Implementation of White-Box AES,” pp. 1–6, 2009.

[13] J.-Y. P. J.-Y. Park, O. Y. O. Yi, and J.-S. C. J.-S. Choi, “Methods for practical whitebox cryptography,” *Inf. Commun. Technol. Converg. (ICTC), 2010 Int. Conf.*, pp. 474–479, 2010.

[14] J. Yoo, H. Jeong, and D. Won, “A method for secure and efficient block cipher using white-box cryptography,” *Proc. 6th Int. Conf. Ubiquitous Inf. Manag. Commun. - ICUIMC ’12*, p. 1, 2012.

[15] D. Klinec, “White-box attack resistant cryptography,” *Is.Muni.Cz*, 2013.

[16] D. Crockford, *JavaScript: The Good Parts*, vol. 44. 2008.

[17] B. Adida, A. Barth, and C. Jackson, “Rootkits for JavaScript environments,” *WOOT’09 Proc. 3rd USENIX Conf. Offensive Technol.*, p. 7, 2009.

[18] J. Xiao and Y. Zhou, “Generating large non-singular matrices over an arbitrary field with blocks of full rank,” pp. 1–6, 2002.

[19] Y. Han and I. Koren, “Fault Detection in AES Encoder and Decoder - Instructions.” [Online]. Available: http://www.ecs.umass.edu/ece/koren/FaultTolerantSystems/simulator/AES-L/round.jpg. [Accessed: 11-Feb-2016].

[20] B. Wyseur, “White-Box Cryptography: Hiding Keys in Software,” *MISC HS 5 Mag.*, pp. 58–64, 2012.

[21] “Modo de operação ECB da cifra por bloco.” [Online]. Available: https://upload.wikimedia.org/wikipedia/commons/thumb/e/e6/ECB_decryption.svg/1024px-ECB_decryption.svg.png. [Accessed: 17-Jun-2016].

[22] “Função de hash.” [Online]. Available: https://upload.wikimedia.org/wikipedia/commons/thumb/5/58/Hash_table_4_1_1_0_0_1_0_LL.svg/240px-Hash_table_4_1_1_0_0_1_0_LL.svg.png. [Accessed: 17-Jun-2016].

[23] R. W., “Função cálculo números Fibonacci,” 2011. [Online]. Available: http://stackoverflow.com/questions/7944239/generating-fibonacci-sequence. [Accessed: 17-Jun-2016].

[24] T. Hopp, “Função cálculo números primos,” 2012. [Online]. Available: http://stackoverflow.com/questions/11966520/how-to-find-prime-numbers-between-0-

100. [Accessed: 17-Jun-2016].