UNIVERSIDADE TECNOLÓGICA FEDERAL DO PARANÁ
DEPARTAMENTO ACADÊMICO DE INFORMÁTICA
CURSO DE ESPECIALIZAÇÃO EM TECNOLOGIA JAVA

DANIEL SOMEKAWA MARQUES

OFUSCAMENTO DE CÓDIGO PARA PROTEÇÃO DE PROGRAMAS JAVA CONTRA ENGENHARIA REVERSA

MONOGRAFIA DE ESPECIALIZAÇÃO

CURITIBA
2012

DANIEL SOMEKAWA MARQUES

OFUSCAMENTO DE CÓDIGO PARA PROTEÇÃO DE PROGRAMAS JAVA CONTRA ENGENHARIA REVERSA

Monografia de Especialização apresentada ao Departamento Acadêmico de Informática, da Universidade Tecnológica Federal do Paraná como requisito parcial para obtenção do título de Especialista em Tecnologia Java.

Orientador: Prof. Robson Ribeiro Linhares

CURITIBA
2012

Dedicatória

Á memória de meu pai, Ronaldo, por ser minha maior inspiração no estudo de Informática.

Á minha mãe, Emiko, e minha tia, Antonia pelo seu apoio constante.

Agradecimentos

Agradeço a todos os meus amigos e familiares pelo apoio e pela compreensão, aos professores do curso de Especialização em Tecnologia Java pela inspiração, e, especificamente, ao professor Robson, que me auxiliou sempre com muita atenção no desenvolvimento deste trabalho.

Resumo

Marques, Daniel S., Ofuscamento de código para proteção de programas java contra engenharia reversa – Especialização em Tecnologia Java, Universidade Tecnológica Federal do Paraná. Curitiba 2012.

Este trabalho visa apresentar e definir o ofuscamento de código como um meio de proteção à propriedade intelectual presente em programas Java, cobrindo suas técnicas e mostrando de que maneira cada uma delas funciona. Além disto, discute suas vantagens e desvantagens, o nível de defesa proporcionado por cada técnica contra um ataque de engenharia reversa, as soluções existentes no mercado, as consequências da aplicação do ofuscamento em um projeto de software, e, por fim, demonstra o conceito do ofuscamento de programas Java por meio de um algoritmo implementado usando as tecnologias apresentadas.

Palavras-chave: Ofuscamento. Transformação. Engenharia reversa. Proteção. Propriedade Intelectual.

Abstract

Marques, Daniel S., Code obfuscation for protection of Java programs against reverse engineering – Specialization in Java Technology, Federal Technological University of Parana. Curitiba 2012.

This paper aims to present and define code obfuscation as means of protecting intellectual property present on Java programs, covering it's techniques and showing how each of them work. Furthermore, discussesit's advantages and disadvantages, defense level provided by each technique against a reverse engineering attack, existent solutions in the market, the consequences of applying obfuscation on a software project, and, finally, demonstrates the concept of Java programs obfuscation through an algorithm implemented using presented technologies.

Keywords: Obfuscation. Transformation. Reverse engineering. Protection. Intellectual property.

Lista de Figuras

Sumário

1. Introdução

1.1 Estudo do domínio do problema

O ofuscamento de código é uma área específica do conhecimento que utiliza técnicas e ferramentas que visam atender às necessidades de proteção à propriedade intelectual presente no coração de qualquer aplicação de software, o código-fonte. Sendo o código-fonte uma rica fonte de conhecimentos, informações e detalhes cruciais ao sucesso de um produto digital, muitas pessoas mal-intencionadas, que podem incluir outros desenvolvedores, crackers ou mesmo companhias inteiras, tentarão encontrar meios que possibilitem a clara e precisa leitura da natureza textual de um programa de computador.

O mercado de software é um meio repleto de criatividade, conhecido por sua velocidade em disponibilizar soluções para os novos problemas que vão surgindo. É, também, o nicho de uma acirrada competição, que determina a qualidade dos produtos e que, sem dúvida, é fator decisivo no sucesso das grandes companhias desenvolvedoras de aplicações.

Em todas as formas de comércio, há pessoas que pretendem sempre seguir aquilo que é ético: crescer à custa de seu próprio esforço, competir com seus concorrentes da forma mais honesta possível, não lucrar de maneira desonesta, etc. Por outro lado, outras pessoas podem querer obter alguma vantagem sobre o trabalho alheio, roubando produtos desenvolvidos por outros fabricantes e vendendo como se fossem seus, quebrando a proteção criptográfica de arquivos de mídia ou, ainda, removendo uma checagem de licença contida em um jogo, por exemplo.

Em um mercado repleto de profissionais mal-intencionados, é necessário que haja métodos para que o desenvolvedor ético se defenda contra possíveis ataques à integridade do seu trabalho. Aqueles dispostos a realizar tais ataques verão nos segredos contidos em um programa uma excelente maneira de obter benefícios para si, seja na forma de lucro ou mesmo do uso de um produto sem que por ele tenham pago.

Se uma pessoa deseja desencorajar um ladrão a invadir sua casa, terá de estudar como o ladrão costuma agir e aplicar técnicas que irão fazer o ladrão pensar duas vezes

antes de invadir a casa novamente. É exatamente este o papel do ofuscamento de código: prover meios para que o desenvolvedor possa retardar ou desencorajar a ação de pessoas interessadas em ter acesso às minúcias de seus códigos-fonte. Como um meio de ocultar informações sensíveis, o ofuscamento se mostra uma interessante ferramenta que pode fazer os atacantes menos experientes desistirem rapidamente, diminuindo a influência da pirataria sobre o mercado de software e estimulando a legalidade da aquisição de produtos de software em geral.

1.2 Objetivo

O objetivo deste trabalho é apresentar uma visão de alto nível do ofuscamento de código como meio para proteção de software contra tentativas de acesso ao código-fonte, cobrindo as principais técnicas envolvidas, ferramentas e algoritmos pertencentes à área de estudo proposta, utilizando para isso a linguagem e plataforma Java.

Além disto, este trabalho aborda:

- realização de testes com algoritmos e ferramentas de ofuscamento existentes conjuntamente com ferramentas de engenharia reversa, com a posterior análise do resultado dos mesmos, a fim de se medir a eficiência das soluções existentes no mercado;
- implementação, na forma de um protótipo, de um pequeno algoritmo de ofuscamento, levando em conta as melhores práticas e priorizando o melhor balanço entre a proteção e o desempenho;
- discussão sobre os prós e contras, obtida por meio dos resultados do estudo sobre as tecnologias de ofuscamento existentes, que indicará a necessidade ou não do uso de tais técnicas em um projeto de software, e que consequências isto pode trazer para o projeto e ao próprio aplicativo.

1.3 Justificativa

Programas Java funcionam por meio de uma máquina virtual, que interpreta código-fonte compilado em formato *bytecode*, um estágio intermediário entre o código-fonte e o código nativo, executado pela máquina virtual sobre uma determinada plataforma. Se um

desenvolvedor Java deseja conferir um grau mínimo de proteção a seu projeto e não quiser publicá-lo como *open source*, deverá distribuir seus produtos em pacotes de classes em formato *class*, que identifica cada uma das classes em uma aplicação Java orientada a objetos.

Para garantir a interoperabilidade entre diferentes plataformas de hardware/software, O *bytecode* Java não foi originalmente projetado para dificultar qualquer tipo de engenharia reversa, afim de se tentar revelar o código-fonte por traz do código intermediário, pois sempre visou a independência de plataforma especialmente. Há muitos projetos *open source* que tiram vantagem disto, porém muitas aplicações comerciais escritas em Java estão sujeitas a intrusão por programadores e companhias mal intencionadas, e ante as características essenciais da plataforma Java, não é muito difícil para alguém disposto a quebrar a integridade de programas Java revertê-los em código legível. Em suma: o código Java, na sua forma natural, está exposto a uma série e riscos que empresas podem não estar dispostas a correr.

Aplicar técnicas de proteção, como o ofuscamento, sobre o código irá potencialmente atrasar e desencorajar o trabalho de profissionais mal intencionados, dando mais tempo ao desenvolvedor para lançar atualizações e minimizar falhas que podem facilitar a ação de crackers.

Sendo este um problema compartilhado entre as tecnologias baseadas em interpretação de código (como, além do Java, .Net[10]), característica que contrasta com a grande abrangência tida pela plataforma Java no mercado de aplicações e que nada mais é do que uma consequência natural da característica multiplataforma da mesma, é uma solução a se pensar primariamente ao invés da troca por outras tecnologias que podem não ter a mesma relação de custo-benefício como o Java.

Proteger o código fonte contra acessos não autorizados significa não apenas reforçar a segurança das aplicações no que se diz respeito à integridade da lógica de negócios, mas também se traduz em uma importante estratégia para a redução da pirataria de software, seja ela vinda do próprio usuário ou de um concorrente decidido a roubar os segredos contidos no mesmo e vendê-lo como se fosse de sua propriedade.

1.4 Escopo

Não é objetivo deste trabalho de monografia expor todos os detalhes com relação ao ofuscamento de código, com relação às ferramentas, algoritmos e técnicas para ofuscamento, mas descrever as tecnologias envolvidas apenas na sua essência.

Aqui são cobertas apenas as informações essenciais para a compreensão do problema proposto e do estudo realizado sobre o mesmo, consistindo nas soluções principais de cada uma das categorias mencionadas.

1.5 Organização do trabalho

O capítulo 2 descreve as tecnologias empregadas neste trabalho, de modo a clarificar a estrutura utilizada para o estudo do ofuscamento de código proposto neste trabalho.

O capítulo 3 aborda o processo de construção do algoritmo de ofuscamento na forma de protótipo proposto para auxílio nos estudos deste trabalho, assim como aplicações do mesmo em códigos reais para fins de avaliação.

O capítulo 4 aborda os estudos de caso com algoritmos e ferramentas de ofuscamento, onde são demonstrados o processo e também os resultados dos mesmos.

O capítulo 5 contém as conclusões tiradas a partir do estudo proposto. Nele consta uma discussão sobre os resultados, onde são apontados os pontos fortes e fracos das tecnologias estudadas, as dificuldades envolvidas, etc.

No capítulo 6 são enumeradas todas as referências consultadas para a realização deste estudo.

O capítulo 7 contém os anexos deste trabalho.

2. Estudo das tecnologias

2.1 Ofuscamento de código

2.1.1 Conceito

O propósito principal do ofuscamento é adicionar diferentes níveis de “confusão” a um determinado código. O código ofuscado é, portanto, extremamente difícil de se compreender e oculta detalhes cruciais de sua programação em grupos de instruções que parecem não fazer sentido, ainda que esteja completamente funcional ao ser compilado. Em suma, é um código inicialmente incompreensível, que exige tempo e recursos de quem quiser estudar seu funcionamento e, posteriormente, extrair ou modificar algo depois que este passou pelo processo de ofuscamento.

O princípio por trás do ofuscamento é justamente tornar o código confuso e difícil de compreender, assim, se alguém que não deveria conhecer este código tentar ter acesso a ele obtendo sua forma textual clara, não será fácil descobrir algo que seria trivial para o desenvolvedor, que conhece detalhadamente seu software e seu código-fonte.

Após ser ofuscado, um programa se comporta exatamente da mesma maneira como seu equivalente não ofuscado se comportaria. Não é objetivo do ofuscamento modificar a maneira como programas se comportam, mas alterar a sua aparência e eliminar qualquer evidência que identifique funções específicas que determinados trechos do código desempenham. É, portanto, uma medida de defesa que visa proteger os detalhes por trás do funcionamento e da estrutura do código, e o mais importante: a propriedade intelectual presente no mesmo. Vale lembrar que o ofuscamento é um meio de proteção versátil, que ajuda a barrar diversos tipos de adversários, que podem ser concorrentes interessados em utilizar a tecnologia alheia sem ter os devidos direitos de uso sobre ela, clientes que não desejam investir seu dinheiro em um produto mas querem ter acesso a suas funções, ou mesmo pessoas que enxergam uma potencial oportunidade de lucro em aplicações que não passaram por qualquer processo de proteção.

É importante mencionar que aplicar técnicas de ofuscamento não significa eliminar qualquer possibilidade de invasão em um dado código, mas dará ao desenvolvedor a oportunidade de vencer os crackers pela exaustão, retardando seu sucesso em sua tentativa

de quebra de segurança, pois todos sabemos que impor barreiras de proteção realmente intransponíveis em qualquer sistema digital é algo alheio à realidade.

2.1.2 Aplicações

Ofuscamento pode ser adotado em uma variedade de situações.

Aplicações comerciais, como jogos, editores de vídeo e conversores de áudio possuem geralmente um sistema de checagem de licença, que identifica se o usuário que as está tentando executar pagou por elas previamente e tem de fato o direito de as utilizar de acordo com o contrato firmado entre a empresa desenvolvedora e o comprador. Se a checagem falhar, significa que o usuário não pagou a licença de uso do software, e, portanto, este iniciará alguma rotina de bloqueio, que pode encerrar o programa, fazer com que determinadas opções fiquem restritas ou mesmo impedir que o usuário efetivamente execute o programa.

Neste cenário fictício, o maior adversário do desenvolvedor seria justamente o seu próprio cliente, que poderia tentar ter acesso ao código, analisá-lo e enfim encontrar a parte do mesmo que deve ser modificada para remover a checagem de licença, por fim recompilando a aplicação para seu uso particular. Assim, o desenvolvedor pode achar interessante utilizar o ofuscamento para alterar a aparência da mecânica do processo de checagem de licença para que esta revele menos informação, e pareça menos óbvia, fazendo com que o invasor gaste muito mais tempo para tentar alterar o código nesta região específica.

Também podemos exemplificar o seu uso no caso específico de uma determinada aplicação reprodutora de músicas. Hoje temos muitos serviços baseados na internet que vendem não álbuns inteiros – discos contendo músicas, encartes, lista de letras, etc – mas músicas, compradas separadamente, uma estratégia que diminui os gastos de quem deseja pagar apenas pelas músicas que aprecia, ao invés de investir em um disco completo. Esta aplicação de reprodução pode conter um sistema que verifique se a música que o usuário quer reproduzir foi previamente comprada utilizando um determinado serviço de vendas. De maneira similar, o desenvolvedor pode aplicar o ofuscamento para impor dificuldades ao usuário que deseja utilizar o reprodutor, mas não quer investir seu dinheiro em músicas, que poderia facilmente encontrar pesquisando em serviços de busca.

Outra área conhecida pelo uso extensivo de técnicas de ofuscamento é a programação de vírus de computadores. Dentre os procedimentos adotados pelos softwares que se propõe a identificar e eliminar vírus, softwares conhecidos popularmente como anti-vírus, há a análise de assinatura. Assinaturas podem ser algoritmos, sequências numéricas geradas por *hashing*, padrões de funcionamento ou qualquer outra característica particular que possa identificar um vírus específico dentre uma família de vírus conhecidos. Utilizando este método, o anti-vírus vasculha o conteúdo binário de arquivos, comparando a assinatura dos mesmos com seu banco de dados e assim identificando e diferenciando arquivos infectados dos não infectados.

Para se proteger contra detecção, vírus implementam muitas vezes técnicas de ofuscamento para ocultar a maneira como eles se comportam, fazendo modificações estruturais, alterando fluxos de controle e adicionando instruções confusas à sua programação modificando totalmente sua assinatura perante os anti-vírus, que terão dificuldades muito maiores para detectá-los. Determinadas classes de vírus podem fazer isto em tempo de execução, como os vírus metamórficos, que têm a capacidade de se reescrever e se recompilar em novas variações, cujo algoritmo essencialmente é o mesmo, mas cuja estrutura é completamente diferente do original[4].

2.1.3 Ferramentas de ofuscamento

Há diversas soluções para ofuscamento de programas Java no mercado, algumas pagas, porém grande parte delas de código aberto. Entre as ferramentas existentes, é bastante comum a combinação de funções de ofuscamento com tarefas de *shrinking* e compactação, que otimizam e tornam o código menor e mais veloz em execução. É possível ofuscar códigos tanto a partir do código-fonte quanto a partir do mesmo código em formato *bytecode*, porém a maioria das soluções implementa o ofuscamento sobre o *bytecode* apenas. Abaixo algumas das ferramentas existentes:

Open-source/Free:

- ProGuard (http://proguard.sourceforge.net) – ferramenta de ofuscamento mais conhecida do mercado. Escrita em Java e de código aberto, executa operações de ofuscamento e, além disso, também suporta rotinas de *shrinking* e pré-verificação para compactação e otimização de *bytecode*.
  - Pode ser usado via GUI ou linha de comando;

- Trabalha sobre arquivos jar e *class*.

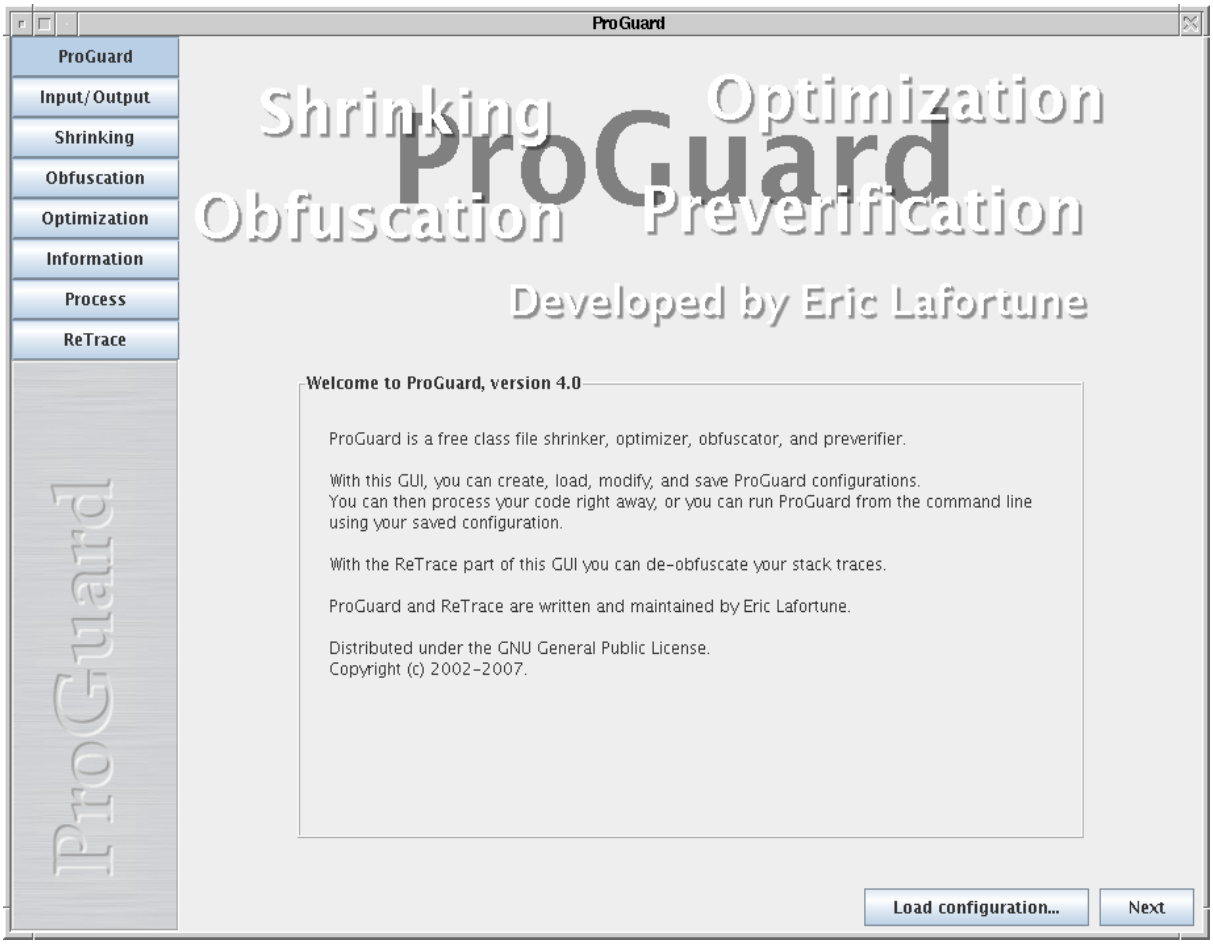

Figura 1 - Captura de tela do ProGuard.

- Jarg (http://jarg.sourceforge.net/) - *Shrinker* e compactador de códigos, mas também ofuscador, é uma ferramenta escrita em Java e de código aberto, mais simples que as demais.
  - Usado via linha de comando;
  - Trabalha sobre arquivos jar e *class*.

- JODE (http://jode.sourceforge.net/) - Otimizador e também decompilador, executa operações de ofuscamento simples, além de modificar classes priorizando o desempenho. Suporta execução de tarefas através de arquivos *script*.
  - Pode ser usado via GUI ou linha de comando;
  - Trabalha sobre arquivos jar e *class*.

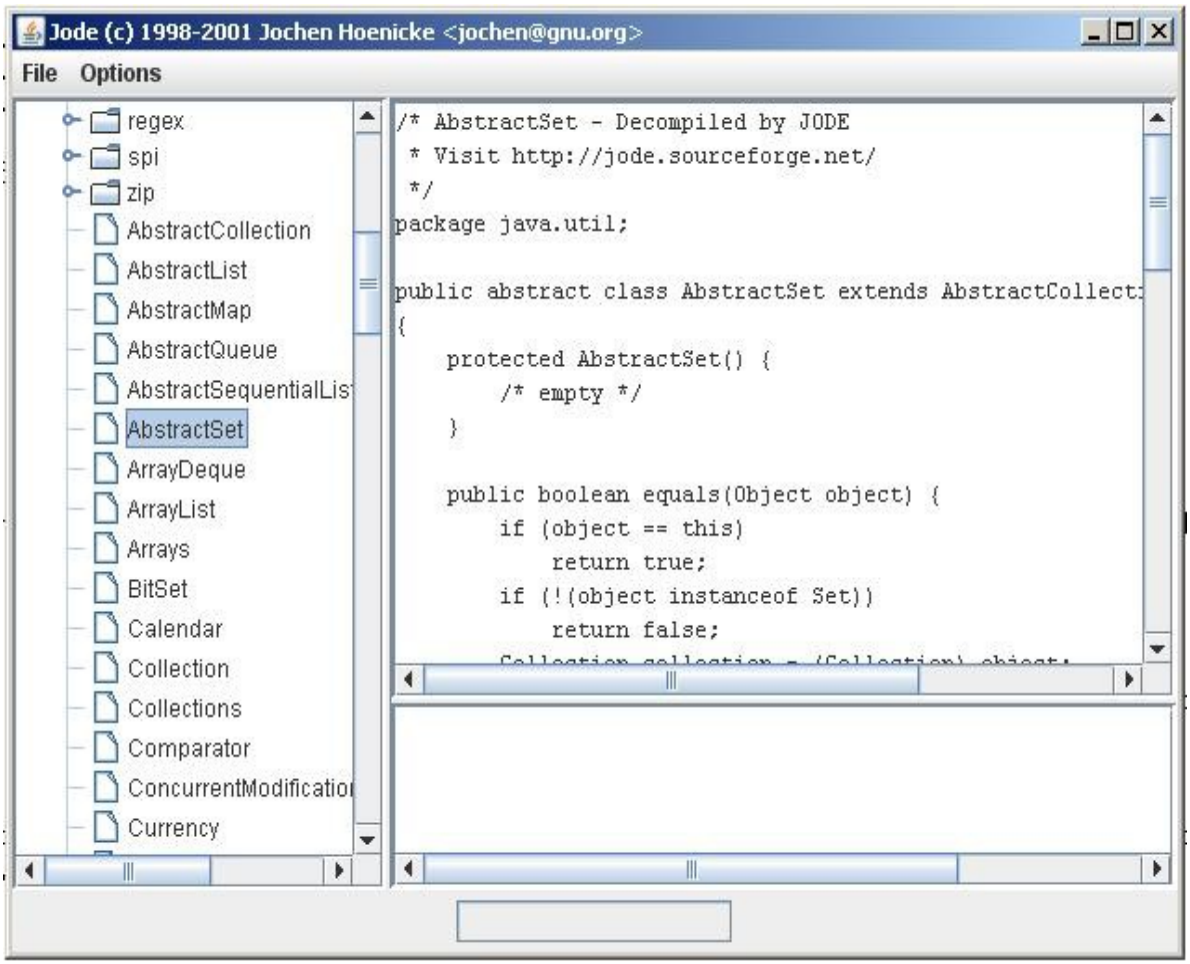

Figura 2 - Captura de tela do JODE.

- VasObfuLite (http://vasile.chez.com/obfu/VasObfuLite.html) – Simples aplicação online que faz transformações léxicas diretamente em um código-fonte.
  - Utilizado via *browser*;
  - Trabalha sobre texto plano.

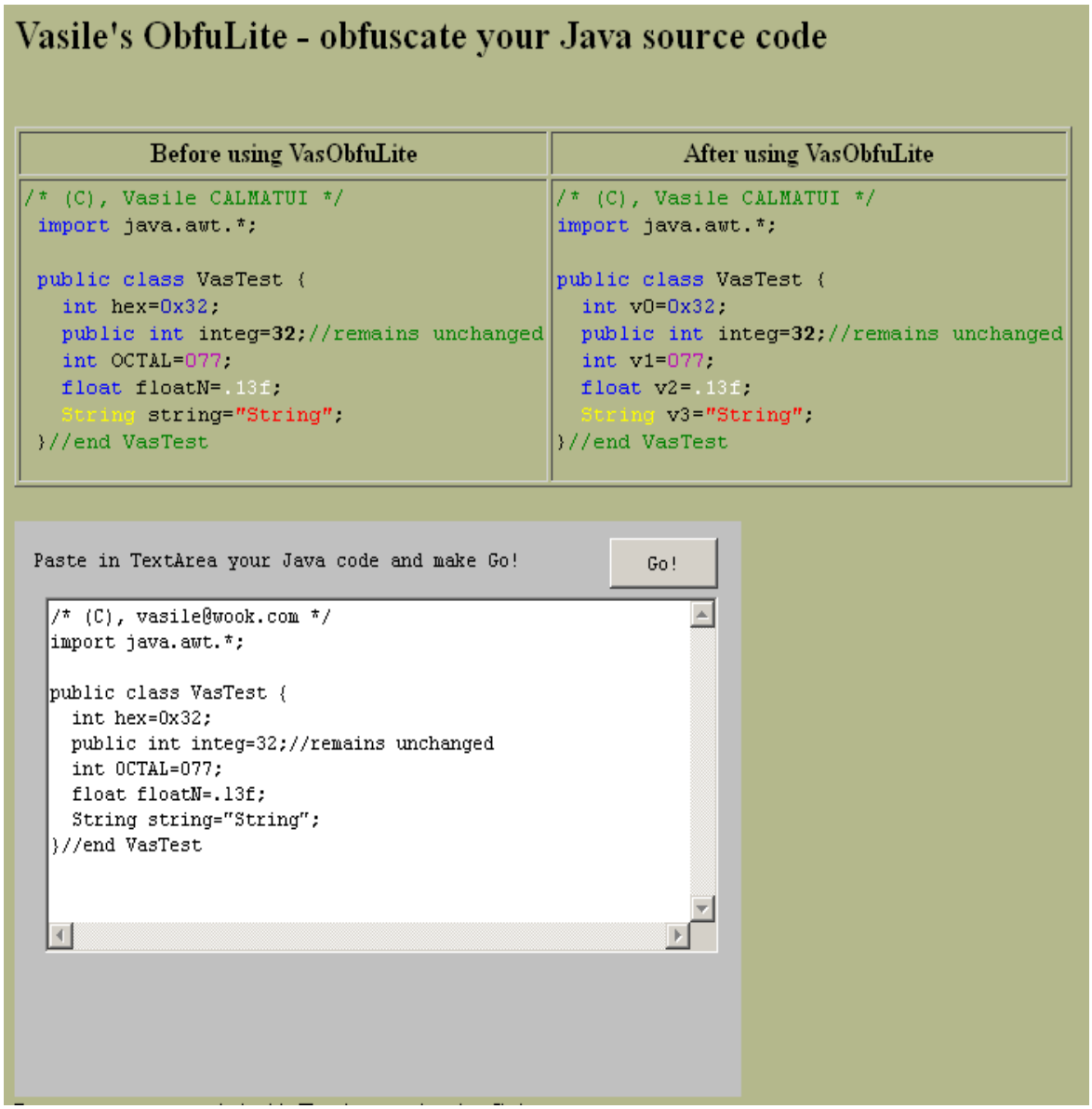

Figura 3 - Captura de tela do VasObfuLite.

- SandMark (http://sandmark.cs.arizona.edu) – Aplicação voltada para o estudo de técnicas de proteção de software, agregando diversas implementações de algoritmos de ofuscamento, *watermarking*, entre outros.
  - Usado via GUI;
  - Trabalha sobre arquivos jar.

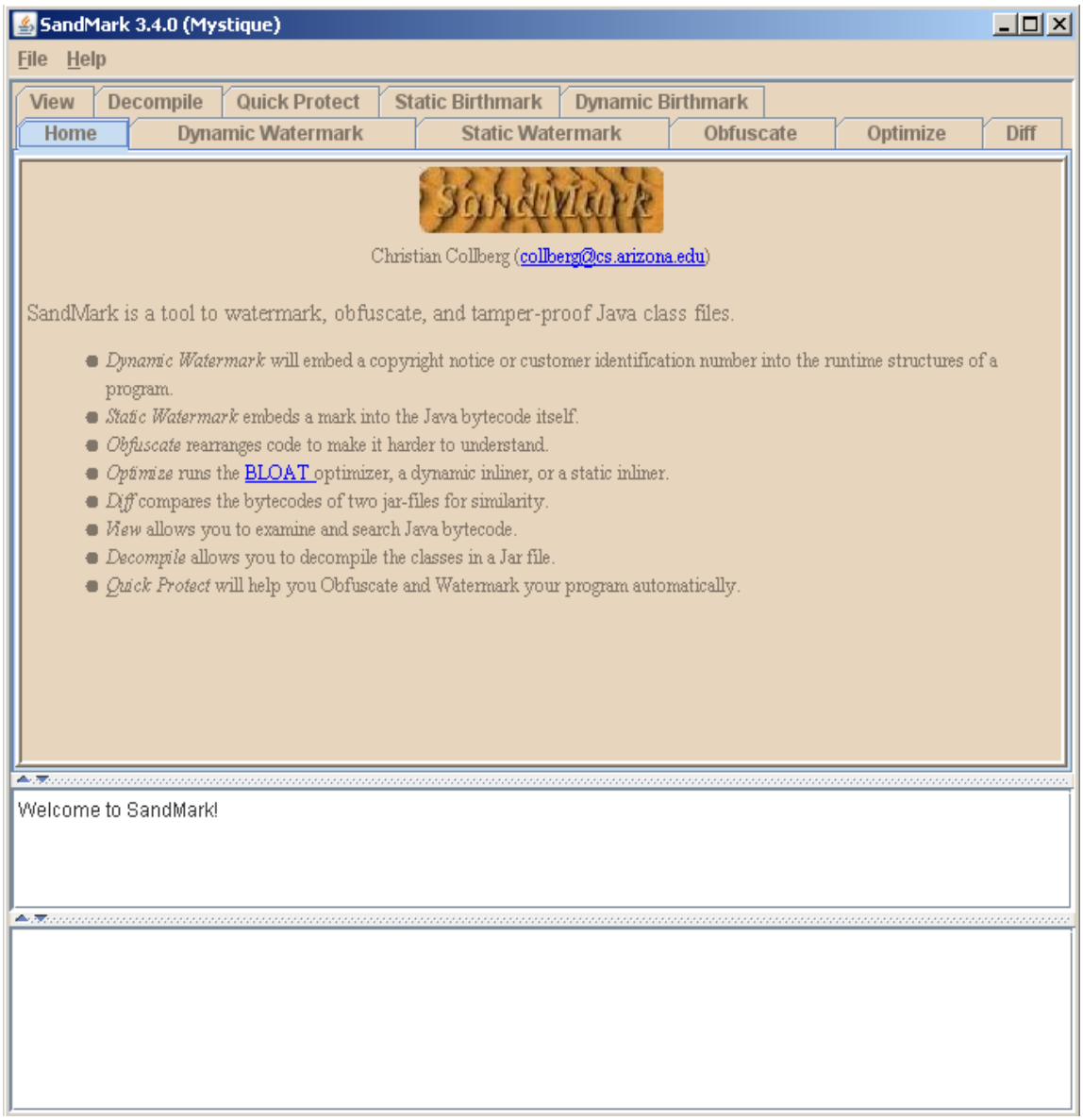

Figura 4 - Captura de tela do Sandmark.

- Yguard (http://www.yworks.com/en/products_yguard_about.htm) – Versão atualizada do ofuscador RetroGuard, pode executar transformações léxicas alterando nomes de classes, pacotes, métodos, etc, além de compactar e otimizar o *bytecode* através de rotinas de *shrinking*.
  - Utilizado juntamente ao *framework* Ant;
  - Trabalha com arquivos XML;

Pagas:

- KlassMaster (http://www.zelix.com/klassmaster/features.html) – Ferramenta comercial que aborda técnicas avançadas de ofuscamento, incluindo encriptação de strings e transformações de fluxo de controle, além de também otimizar o *bytecode* como seus concorrentes.
  - Utilizado via GUI;

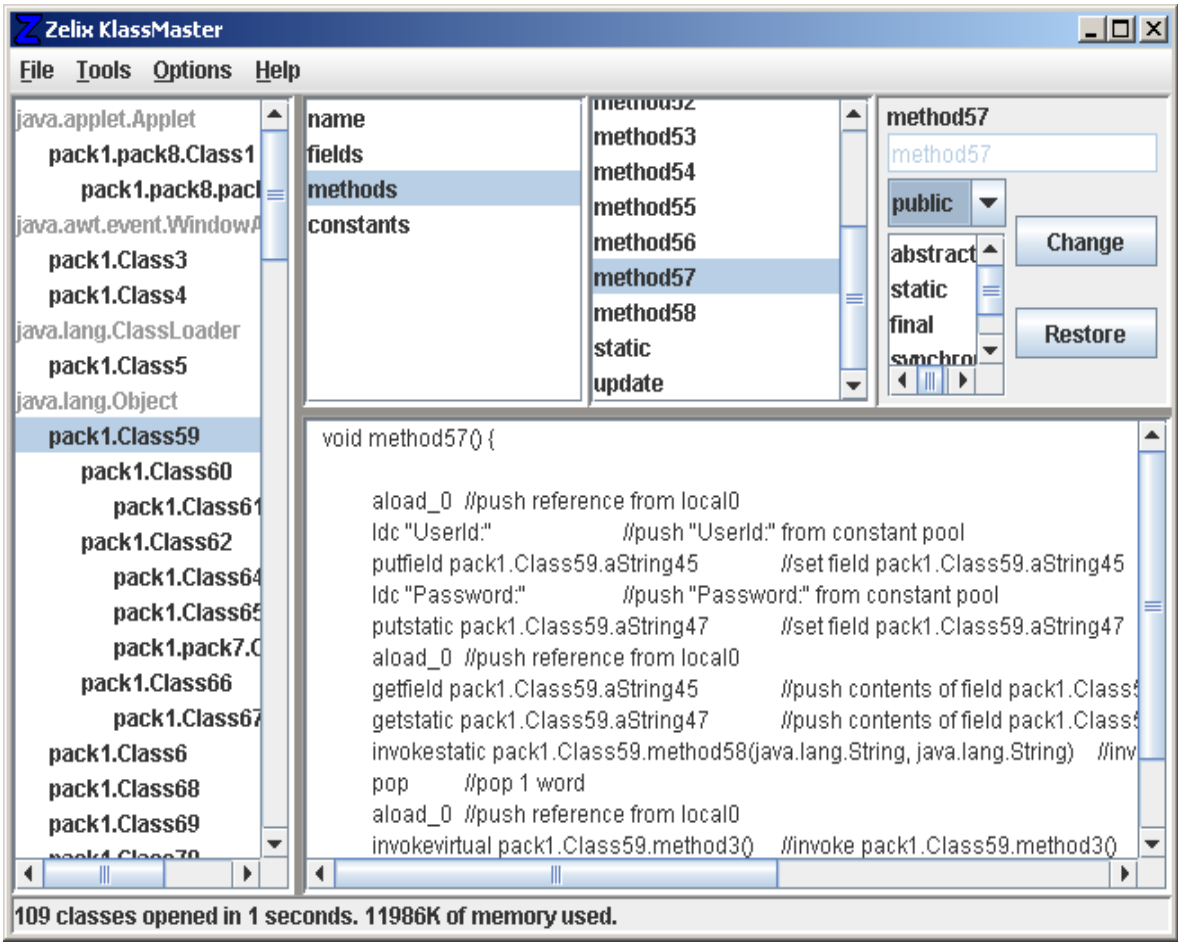

Figura 5 - Captura de tela do KlassMaster.

- DashO (http://www.preemptive.com/products/dasho/overview) – Implementa técnicas avançadas de ofuscamento, além de detectar tentativas de adulteração através de rotinas de *tamper checking*.
  - Utilizado como *plugin* do Eclipse, Ant *task*, via linha de comando ou GUI própria;

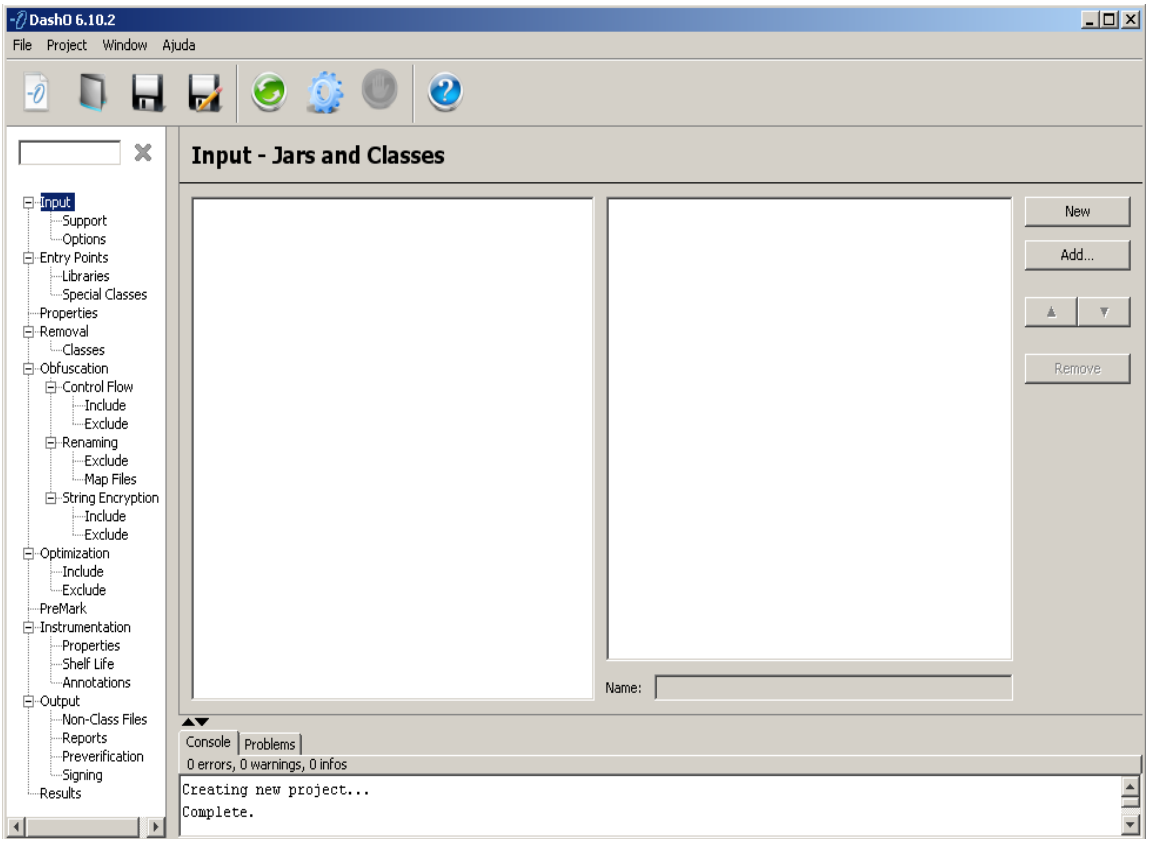

Figura 6 - Tela inicial da versão de avaliação do DashO.

- Jshrink (http://www.e-t.com/jshrink.html) – Ofuscador e otimizador, implementa técnicas simples de ofuscamento e compactação de código. Suporta *scripting*.
  - Utilizado via GUI ou linha de comando;
  - Trabalha sobre arquivos jar e *class*.

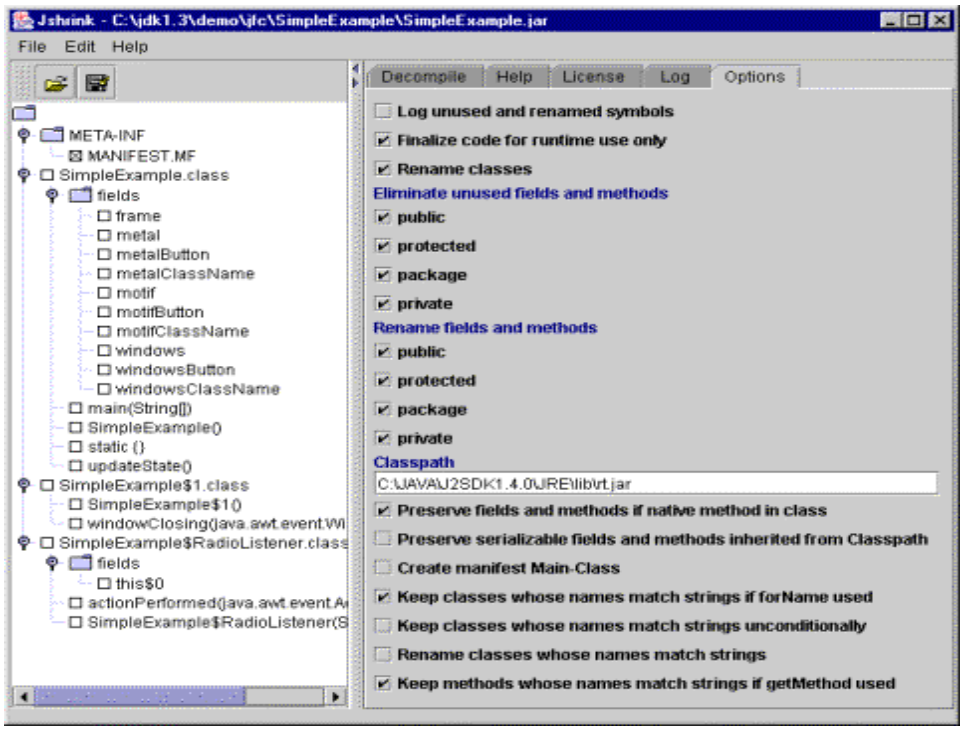

Figura 7 - Captura de tela do Jshrink.

- SmokeScreen (http://www.leesw.com/smokescreen/index.html) – Dentre suas funções, aborda encriptação de strings, transformações de fluxo de controle e ofuscamento incremental, além de também implementar rotinas de *shrinking*.
  - Pode ser usado via GUI ou linha de comando;
  - Trabalha sobre arquivos jar e *class*.

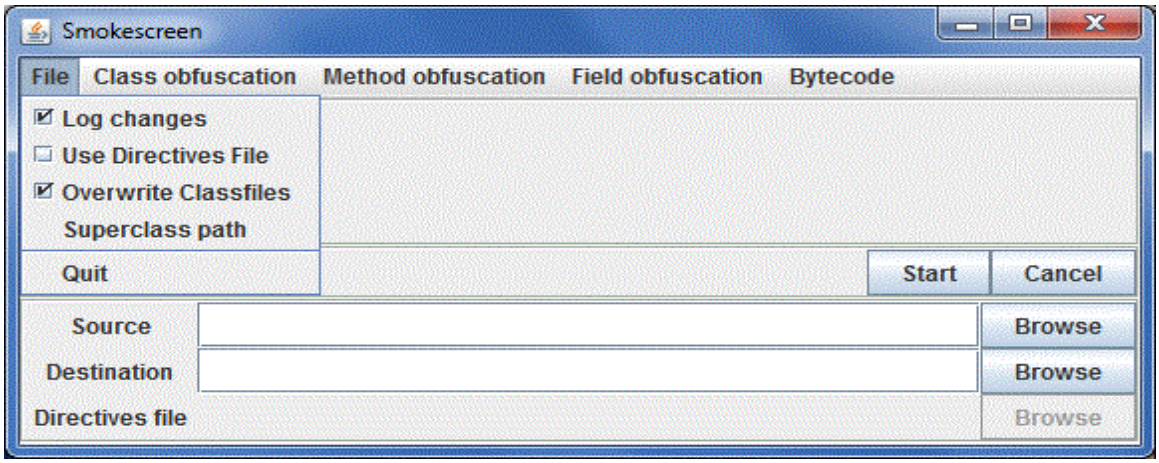

Figura 8 - Captura de tela do Smokescreen.

- Allatori (http://www.allatori.com/features.html) – Ferramenta de proteção bastante versátil, que implementa diversas classes de ofuscamento, além de oferecer funções de *watermarking* e ofuscamento para as plataformas Android e JME.
  - Pode ser usado via linha de comando, ou como Ant *task*;
  - Trabalha com arquivos XML.

- JCloak (http://www.force5.com/JCloak/ProductJCloak.html) – Ofuscador totalmente escrito em Java e voltado para *applets*, implementa diversas técnicas de ofuscamento básicas e avançadas.
  - Pode ser usado via GUI ou linha de comando;

- Java Obfuscator (http://www.semdesigns.com/Products/Obfuscators/JavaObfuscator.html) – Ferramenta especializada em ofuscamento, implementa transformações variadas.
  - Pode ser usado via GUI;
  - Trabalha com arquivos java, em texto plano;

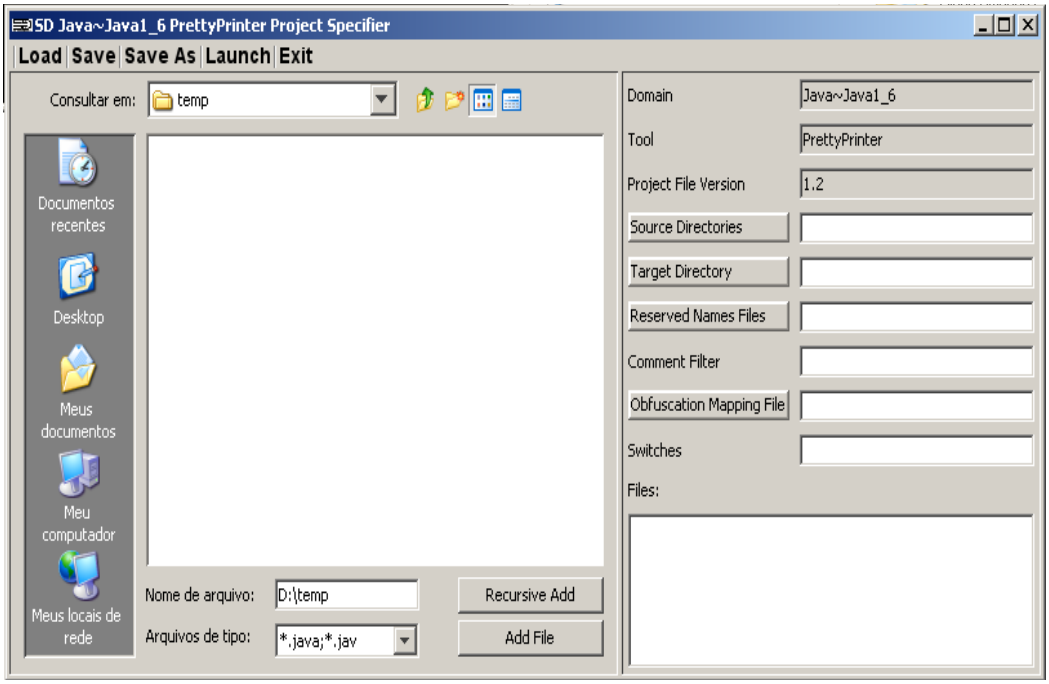

Figura 9 - Captura de tela do Java Obfuscator.

- Jobfuscate (http://www.duckware.com/jobfuscate/index.html) – Simples ferramenta de ofuscamento que implementa algumas técnicas de transformação, além de ser capaz de se integrar a *scripts* e ambientes de desenvolvimento.
  - Utilizado via linha de comando;
  - Trabalha com arquivos jar;

- GuardIt (http://www.arxan.com/software-protection-products/java-GuardIt/index.php) - Ferramenta especializada em proteção de código Java através de ofuscamento.

2.1.4 Modalidades de ofuscamento

O processo de ofuscar código consiste na execução de operações denominadas transformações. Cada classe de transformação irá alterar diferentes aspectos de um programa, de sua organização estrutural a seus fluxos de controle.

Abaixo os principais tipos de transformações[1 2 3]:

- Transformação léxica – substitui nomes de variáveis, métodos, classes e pacotes por outros identificadores que revelam menos informação.
- Transformação de abstração/*layout*- altera o programa modificando sua estrutura de

classes, pacotes, funções, etc.

- Transformação de controle - modifica as instruções de controle de fluxo afim de ocultar os diferentes caminhos tomados pelas instruções do programa durante sua execução.
- Transformação de dados - substitui estruturas de dados existentes por outras que revelam menos informação.
- Transformação dinâmica - através de um algoritmo de ofuscamento embutido dentro do próprio código, executa alterações em tempo de execução.
- Transformação preventiva – atrapalha o funcionamento de decompiladores, através da inserção de instruções específicas.

2.1.4.1 Transformações léxicas

Geralmente a escolha de identificadores para as partes integrantes de um programa depende de seu papel dentro da lógica proposta pelo programa. Se um método deve, por exemplo, verificar a integridade de um arquivo, este provavelmente levará o rótulo de “verificarArquivo” ou “checarIntegridade”, identificadores que designam em poucas palavras o que o método deve fazer. Dentro da comunidade de desenvolvedores Java costuma-se dar grande importância às convenções de escrita, que ditam as melhores maneiras de se escrever códigos, afim de maximizar a legibilidade e velocidade de compreensão dos mesmos. Padrões são recomendados para elaboração de nomes de variáveis, métodos, classes e pacotes.

O papel das transformações léxicas é justamente eliminar o significado presente nos identificadores – de classes, métodos, atributos, variáveis, pacotes, etc - acrescentando em seu lugar outros rótulos que podem ser aplicados aleatoriamente ou baseado em uma tabela de rótulos (funcionalidade disponível em algumas ferramentas).

Abaixo um exemplo, um simples programa que identifica a posição de uma *string* em uma coleção e a remove quando encontrada:

```
package test;
import java.util.*;
class Demo {
   private Vector buffer = new Vector();
   int getStringPos(String string) {
      for(int counter=0; counter < buffer.size(); counter++) {
         String curString = (String)buffer.elementAt(counter);
         if (curString.equals(string)) {
            buffer.remove(counter);
            return counter;
         }
      }
      return -1;
   }
}
```

Código original, antes da transformação.

```
package a;
import java.util.Vector;
class a {
    private Vector a = new Vector();
    int a(String s) {
        for(int i = 0; i < a.size(); i++) {
            String s1 = (String)a.elementAt(i);
            if(s1.equals(s)) {
                a.remove(i);
                return i;
            }
        }
        return -1;
    }
}
```

O mesmo código, após compilação, transformação e então decompilação.

Abaixo, um outro exemplo, um método de cálculo baseado em matrizes:

```
int sumOfElements(int[][] matrix, int rowsCount, int columnsCount) {
      int sum = 0;
      for (int row = 0; row < rowsCount; row++)
          for (int column = 0; column < columnsCount; column++)
              sum += matrix[row][column];
      return sum;
```

```
    }
```

Original, antes da transformação.

```
int a(int a[][], int a, int a) {
        int i = 0;
        for(int j = 0; j < a; j++) {
            for(int k = 0; k < a; k++)
                i += a[j][k];
        }
        return i;
    }
```

O mesmo código, após compilação, transformação e então decompilação.

Note como o significado dos identificadores dentro da lógica do programa é totalmente anulado quando este é transformado lexicamente. Como estas transformações modificam apenas nomes, sem alterar estrutura, abstrações ou fluxos de controle, não há impacto no desempenho geral da aplicação e isto pode até mesmo contribuir para reduzir o tamanho em bytes dos executáveis java e arquivos *class*.

2.1.4.2 Transformações de abstração/*layout*

Abstração é o conceito chave da linguagem Java. Programas Java implementam a abstração para separar os diferentes níveis de detalhamento em sua estrutura, o que se torna evidente na maneira como classes são construídas, assim como na forma como as hierarquias de classes são montadas. Definições de métodos e atributos, construção de tipos e subtipos, são algumas maneiras de inserir doses de abstração a um programa orientado a objetos.

O papel das transformações de abstração ou *layout* é justamente provocar quebras nas abstrações adotadas, fundindo o que deve ser separado, e separando o que deve ser fundido, uma tática útil não para camuflar os fluxos de controle ou o retirar o sentido dos identificadores utilizados, mas para camuflar a organização e estrutura da lógica do programa.

Abaixo um exemplo:

```
public class Any {
        public static void main(String args[]) {
                foo(10);
                bar(3, 3);
        }
        private int foo(int x) {
                System.out.println(x*5);
        }
        private void bar(int x, int z) {
                if(x == z)
                        System.out.println("Same");
        }
}
```

Uma classe qualquer que possui dois métodos com funções bem definidas.

```
public class Any {
        public static void main(String args[]) {
                foobar(10, 0, true);
                foobar(3, 3, false);
        }
        private foobar(int x, int z, boolean flag) {
                if(flag)
                        System.out.println(x*5);
                else {
                        if(x == z)
                        System.out.println("Same");
                }
        }
}
```

A mesma classe, após sofrer uma transformação de abstração ou *layout*.

Após transformada, a classe teve seus dois métodos foo e bar fundidos em um só, foobar. No método principal, pode-se ver que há duas chamadas ao mesmo método, foobar. O grande trunfo é que, na verdade, temos dois blocos de código distintos camuflados em um único método. Pode-se dizer, de forma imaginária, que temos dois métodos em um.

Note que os fluxos de controle logicamente continuam os mesmos, pois não há desvios para outras regiões do programa, apenas a aparência da estrutura de métodos foi alterada. Como parte da abstração foi removida, o código está agora em um nível de compreensão menor e menos intuitivo, impondo dificuldades ao cracker em sua tarefa de

desvendar a lógica de negócios contida no mesmo.

Abaixo um outro exemplo:

```
class C {
        int f;
        public C(int x) {
                f=x;
        }
        int m(int x) {
                return (f+=x);
        }
}
class A extends C {
        int f;
        public A(int x) {
                super(x);
                f=x+1;
        }
        public int m(int x) {
                return (f-=x);
        }
}
class B extends C {
        public B(int x) {
                super(x);
        }
}
class D extends A {
        public D(int x) {
                super(x);
        }
}
```

Antes da transformação.

```
class C {
        public String kind;
        int f;
        public C() {}
        public C(int x) {
                f=x;
                kind="C";
```

```
        }
        int m(int x) {
                if(kind.equals("C"))
                        return (f+=x);
                else
                        return A_m(x);
        }
        int A_f;
        public C(int x, int dummy) {
                this(x);
                kind="A";
                A_f=x+1;
        }
        int A_m(int x) {
                return (A_f-=x);
        }
}
class B extends C {
        public B(int x) {
                super(x);
        }
}
class D extends C {
        public D(int x) {
                super(x, 42);
        }
}
```

Após a transformação.

Acima temos um outro exemplo de quebra de abstração, onde as classes A e C, pertencentes à mesma hierarquia são fundidas em uma só classe, que contém o corpo de ambas. Note como a visão de alto nível em relação às duas classes fundidas não existe mais, e o cracker terá o trabalho de reconstruir a abstração removida para melhor compreender esta estrutura de classes.

2.1.4.3 Transformações de controle

Os fluxos de controle revelam importantes detalhes sobre o funcionamento dos programas, e a lógica de negócios é essencialmente fundamentada nos mesmos. Por definição, fluxos de controle são os diferentes caminhos que a execução de um programa

pode tomar, e a escolha destes caminhos é controlada por instruções específicas, notadamente as de natureza condicional, que tem a capacidade de desviar o programa de um bloco de código para outro, saltando entre diferentes escopos, dependendo das especificações do projeto.

Pode-se dizer que os fluxos de controle de um programa são parte de sua identidade funcional, e sabendo-se que direções o programa pode tomar em tempo de execução, pode-se conhecer muito sobre ele. O estudo dos fluxos de controle pode ajudar a revelar regiões específicas de um programa, por exemplo a região exata onde é feita a checagem de uma licença, ou onde é feito o bloqueio de uma funcionalidade, ou ainda onde uma informação sensível é descriptografada para ser utilizada na sua forma normal. Por isso temos transformações especiais a serem aplicadas neste aspecto dos programas.

Abaixo um exemplo:

```
package test;
import java.util.*;
class Demo {
   private Vector buffer = new Vector();
   int getStringPos(String string) {
      for(int counter=0; counter < buffer.size(); counter++) {
         String curString = (String)buffer.elementAt(counter);
         if (curString.equals(string)) {
            buffer.remove(counter);
            return counter;
         }
      }
      return -1;
   }
}
```

Código original, antes da transformação.

```
package a;
import java.util.Vector;
class a {
    private Vector a;
    public static int b;
    a() {
```

```
        a = new Vector();
    }
    int a(String s) {
        int i;
        int j;
        j = b;
        i = 0;
        if(j == 0) goto _L2; else goto _L1
_L1:
        String s1 = (String)a.elementAt(i);
        s1.equals(s);
        if(j != 0) goto _L4; else goto _L3
_L3:
        JVM INSTR ifeq 48;
           goto _L5 _L6
_L5:
        break MISSING_BLOCK_LABEL_37;
_L6:
        continue;
        a.remove(i);
        i;
_L4:
        return;
_L2:
        if(i >= a.size())
            return -1;
        if(true) goto _L1; else goto _L7
_L7:
    }
}
```

O mesmo código, compilado, e depois sofrendo transformações léxica e de controle, e então decompilado.

A seguir apresenta-se outro exemplo, um método de cálculo baseado em matrizes:

```
int sumOfElements(int[][] matrix, int rowsCount, int columnsCount) {
        int sum = 0;
        for (int row = 0; row < rowsCount; row++)
            for (int column = 0; column < columnsCount; column++)
                sum += matrix[row][column];
        return sum;
    }
```

Original, antes da transformação.

```
int a(int a[][], int a, int a) {
        int i = 0;
        int j = 0;
          goto _L1
_L6:
        int k = 0;
          goto _L2
_L4:
        i += a[j][k];
        ++k;
_L2:
        a;
        JVM INSTR icmplt 17;
           goto _L3 _L4
_L3:
        ++j;
_L1:
        a;
        JVM INSTR icmplt 10;
           goto _L5 _L6
_L5:
        return i;
}
```

O mesmo código, compilado, e depois sofrendo transformações léxica e de controle, e então decompilado.

Note como a inclusão de instruções goto e seus blocos, apenas permitidas no formato *bytecode* do código Java, cria diversos desvios e conduz o fluxo de controle do programa de forma extremamente confusa. A versão transformada visualmente é muito diferente do código original, em parte devido à incapacidade dos decompiladores de converter as instruções goto de volta para a instrução original for, e portanto demandará um esforço adicional por parte do cracker ao tentar revelar o fluxo de controle originalmente presente. Instruções do tipo MISSING_BLOCK_LABEL_ indicam blocos de código não reconhecidos pelos decompiladores mais antigos, como o Jad. Nota-se também que algumas instruções de montagem, como icmplt e ifeq, são trazidas erroneamente pelo decompilador.

Além disso, também é possível causar certos problemas ao cracker e suas ferramentas através da inclusão de blocos de código sem relevância, que são ou não executados dentro do fluxo de controle do programa, o que pode ajudar a reduzir a precisão

e aumentar o número de interpretações em uma análise por parte do cracker. Isso muitas vezes é feito juntamente à técnica de inserção de predicados opacos. Predicado opaco é qualquer expressão cujo valor é conhecido pelo desenvolvedor, mas difícil de ser revelado em uma análise feita por homem ou máquina. Este valor pode ser usado para desviar o fluxo de controle para regiões contendo estes blocos de código irrelevantes, conhecidos por *dead code blocks*, e afastar o cracker da lógica verdadeira do programa.

2.1.4.4 Transformações de dados

Em um programa, diversos tipos de dados são manipulados e estes servem como base para execução de diversos tipos de operações matemáticas e lógicas. Muitas vezes estes dados guardam informações cruciais sobre o programa, informações de grande utilidade ao cracker, e que podem ser facilmente interceptadas através de *debugging*. No contexto da proteção à propriedade intelectual contida em programas de computador, é importante que estes dados permaneçam a maior parte do tempo codificados, e decodificados apenas quando seu uso em formato natural for necessário. Alternativamente pode-se adaptar a lógica de negócios para tratamento dos dados somente em sua forma codificada, assim em nenhuma oportunidade estarão sujeitos a detecção sem que antes o cracker precise perder tempo decifrando-os em busca de seu real significado.

Um bom exemplo é a técnica de encriptação de strings, que é bastante utilizada pelas ferramentas comerciais de ofuscamento. Abaixo um exemplo:

```
private void checkLicense() throws Exception {
        if (!isLicenseValid())
            throw new Exception("Invalid License.");
        else
            return;
    }
```

Código original.

```
private void b() throws Exception {
        if(!a())
            throw new Exception(a.a("\\z`t}}v5Q}}pwg\177{"));
        else
            return;
    }
```

Após compilação, transformação léxica, aplicação da técnica de encriptação de strings, e então decompilação.

É também importante que as transformações de dados estejam acompanhadas de outras modalidades de transformações, já que para o caso de dados estarem codificados e forem decodificados para uso, a chave para a decodificação não deve estar em evidência, e para isso são úteis as outras classes de transformações, como as léxicas e as de controle.

2.1.4.5 Transformações dinâmicas

Diferente das demais transformações, cujas modificações são essencialmente estáticas e ocorrem anteriormente (quando o código-alvo ainda está em formato de texto plano) ou posteriormente (ofuscamento de *bytecodes*) ao tempo de compilação e antes do tempo de execução, estas modificam o programa durante seu funcionamento, por isso chamadas de dinâmicas.

Um algoritmo que transforma dinamicamente um programa pode ser uma implementação de qualquer classe de algoritmos de transformação léxica, de controle, de abstração, de dados, ou de qualquer outro tipo, sendo que o objetivo é que o código do programa tenha sempre uma aparência diferente, ainda que desempenhe sempre a mesma função.

Além de uma potencial arma contra ferramentas de decompilação, também é uma útil defesa contra debugging e contra análises visuais por parte do cracker, porém é a classe de transformações que mais prejudica o desempenho geral da aplicação, devido justamente à sua natureza dinâmica.

Programas Java infelizmente não são adequados para a aplicação de transformações dinâmicas, devido às restrições impostas pela JVM com relação ao acesso interno ao *bytecode*. Esta classe de transformações exige que a linguagem de programação escolhida permita que um programa escrito nesta linguagem possa modificar a si mesmo em tempo de execução, como a linguagem ActionScript[6]. A máquina virtual Java permite o carregamento dinâmico de classes através da técnica de *class loading*[7], mas não permite que uma classe possa modificar a si mesma sem que seja antes recarregada em memória. Portanto

utilizando-se apenas a tecnologia Java não é possível implementar programas que transformam-se dinamicamente.

2.1.4.6 Transformações preventivas

O funcionamento de ferramentas de decompilação se baseia na tradução do código de máquina para texto, no qual o código binário é convertido para o código-fonte correspondente, legível por humanos. Transformações preventivas têm seu alvo exatamente em tais ferramentas.

Transformações preventivas podem ser usadas para causar falhas de interpretação na leitura feita pelos decompiladores, induzindo-lhes à geração de resultados incorretos. Com isso, é possível fazer com que decompiladores sejam incapazes de gerar código-fonte compilável a partir de um *bytecode* ofuscado. Isso pode ser atingido de diversas maneiras[5]:

- acrescentando instruções permitidas no formato *bytecode*, mas proibidas no formato de código-fonte Java, como a instrução goto;
- substituindo identificadores por palavras reservadas, como nomes de tipos primitivos ou especificadores de acesso, ou símbolos restritos;
- dando-se o mesmo nome a classes aninhadas;

Há também algumas implementações particulares de transformações preventivas, como o algoritmo HoseMocha[2], que acrescenta instruções extras após instruções *return*, causando *crash* no decompilador Mocha.

Vale lembrar que estas alterações preservam a semântica e os padrões de funcionamento de um dado *bytecode*, pois apenas visam quebrar o funcionamento dos algoritmos de tradução utilizados pelos decompiladores.

2.1.4.7 Nota sobre transformações

Tipos particulares de transformações, notadamente as léxicas e de abstração/layout, funcionam baseadas na redefinição de identificadores de métodos, classes, pacotes, etc. Se aplicadas isoladamente, em apenas uma classe, ou apenas alguns métodos, podem causar

problemas na execução da aplicação, pois deste modo irão potencialmente inutilizar chamadas polimórficas a métodos, ou mesmo provocar falhas na resolução de nomes, que é feita em tempo real pela JVM.

Transformações de abstração/layout podem também provocar falhas ao alterar a interface pública de classes presentes em aplicações Java distribuídas, por exemplo, na forma de APIs, para reutilização em aplicações de terceiros. A aplicação do ofuscamento neste caso deve seguir rígidos padrões para evitar que dentre as versões da aplicação ofuscada haja diferenças que comprometam o acesso às propriedades públicas das classes. A utilização de dicionários de nomes, funcionalidade muitas vezes presente nas ferramentas de ofuscamento, pode ajudar a estabelecer os padrões necessários para manter a integridade das interfaces públicas de uma aplicação na qual o ofuscamento será aplicado.

Por isso é preciso que toda a aplicação seja ofuscada utilizando os mesmos formatos: mesmos nomes para cada referência a uma classe, método, ou pacote específico, substituição de chamadas de métodos inexistentes por novos criados a partir de quebras de abstração, etc. Algumas ferramentas tomam a precaução de ofuscar pacotes de classes jar, para evitar este problema, mas muitas vezes permitem a transformação de arquivos *class* individuais, o que pode causar graves problemas em aplicações com diversas classes. É preciso, portanto, ter certo cuidado ao transformar classes de um projeto, sempre levando em conta esta premissa, de que aplicações com múltiplas classes que se interconectam através de herança e polimorfismo devem sofrer ofuscamento juntas.

Note que os exemplos duplos acima, contendo o código original e o código transformado, foram apresentados da maneira como o decompilador reconstruiu o código original, e como o processo de tradução de *bytecode* feito por estas ferramentas nunca é 100% preciso, com frequência podem aparecer instruções ilegais, ou blocos de código não traduzidos, notadamente quando este sofre transformações de controle, tornando incompiláveis os códigos. Isto também pode ser verdade se transformações preventivas, cujo objetivo é justamente este, não forem aplicadas.

2.1.5 Consequências do ofuscamento

No capítulo 1.3 foram expostas as principais vantagens do uso do ofuscamento de

código em projetos de software, com relação à proteção conferida aos códigos-fonte contra tentativas de engenharia reversa. Porém, esta proteção traz consigo certas consequências. Dentre estas consequências podemos citar flutuações no desempenho geral da aplicação, modificações no cronograma do projeto, dificuldades no processo de manutenção, etc. O nível de proteção necessário deve ser comparado a estes fatores pois nem sempre estas consequências podem ser aceitas.

Um desenvolvedor provavelmente não perderá tempo executando suas rotinas de *debugging* em cima de códigos ofuscados, já que é suposto que esteja de posse dos códigos-fonte originais, portanto a manutenção não é algo que seja prejudicado diretamente pelo uso do ofuscamento. Uma possível, porém indireta, dificuldade com relação à manutenção talvez seja a necessidade de atualização da aplicação, para acrescentar novas funcionalidades, corrigir funções existentes, etc, pois para isso os novos blocos de código devem ser também ofuscados antes de distribuídos para que sejam compatíveis com o núcleo da aplicação, já ofuscado, demandando um certo tempo adicional, que pode ser indesejável.

Dependendo do nível de ofuscamento aplicado a um código, o programa resultante pode não ter um desempenho tão bom quanto o que seu equivalente não ofuscado teria, em razão da presença de blocos de instruções extras e saltos no fluxo de controle adicicionados por transformações de controle. Isto também contribui para produzir variações de tamanho em bytes em programas ofuscados, e isso pode se tornar um empecilho na distribuição de softwares através de redes em geral, já que nem todos tem acesso às mais altas velocidades de transferência de dados via *web*.

São estes alguns *overheads* produzidos pela aplicação do ofuscamento a um projeto. A decisão de utilizar ou não o ofuscamento dependerá do resultado obtido da comparação do nível de proteção necessário com a quantidade de *overhead* aceitável, já que estas são geralmente variáveis inversamente proporcionais. Isto na prática significa que um projeto que implementa técnicas de ofuscamento em grandes proporções provavelmente causará muitos transtornos, tanto para o desenvolvedor quanto para o usuário, e isso pode não ser interessante.

2.2 Tecnologias empregadas

Para o desenvolvimento do estudo, foram empregadas, além da tecnologia Java, outras tecnologias para a execução dos casos de testes, e para a construção e testes do algoritmo de ofuscamento implementado junto a este trabalho. Abaixo estão descritas:

- **IDE Eclipse** – para construção e testes do algoritmo;

- **Ofuscadores ProGuard e Smokescreen** e **decompilador Jad** – para geração e análise de bytecodes ofuscados.

- **Expressões regulares** – para a implementação do interpretador de símbolos presente no algoritmo.

2.2.1 Eclipse

O Eclipse (http://www.eclipse.org) é uma conhecida ferramenta para desenvolvimento interativo de software, compatível com Java e diversas outras linguagens, como C, C++, Ruby. É largamente utilizada pela sua extensibilidade e pela sua facilidade de personalização, possibilitando a instalação de *plugins* para torná-lo compatível com qualquer tecnologia existente. É um ambiente de desenvolvimento interativo de código aberto, desenvolvido e mantido pela comunidade *open source*.

No que tange ao contexto da tecnologia Java, o Eclipse fornece uma vasta gama de funções que abrangem todos os pacotes de desenvolvimento Java, incluindo a Standard Edition, Mobile Edition e Enterprise Edition, sendo uma ferramenta bastante adequada para desenvolvimento Java em qualquer plataforma. Pode ser facilmente configurado com diferentes compiladores e servidores, além de atuar como gerenciador de bases de dados e também permitir modelagem de software a partir do seu próprio ambiente.

2.2.2 ProGuard

ProGuard (http://proguard.sourceforge.net) é uma ferramenta de ofuscamento de código, de código aberto, voltada exclusivamente para a plataforma Java, e talvez a solução mais conhecida e utilizada no mercado. Capaz de otimizar e aumentar a velocidade de execução de códigos, além de propriamente ofuscá-los, é uma ferramenta gratuita bastante

versátil, e pode inclusive se integrar ao Wireless Toolkit, ambiente para desenvolvimento de aplicações Java para plataformas móveis, exercendo a função de compactador. Possui uma vasta documentação e permite personalizar a maneira como se dará o processamento de um dado código.

2.2.3 Smokescreen

O ofuscador Smokescreen (http://www.leesw.com/smokescreen/index.html) é uma ferramenta comercial, voltada para aplicações Java, que conta com diversas funções de ofuscamento e otimização de código. Abrange a maioria das técnicas ofuscamento estudadas e pode ser facilmente integrado a *frameworks* para uso conjunto.

2.2.4 Jad

Jad (http://java.decompiler.free.fr/) é um conhecido decompilador, escrito em C++, presente como motor de decompilação em diversas aplicações, como o JD-GUI e o DJ Java Decompiler, ambos ferramentas com interface gráfica, e também compatível com IDEs através de *plugins*, como o disponível para o IDE Eclipse, o JadClipse, sendo uma ferramenta largamente utilizada no mercado.

2.2.5 Expressões regulares

Sendo uma solução criada para encontrar padrões textuais através de uma linguagem própria, expressões regulares fornecem meios para validação de dados, análise léxica, reconhecimento de padrões, etc, abrangendo diversas plataformas de programação[9]. É implementada por diversas tecnologias, como Java, Python, C++, Javascript, etc, auxiliando o programador na análise de textos, uma tarefa que teria de ser feita de maneira braçal sem as mesmas. Opera através de uma linguagem padrão, que é capaz de reconhecer os mais variados padrões textuais: palavras, frases, conjuntos de caracteres específicos, etc.

3. Desenvolvimento de um algoritmo de ofuscamento

O algoritmo de ofuscamento proposto para este estudo, apresentado na forma de um protótipo funcional, é composto pelos seguintes componentes:

- **SimpleParser** – uma simples implementação de parser para leitura e interpretação de símbolos presentes no código-fonte Java a ser ofuscado, capaz de identificar nomes de métodos, atributos e classes. Sua função é reunir os símbolos encontrados, e fornecer acesso a eles ao componente Obfuscator, para que este possa mapear os símbolos juntamente a seus respectivos novos identificadores.

- **SimpleCipher** – uma simples implementação da cifra de substituição ROT13[8], para uso na substituição de literais String por sua versão criptografada.

- **Obfuscator** – o ofuscador propriamente dito, cujo algoritmo utiliza os símbolos encontrados pelo componente SimpleParser e o algoritmo criptográfico presente no componente SimpleCipher, para processar o código-fonte alvo, modificá-lo e gerar um novo arquivo de código-fonte ofuscado. Cria combinações de nomes através de duas tabelas de ofuscamento e, então, aplica sobre o código alvo as transformações léxica, modificando identificadores de métodos, atributos e classes usando as combinações de nomes, e de dados, criptografando literais String presentes no código-fonte, e adicionando as instruções necessárias para a decriptação dos mesmos em tempo de execução.

3.1 Motivação

A idéia do algoritmo aqui proposto, que converte um arquivo de código-fonte para outro arquivo de código-fonte ofuscado, aplicando as técnicas acima descritas, e não um *bytecode* para outro *bytecode* ofuscado, sustenta-se na relevância inerente à possibilidade da visualização do código ofuscado antes da compilação, não para fins comerciais, mas para fins acadêmicos. Neste formato, a comparação do código original com o código ofuscado é de total precisão, pois não é necessária qualquer conversão do formato binário para o formato textual, algo que pode significativamente reduzir a semelhança entre eles e portanto diminuir a precisão durante esta comparação.

3.2 Funcionamento

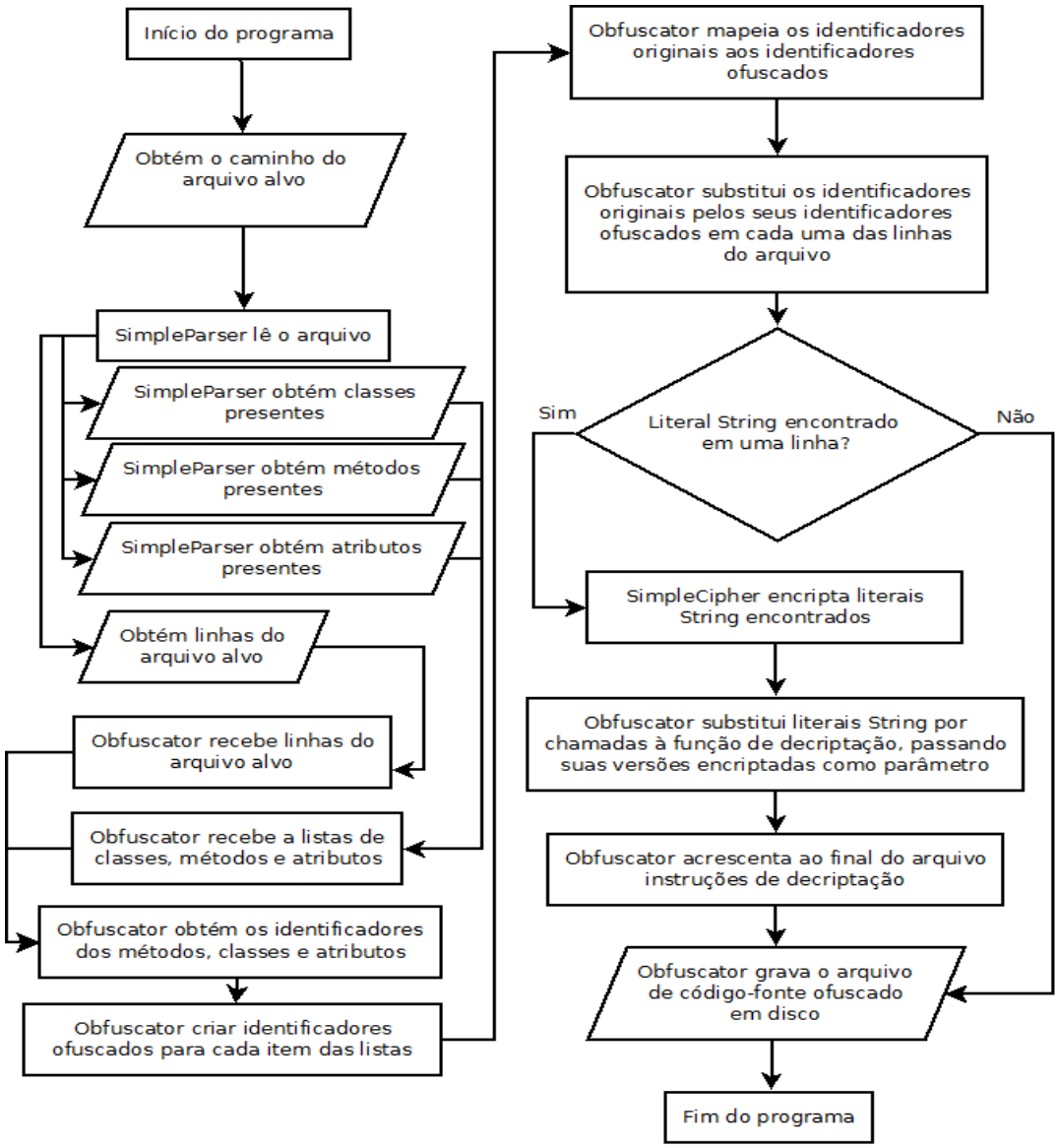

Figura 10 - Diagrama de fluxo simplificado do algoritmo.

O algoritmo proposto inicia obtendo do usuário o local do arquivo de código-fonte Java a ser ofuscado. Com base no arquivo escolhido, SimpleParser age sobre ele identificando, linha a linha, métodos, classes e atributos, procurando por suas declarações no arquivo, e extraindo seus identificadores para uso posterior por Obfuscator. SimpleParser funciona baseado no reconhecimento de padrões de texto através de expressões regulares[9], técnica que auxilia na identificação das informações necessárias com certa precisão.

Após a leitura e interpretação dos símbolos, SimpleParser popula suas listas de declarações, disponíveis para obtenção em sua interface pública, com os métodos, classes e atributos encontrados no arquivo de código-fonte alvo.

Terminada a análise do código-fonte, Obfuscator obtém de SimpleParser as listas de métodos, classes e atributos, assim como a lista de linhas do arquivo, organizadas em sequência, da forma como foram lidas. Esta lista de linhas será manipulada para a aplicação das transformações léxica e de dados.

Utilizando duas tabelas de ofuscamento, compostas pelas 23 letras do alfabeto português excluindo as lestras Y, K e W; e pelos números de 0 a 9, respectivamente, Obfuscator cria combinações pseudo-aleatórias de identificadores, que posteriormente serão combinados com os identificadores reais dos métodos, classes e atributos em um mapa de dados no formato HashMap.

Através do mapa de dados construído, onde cada par de valores é composto pelo nome real de um método, classe ou atributo, e seu novo identificador a ser aplicado na versão ofuscado do arquivo de código-fonte, Obfuscator inicia o processo de transformação léxica do arquivo, substituindo cada identificador real de método, classe ou atributo, por sua versão ofuscada, percorrendo cada uma das linhas do arquivo. Além disto, Obfuscator identifica literais String presentes no texto das linhas, para que sejam encriptadas por SimpleCipher.

SimpleCipher é uma implementação da cifra de substituição ROT13, que desloca cada caractere de uma sequência, de acordo com sua posição no alfabeto, 13 posições à frente, preservando maiúsculas e minúsculas e caracteres diferentes de letras[8]. Sua função no contexto do ofuscamento é encriptar literais String encontrados por Obfuscator, que por sua vez substituirá suas ocorrências por chamadas ao método de decriptação, presente no conjunto de instruções que será adicionado ao final do arquivo, executando transformação de dados sobre o código-fonte alvo.

Terminada a etapa de transformações, Obfuscator grava, linha a linha, o novo arquivo de código-fonte ofuscado, no mesmo local de onde foi lido o arquivo de código-fonte original.

3.3 Considerações

Como o algoritmo é um protótipo, foi desenvolvido dentro de um escopo restrito e como já mencionado, para fins acadêmicos. Para que o algoritmo funcione corretamente, algumas regras devem ser seguidas na escrita do código-fonte a ser ofuscado. Abaixo estão descritas:

- Deve haver apenas uma declaração de método, atributo ou classe por linha de código;

- Todas as classes da aplicação a ser ofuscada devem ser declaradas num único arquivo;

- Em declarações de métodos, o nome dos métodos deve estar imediatamente seguido de parênteses;

- Em declarações de classes, deve haver pelo menos um espaço entre o nome da classe e o sinal de chaves de abertura do bloco da classe;

- Em declarações de atributos, o nome dos atributos deve estar imediatamente seguido de ponto e vírgula;

- Não pode haver mais que um literal String em uma mesma linha de código;

Além disto, o componente parser do algoritmo, SimpleParser, não é capaz de reconhecer declarações com os modificadores *synchronized* e *volatile,* e também não é capaz de identificar variáveis de escopo local. Ambas as incapacidades não comprometem a maneira como o conceito é demonstrado através do algoritmo, portanto foram mantidas fora do escopo do mesmo.

Tratando-se de um protótipo que pretende demonstrar uma idéia pouco difundida entre as soluções de ofuscamento existentes, a de gerar código-fonte a partir de outros códigos-fonte, não é o objetivo desta implementação apresentar os níveis adequados de segurança e proteção esperados de um algoritmo de ofuscamento, mas sim demonstrar o conceito da aplicação do ofuscamento diretamente sobre o código-fonte em formato textual, por isso o escopo do componente parser, SimpleParser, e do componente criptográfico, SimpleCipher, não foi estendido além do necessário para a compreensão do mesmo.

4. Estudo de caso

Neste estudo de caso, serão analisadas as seguintes variáveis, acerca das técnicas de ofuscamento implementadas pelas soluções existentes no mercado:

- **Tamanho dos arquivos class gerados;**
- **Tempo de execução;**
- **Grau de dificuldade encontrado na decompilação.**

O **tamanho dos arquivos** de uma aplicação é uma importante característica, pois vivemos em uma época onde os principais meios de transmissão são as redes cabeadas e sem fio, e com variados tipos e velocidades de conexão, é preferível que seja possível compactar da melhor forma possível os arquivos de uma aplicação. Aplicar o ofuscamento, sem que técnicas de otimização sejam paralelamente utilizadas, sobre uma aplicação Java pode implicar em um aumento substancial do tamanho dos arquivos compilados, o que pode tornar mais lenta a transmissão desta por redes sem fio, por exemplo.

Um **tempo de execução** considerado adequado indica geralmente que a aplicação implementa técnicas para otimizar a maneira como seu código é processado, o que significa um aumento na sua velocidade de execução e portanto diminuição de seu tempo de execução. É essencial que o ofuscamento não comprometa a velocidade com que o algoritmo desempenha sua tarefa. Priorizar a segurança e sacrificar o desempenho pode nem sempre ser interessante.

Também é importante que o ofuscamento impossibilite, ou ao menos dificulte, o processo de decompilação, induzindo a produção de resultados incorretos, e assim afastando o atacante da interpretação real do programa ofuscado. O **grau de dificuldade** encontrado durante esse processo deve ser preferivelmente o maior possível, sendo que o maior grau de dificuldade significa uma menor eficiência na tradução do código ofuscado para código-fonte, apresentando erros ou inconsistências (como instruções que provocam erros de compilação ou instruções ilegais), e o menor grau de dificuldade significa maior eficiência, onde o código é traduzido com o mínimo de erros ou inconsistências. As diferentes técnicas de ofuscamento reduzem a precisão de um decompilador, impondo obstáculos e induzindo-o a provocar estes erros, mas o grau de dificuldade no processo de

decompilação também depende do esforço que o cracker está disposto a fazer para chegar do código ofuscado ao código real, se o uso de um decompilador não for o bastante, além de, é claro, o tempo que a decompilação manual levaria. Neste caso, pode-se dizer que maiores graus de dificuldade desmotivam o cracker a continuar seu trabalho e o induzem à desistência não por falta de conhecimento, mas por exaustão, como já discutido nos capítulos anteriores.

Para isto, foram utilizadas as seguintes ferramentas:

- Para ofuscamento: ProGuard e Smokescreen (o último em sua versão de avaliação);
- Para decompilação: JAD;

O programa-exemplo abaixo, contendo um algoritmo recursivo de cálculo de fatoriais, que calcula e exibe o fatorial do número 20, repetindo o cálculo por dez mil vezes, foi utilizado para demonstrar os resultados obtidos:

```
public class Main {
      public static void main(String[] args) {
                long startTime = System.currentTimeMillis();
                Factorial fact = new Factorial();
                fact.setAmount(10000);
                fact.process();
                long stopTime = System.currentTimeMillis();
                long runTime = stopTime - startTime;
                System.out.println("Run time: " + runTime);
                System.exit(0);
    }
}

class Factorial {
        private int amount;

        public void setAmount(int amount) {
                this.amount = amount;
        }

        public void process() {
                for(int i = 1; i <= amount; i++)
                        System.out.println(factorial(20));
        }
```

```
        private long factorial(long number) {
                if (number <= 1)
                        return 1;
                else
                        return number * factorial(number - 1);
        }

}
```

Programa de cálculo de fatoriais.

Os bytecodes aplicados nos testes abaixo foram modificados sob as seguintes circunstâncias:

- ProGuard: com todas as opções de ofuscamento ativadas (transformação léxica apenas) e as opções de otimização de código desativadas;

- ProGuard: com todas as opções de ofuscamento ativadas (transformação léxica apenas);

- Smokescreen: com todas as opções de ofuscamento ativadas (transformação léxica, de controle e de dados);

4.1 Teste de tempo de execução

Neste teste, o objetivo foi mensurar a velocidade de execução das diferentes versões do *bytecode* ofuscado em comparação com a velocidade de execução do *bytecode* original, sem modificações. Cada versão do mesmo algoritmo foi processada pelas ferramentas, e executada 20 vezes cada uma, em sequência, produzindo os seguintes tempos de execução, em milissegundos, e as seguintes porcentagens de aumento ou diminuição em relação aos tempos e execução do código original:

	Original	ProGuard[1]		ProGuard[2]		Smokescreen	
	Tempo	Tempo	Proporção	Tempo	Proporção	Tempo	Proporção
	952	1544	+62.18%	1326	+39.28%	1248	+31.09%
	1123	1373	+22.26%	1123	-	842	-25.02%
	842	1139	+35.27%	1139	+35.27%	1123	+33.37%
	1030	1498	+45.43%	1294	+25.63%	1544	+49.90%
	1466	1263	-13.84%	1482	+1.09%	1124	-23.32%
	1451	1123	-22.60%	1155	-20.39%	1139	-21.50%
	1264	842	-33.38%	1139	-9.88%	842	-33.38%
	1123	1139	+1.42%	842	-25.02%	1139	+1.42%
	843	1482	+75.80%	842	-0.11%	842	-0.11%
	1123	1544	+37.48%	1280	+13.98%	1139	+1.42%
	1124	1139	+1.33%	1139	+1.33%	1123	-0.08%
	1528	827	-45.87%	1124	-26.43%	843	-44.82%
	1123	842	-25.02%	1342	+19.50%	889	-20.83%
	905	1138	+25.74%	842	-6.96%	1124	+24.19%
	1123	842	-25.02%	843	-24.93%	1123	-
	1357	1498	+10.39%	858	-36.77%	1123	-17.24%
	1123	1310	+16.65%	1466	+30.54%	1123	-
	1124	1124	-	1341	+19.30%	1326	+17.97%
	1123	1108	-1.33%	858	-23.59%	1419	+26.35%
	1466	843	-42.49%	1124	-23.32%	1404	-4.22%
Média	1160,65	1180,9	+1.74%	1054,65	-9.13%	1123,95	-3.16%

Vê-se que as médias de tempo de execução obtidas dos tempos de execução do código original e do código ofuscado pelo ofuscador ProGuard, com suas opções de otimização desativadas, são ligeiramente semelhantes. O ofuscador ProGuard aplica sobre o código apenas ofuscamento de natureza estática, por isso não provoca grandes mudanças no tempo de execução.

Pode-se notar que os ofuscadores Smokescreen e ProGuard, este último quando

1 Com opções de otimização desativadas.

2 Com opções de otimização ativadas.

devidamente configurado para otimizar o código, produziram melhores médias de velocidade de execução em relação ao *bytecode* original, por incluir rotinas de otimização juntamente aos algoritmos de ofuscamento léxico, de fluxo de controle e encriptação de strings utilizados para modificar o *bytecode*.

Além disto, é importante mencionar que o ofuscador ProGuard permite optar ou não pela otimização de *bytecode*, e isso se reflete nos tempos médios de cada *bytecode*, que diferem-se de maneira significativa.

4.2 Teste de tamanho dos arquivos class

Este teste visou comparar as variações de tamanho das classes do programa-exemplo produzidas pelas ferramentas de ofuscamento. Os tamanhos são apresentados em bytes:

	Original	**ProGuard[1]**	**ProGuard[2]**	**Smokescreen**
Factorial	631b	457b	459b	660b
Main	804b	667b	610b	1,19kb

Note que, como o ofuscador ProGuard implementa apenas a técnica de ofuscamento léxico, encurtando os identificadores presentes no código, houve uma diminuição significativa no tamanho dos arquivos gerados por ele, tanto com opções de otimização ativadas quanto desativadas, e é possível verificar uma interessante vantagem em relação aos *bytecodes* originais, não ofuscados.

O ofuscador Smokescreen produziu *bytecodes* um pouco maiores, chegando a exceder 1 kilobyte no caso da classe Main. Isso se deve ao fato deste implementar técnicas avançadas de ofuscamento, não estáticas, que incluem transformações de fluxo, e tendem a provocar um certo aumento no tamanho do arquivo. Também inclui a técnica de encriptação de strings que implica na inclusão de código adicional, contendo funções criptográficas entre as instruções do programa original. Tais técnicas oferecem um nível de proteção superior, ao custo do aumento no tamanho dos *bytecodes*.

1 Com opções de otimização desativadas.

2 Com opções de otimização ativadas.

4.3 Teste de grau de dificuldade da decompilação

Utilizando a ferramenta Jad, os *bytecodes* produzidos pelos ofuscadores ProGuard e Smokescreen passaram pelo processo de decompilação, apresentando os seguintes resultados:

	ProGuard	**Smokescreen**
Eficiência da tradução do bytecode para código-fonte	Total	Parcial
Presença de erros de tradução	Não	Sim
Semelhança do código ofuscado com o original	Similar	Pouco similar

Jad conseguiu traduzir o *bytecode* ofuscado pelo ofuscador ProGuard sem problemas, produzindo código-fonte sem erros que impediriam a compilação.

Já os *bytecodes* ofuscados pelo ofuscador Smokescreen foram parcialmente traduzidos corretamente, mas apresentando vários blocos de erros de decompilação, em razão da incapacidade do decompilador Jad de traduzir certos trechos de código modificados pelo ofuscador Smokescreen.

A semelhança em relação ao código original foi muito maior no *bytecode* gerado pelo ofuscador ProGuard,devido ao fato deste implementar técnicas de ofuscamento simples, que não provocam falhas no funcionando do decompilador e tampouco alteram drasticamente a aparência do programa. Desta forma, o uso do decompilador seria o bastante para que o cracker obtivesse o código original com maior precisão, se este fosse ofuscado com o ofuscador ProGuard.

4.4 Conclusões tiradas a partir dos testes

Das ferramentas testadas, o ofuscador ProGuard mostrou-se uma solução onde a relação das variáveis estudadas pode oferecer, a aplicações, a possibilidade de gerar arquivos menores e código mais veloz em um nível de proteção mediano. Porém, o *bytecode* gerado por este ofuscador não oferece grandes dificuldades para que ferramentas de decompilação traduzam-no corretamente para um código-fonte próximo à aparência real do programa.

Como o ofuscador ProGuard não implementa técnicas de ofuscamento dinâmicas, que alteram o fluxo e a aparência dos dados presentes no código, a proteção por ele oferecida pode ser facilmente violada, como também demonstrado no teste de nível de dificuldade no processo de decompilação, no qual o código gerado por este ofuscador se mostrou similar ao original. Sendo assim esta ferramenta não deveria ser utilizada em projetos onde a prioridade é a integridade da lógica de negócios em relação a ataques de crackers.

Quanto ao nível de segurança, o ofuscador Smokescreen se mostrou mais eficiente, pois além de implementar técnicas de ofuscamento que alteram o código de maneira dinâmica, provou-se eficaz ao atrapalhar o processo de decompilação, induzindo a ferramenta, neste caso decompilador Jad, a erros e traduções incorretas. Oferecer um nível alto de proteção contra ataques à integridade da lógica de negócios é uma vantagem que vem ao custo de um aumento substancial no tamanho dos arquivos gerados, pois há inclusão de código adicional, como as rotinas criptográficas que protegem os dados que transitam em formato *string* pelo programa, e isto ficou evidente no resultado obtido através dos arquivos gerados por este ofuscador.

Para não prejudicar a velocidade de execução, é interessante que uma ferramenta que implemente técnicas de ofuscamento dinâmico seja capaz de otimizar o código gerado, pois transformações de dados e de fluxo podem causar flutuações no tempo de execução da aplicação, devido à adição de desvios e instruções extras, e isto também pode ser verificado no ofuscador Smokescreen, vistos os testes de tempo de execução em que este produziu *bytecodes* com boas velocidades de execução.

Em relação ao ofuscador ProGuard, o ofuscador Smokescreen oferece um maior nível de proteção, ao custo de maximizar o tamanho dos *bytecodes*, uma características que pode ou não ser uma desvantagem, mas que em alguns casos pode ser uma variável irrelevante, diante da maior proteção com ele obtida.

5. Conclusões do estudo

Através do estudo, pode-se concluir que ofuscamento de código é uma área de conhecimento ainda pouco explorada, com poucas obras escritas especificamente sobre o assunto, mas que pode oferecer uma alternativa, ou mesmo uma adição, interessante aos sistemas de proteção à propriedade intelectual presente em softwares.

Há um grande domínio sobre tecnologias relacionadas por um pequeno conjunto de empresas, que detêm a maioria do conhecimento em técnicas de ofuscamento e distribuem comercialmente suas soluções, o que notadamente contrasta com a relativa ausência da comunidade *open source* na implementação das técnicas de ofuscamento mais avançadas, portanto oferecem soluções que não são capazes de oferecer grandes níveis de proteção a uma aplicação, e uma aplicação de software voltada para o comércio pode não ser o melhor substrato para a aplicação de técnicas mais simples. Os maiores níveis de proteção são obtidos pelo maior investimento, já que, como dito, as soluções avançadas e específicas são mantidas em formato de código fechado, para desenvolvedores que desejam realmente dificultar e desencorajar o esforço de um atacante ao violar a integridade de seu trabalho, mas estejam dispostos a adquirir estas soluções.

Entre as dificuldades envolvidas no desenvolvimento do estudo, está justamente a falta da divulgação de conhecimento sobre as técnicas de ofuscamento no meio não comercial, devido à já citada falta de exploração e aplicação das mesmas em projetos de software. É uma área, até o presente momento, fortemente baseada em conceitos teóricos com poucas aplicações práticas reais, o que acaba por desmotivar o surgimento de novas publicações sobre o assunto, o que também contribuiu para impor obstáculos ao estudo.

O ofuscamento em relação à tecnologia Java não se mostrou difícil de se aplicar, pois no mercado temos muitas soluções, comerciais e abertas, disponíveis especificamente para o programador Java, que não deseja migrar para outras plataformas para obter maiores níveis de proteção. Implementar o ofuscamento de código sobre a plataforma Java não se provou uma tarefa complicada, pois a mesma oferece o suporte necessário para a aplicação da maioria das técnicas existentes, mostrando-se uma potencial tecnologia para o futuro, dizendo-se respeito às técnicas de proteção de software contra *cracking*.

Várias são as tecnologias relacionadas que ajudam a montar as peças de um

algoritmo de ofuscamento: funções criptográficas, reconhecimento de padrões, analisadores léxicos, aplicação de modelos matemáticos (como sistemas binários, árvores e grafos), comportamento dinâmico, programas automodificáveis, etc. Estas áreas citadas poderiam receber estudos mais aprofundados e mais particulares para uma melhor compreensão do ofuscamento e suas técnicas.

Para extensão do conceito demonstrado pela implementação do algoritmo de ofuscamento deste estudo, a aplicação de técnicas adicionais, como transformações de controle ou abstração, seria ideal, e enriqueceria provavelmente a quantidade de conhecimento transmitido em trabalhos futuros sobre este tema.

6. Referências

1 Collberg, C. e Nagra, J. **Surreptitious Software – Obfuscation, Watermarking and Tamperproofing for Software Protection**, Addison-Wesley, 1ª edição, 2009, EUA.

2 Low, D. **Protecting Java Code Via Code Obfuscation**. Disponível em <http://www.cs.arizona.edu/~collberg/Research/Students/DouglasLow/obfuscation.html>, acessado em 15/05/2012.

3 Leskov, D. **Protect Your Java Code - Through Obfuscators And Beyond**. Disponível em <http://www.excelsior-usa.com/articles/java-obfuscators.html>, acessado em 15/05/2012.

4 Venkatesan, A. **Code Obfuscation and Virus Detection**. Disponível em <http://scholarworks.sjsu.edu/cgi/viewcontent.cgi?article=1115&context=etd_projects&sei-redir=1>, acessado em 15/05/2012.

5 Chan, J. e Yang W. **Advanced Obfuscation Techniques for Java Bytecode**. Disponível em <http://users.rowan.edu/~tang/courses/ref/AdvObfuscator.pdf>, acessado em 17/05/2012.

6 **Classes dinâmicas**. Disponível em <http://help.adobe.com/pt_BR/ActionScript/3.0_ProgrammingAS3/WS5b3ccc516d4fbf351e63e3d118a9b90204-7f89.html>, acessado em 07/07/2012.

7 **Java Class Loading: The Basics** .Disponível em <http://www.developer.com/java/other/article.php/2248831/Java-Class-Loading-The-Basics.htm>, acessado em 07/07/2012.

8 **What is ROT-13?** .Disponível em <http://searchsecurity.techtarget.com/definition/ROT-13>, acessado em 27/07/2012.

9 Jargas, A. M. **Expressões Regulares - Uma abordagem divertida**, Novatec, 2ª edição, 2008, Brasil.

10 **Thwart Reverse Engineering of Your Visual Basic .NET or C# Code** .Disponível em

<http://msdn.microsoft.com/en-us/magazine/cc164058.aspx>, acessado em 09/08/2012.

Java Name Obfuscation. Disponível em <http://www.zelix.com/klassmaster/featuresNameObfuscation.html>, acessado em 14/05/2012.

Java Flow Obfuscation. Disponível em <http://www.zelix.com/klassmaster/featuresFlowObfuscation.html>, acessado em 16/05/2012.

Name Obfuscation. Disponível em <http://www.allatori.com/features/name-obfuscation.html>, acessado em 16/05/2012.

Flow Obfuscation. Disponível em <http://www.allatori.com/features/flow-obfuscation.html>, acessado em 16/05/2012.

String Encryption. Disponível em <http://www.allatori.com/features/string-encryption.html>, acessado em 16/05/2012.

7. Anexos

Códigos-fonte do algoritmo desenvolvido

Classe Main

```
package core;


public class Main {

	public static void main(String[] args) {
		if(args.length == 0 || args.length > 1) {
			System.out.println("Número de argumentos incorreto.");
			System.exit(0);
		}

		String filePath = args[0];

		SimpleParser parser = new SimpleParser(filePath);
		parser.parse();
		Obfuscator obf = new Obfuscator(parser.getMethodList(),
parser.getMemberList(),

parser.getClassList(), parser.getLineList());
		obf.obfuscate(filePath);
	}
}
```

Classe Obfuscator

```
package core;

import java.io.BufferedWriter;
import java.io.File;
import java.io.FileWriter;
import java.io.IOException;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;
import java.util.Random;
import java.util.regex.Matcher;
import java.util.regex.Pattern;
```

```
import javax.swing.JDialog;

import parser.ClassDeclaration;
import parser.Declaration;
import parser.Member;
import parser.Method;

/**
 * Classe de ofuscamento que implementa as técnicas de transformação léxica,
abrangendo nomes de métodos, membros e
 * classes, e transformação de dados, através da encriptação de Strings.
 * Para que funcione corretamente, algumas regras devem ser seguidas:
 * - deve haver apenas um literal String em uma dada linha de código;
 * @author Daniel
 *
 */
public class Obfuscator {

        /**
         * Lista de métodos.
         */
        private ArrayList<Method> methodList;
        /**
         * Lista de membros.
         */
        private ArrayList<Member> memberList;
        /**
         * Lista de classes.
         */
        private ArrayList<ClassDeclaration> classList;

        /**
         * Lista de linhas do arquivo de código-fonte.
         */
        private ArrayList<String> lineList;
        /**
         * Lista de linhas da versão ofuscada do arquivo de código-fonte.
         */
        private ArrayList<String> obfLineList;

        /**
         * Tabela de ofuscamento 1, contém todos os caracteres do alfabeto
português.
         */
```

```
        private String[] obfuscationTable1 = new String[]{"a", "b", "c", "d",
"e", "f", "g",
                                                          "h", "i", "j",
"l", "m", "n", "o",
                                                          "p", "q", "r",
"s", "t", "u", "v",
                                                          "x", "z"};

        /**
         * Tabela de ofuscamento 2, contém todos os números de 0 a 9.
         */
        private String[] obfuscationTable2 = new String[]{"0", "1", "2", "3",
"4", "5", "6", "7", "8", "9"};

        /**
         * Utilizado para receber a combinação atual.
         */
        private String combination;
        /**
         * Utilizado para armazenar o nome do método decriptador.
         */
        private String cipherMethodName;
        /**
         * Utilizado para armazenar o nome da classe que contém o método
decriptador.
         */
        private String cipherClassName;

        /**
         * Lista de combinações, onde são armazenadas cada uma das combinações
criadas.
         */
        private ArrayList<String> combinationList = new ArrayList<String>();

        /**
         * Lista que associa símbolos encontrados por SimpleParser a seus novos
         * identificadores baseados nas combinações para eles criadas.
         */
        private HashMap<Declaration, String> obfuscationMap = new
HashMap<Declaration, String>();

        /**
         * Utilizado para armazenar o nome da classe pública do arquivo de
código-fonte.
         */
```

```
        private String obfPublicClassName;

        /**
         * Utilizado para encriptar Strings encontradas em obfuscate().
         */
        private SimpleCipher simpleCipher;
        /**
         * Flag utilizada para checar se é necessário incluir o código de
decriptação ou não.
         */
        private boolean appendCipherCodeFlag = false;

        /**
         * Local do arquivo de código-fonte.
         */
        private String filePath;

        /**
         * Construtor que iniciliza todas as listas e objetos necessários.
         * Todos os parâmetros são obtidos após SimpleParser agir sobre o
arquivo de código-fonte original.
         * @param methodList lista de métodos encontrados.
         * @param memberList lista de membros encontrados.
         * @param classList lista de classes encontradas.
         * @param lineList arquivo de código-fonte em formato de lista de
Strings.
         */
        public Obfuscator(ArrayList<Method> methodList, ArrayList<Member>
memberList,
                                    ArrayList<ClassDeclaration> classList,
ArrayList<String> lineList) {
                this.methodList = methodList;
                this.memberList = memberList;
                this.classList = classList;

                this.lineList = lineList;

                this.obfLineList = new ArrayList<String>();
                simpleCipher = new SimpleCipher();
        }

        /**
         * Aplica o ofuscamento sobre o código-fonte.
         * @param filePath
         */
```

```
        public void obfuscate(String filePath) {

                this.filePath = filePath;
                removeOriginalFileName();
                buildObfuscationMap();
                transform();
                if(appendCipherCodeFlag)
                        appendCipherCode();
                writeObfuscated();
        }

        /**
         * Remove o nome do arquivo de código-fonte original da String que
contém sua localização.
         */
        private void removeOriginalFileName() {
                Pattern p = Pattern.compile("[a-zA-Z]*.java$");
                Matcher m = p.matcher(this.filePath);
                if(m.find()) {
                        String fileName = m.group(0);
                        this.filePath = this.filePath.replace(fileName, "");
                }
        }

        /**
         * Grava em disco a versão ofuscada do arquivo de código-fonte, no mesmo
local onde o original
         * estava.
         */
        private void writeObfuscated() {
                BufferedWriter bWriter = null;
                try {
                        File output = new File(this.filePath +
this.obfPublicClassName + ".java");
                        if(!output.exists())
                                output.createNewFile();
                        bWriter = new BufferedWriter(new FileWriter(output));
                        Iterator<String> it = obfLineList.iterator();
                        while(it.hasNext()) {
                                bWriter.write(it.next());
                                bWriter.newLine();
                        }
                } catch (IOException e) {
                        e.printStackTrace();
                } finally {
```

```
                        try {
                                bWriter.close();
                        } catch (IOException e) {
                                e.printStackTrace();
                        }
                }
        }

        /**
         * Inclui no final da lista de linhas o código necessário para
encriptação/decriptação.
         */
        private void appendCipherCode() {
                String[] cipherCode = {"class ## {".replace("##",
this.cipherClassName),
                        "public String ##(String str) {".replace("##",
this.cipherMethodName),
                                "StringBuilder strBuilder = new
StringBuilder();",
                        "for (char c : str.toCharArray()) {",
                            "if (c >= 'A' && c <= 'Z') {",
                                "c += 13;",
                                "if (c > 'Z') {",
                                    "c -= 26;",
                                "}",
                            "} else if (c >= 'a' && c <= 'z') {",
                                "c += 13;",
                                "if (c > 'z') {",
                                    "c -= 26;",
                                "}",
                            "}",
                            "strBuilder.append(c);",
                        "}",
                        "return strBuilder.toString();",
                        "}",
                  "}"};

                for(String str: cipherCode) {
                        obfLineList.add(str);
                }
        }

        /**
         * Executa a transformação do código-fonte, fazendo a troca dos
identificadores por novos criados
```

```
        * apartir de combinações de caracteres das tabelas  de ofuscamento e
substituindo literais
        * String por sua versão encriptada na forma de parâmetro para chamada
do método de
        * encriptação/decriptação.
        */
       private void transform() {
               Iterator<String> mainIt = lineList.iterator();

               newCombination();
               this.cipherMethodName = this.combination;
               newCombination();
               this.cipherClassName = this.combination;

               while(mainIt.hasNext()) {
                       String thisLine = mainIt.next();
                       System.out.println(thisLine);
                       StringBuilder cipherCallBuilder = new StringBuilder();

                       String str = this.hasString(thisLine);
                       if(str != null) {
                               this.appendCipherCodeFlag = true;
                               String strEnc = simpleCipher.cipher(str);
                               cipherCallBuilder.append("new ")
                                        .append(cipherClassName)
                                        .append("().")
                                            .append(cipherMethodName)
                                            .append("(")
                                            .append(strEnc)
                                            .append(")");
                               thisLine = thisLine.replace(str,
cipherCallBuilder.toString());
                       }

                       for (Map.Entry<Declaration, String> entry :
obfuscationMap.entrySet()) {
                               if(entry.getKey() instanceof ClassDeclaration) {
                                       ClassDeclaration classDecTemp =
(ClassDeclaration)entry.getKey();
                                       if(classDecTemp.isPublic()) {
                                               this.obfPublicClassName =
entry.getValue();
                                       }
                               }
                               if(thisLine.contains(entry.getKey().getName()))
```

```
{
                                        System.out.println("### Achou -> " +
entry.getKey().getName());
                                        if(entry.getKey() instanceof Method) {
                                                Method methodTemp =
(Method)entry.getKey();
                                                if(!methodTemp.isMain()) {
                                                        thisLine =
thisLine.replace(entry.getKey().getName(), entry.getValue());
                                                }
                                        } else {
                                                thisLine =
thisLine.replace(entry.getKey().getName(), entry.getValue());
                                        }
                                }
                        }
                        obfLineList.add(thisLine);
                }
        }

        /**
         * Checa se uma dada linha contém um literal String.
         * @param line a linha a ser checada.
         * @return um objeto contendo o literal String encontrado.
         */
        private String hasString(String line) {
                Pattern p = Pattern.compile("\\\".*\\\"");
                Matcher m = p.matcher(line);
                String string = null;
                if(m.find()) {
                        string = m.group(0);
                }
                return string;
        }

        /**
         * Monta o mapa de ofuscamento, associando identificadores originais a
seus novos identificadores.
         */
        private void buildObfuscationMap() {
                Iterator<Method> methodIt = methodList.iterator();
                Iterator<Member> memberIt = memberList.iterator();
                Iterator<ClassDeclaration> classIt = classList.iterator();

                while(methodIt.hasNext()) {
```

```
                        newCombination();
                        obfuscationMap.put(methodIt.next(), this.combination);
                }

                while(memberIt.hasNext()) {
                        newCombination();
                        obfuscationMap.put(memberIt.next(), this.combination);
                }

                while(classIt.hasNext()) {
                        newCombination();
                        obfuscationMap.put(classIt.next(), this.combination);
                }
        }

        /**
         * Obtém uma nova combinação de caracteres a partir das tabelas de
ofuscamento.
         */
        private void newCombination() {
                Random rand = new Random();
                this.combination =
obfuscationTable1[rand.nextInt(obfuscationTable1.length)] +

obfuscationTable1[rand.nextInt(obfuscationTable1.length)] +

obfuscationTable2[rand.nextInt(obfuscationTable2.length)];
                if(checkExistentCombination()) {
                        newCombination();
                } else {
                        return;
                }
        }

        /**
         * Checa se a combinação criada atualmente existe ou não na lista de
combinações
         * criadas anteriormente.
         * @return true se a combinação atual existir, false caso contrário.
         */
        private boolean checkExistentCombination() {
                Iterator<String> it = combinationList.iterator();
                boolean exists = false;
                while(it.hasNext()) {
                        if(it.next().equals(this.combination))
```

```
                                exists = true;
                }
                return exists;
        }
}
```

Classe SimpleCipher

```
package core;


/**
 * Implementação em código Java da cifra de substituição ROT13.
 * O mesmo algoritmo pode ser usado tanto para a encriptação quanto para a
decriptação.
 * @author Daniel
 *
 */
public class SimpleCipher {

        /**
         * Recebe um objeto String contendo um dado texto a ser encriptado.
         * @param textToCipher o texto a ser encriptado.
         * @return a versão encriptada do texto passado como parâmetro.
         */
        public String cipher(String textToCipher) {

                StringBuilder strBuilder = new StringBuilder();
        for (char c : textToCipher.toCharArray()) {
            if (c >= 'A' && c <= 'Z') {
                c += 13;
                if (c > 'Z') {
                    c -= 26;
                }
            } else if (c >= 'a' && c <= 'z') {
                c += 13;
                if (c > 'z') {
                    c -= 26;
                }
            }
            strBuilder.append(c);
        }
        return strBuilder.toString();
        }
```

```
}
```

Classe SimpleParser

```
package core;

import java.io.BufferedReader;
import java.io.File;
import java.io.FileNotFoundException;
import java.io.FileReader;
import java.io.IOException;
import java.util.ArrayList;
import java.util.regex.Matcher;
import java.util.regex.Pattern;

import parser.ClassDeclaration;
import parser.Method;
import parser.Member;

/**
 * Simples parser para uso com o algoritmo de ofuscamento léxico. Para que
funcione corretamente, o
 * código deve ser escrito seguindo as regras abaixo, algumas baseadas
sutilmente nas convenções de escrita
 * de código Java:
 * - uma declaração por linha (método, membro, classe, etc);
 * - todas as classes devem ser escritas em um só arquivo;
 * - declarações de atributos, com inicialização opcional, devem ser
imediatamente
 *   seguidas de ponto e vírgula, na mesma linha;
 * - declarações de métodos devem ser imediatamente seguidas de parênteses;
 * Deficiências deste parser:
 * - não reconhece declarações de classes genéricas;
 * - não reconhece declarações de parâmetros de métodos;
 * - não reconhece declarações de escopo local;
 * - não reconhece blocos de comentários;
 * @author Daniel
 *
 */
public class SimpleParser {

        /**
         * Utilizado para obtenção do local de leitura.
         */
        private String filePath;
```

```
        /**
         * Expressão regular para identificação de declarações de métodos.
         */
        private final String METHOD_PATTERN = "(public|private|protected)
( static| abstract)?( final)?( void| int| byte| char| boolean| short| float|
long| double| [A-Z]{1,1}[a-zA-Z\\[\\]<>]*)( [a-zA-Z0-9_]*)\\([a-zA-Z ,\\
[\\]]*\\)";
        /**
         * Expressão regular para identificação de declarações de membros.
         */
        private final String MEMBER_PATTERN = "(public|private|protected)
( static| abstract)?( final)?( int| byte| char| boolean| short| float| long|
double| [A-Z]{1,1}[a-zA-Z]*[\\[<]{0,1}[a-zA-Z ,]*[>\\]]{0,1})( [a-zA-Z0-9_]*)
(.*)?;";
        /**
         * Expressão regular para identificação de declarações de classes.
         */
        private final String CLASS_PATTERN =  "(public|private|protected)?
( abstract)?( final)?( class| interface)( [A-Z]{1,1}[a-zA-Z0-9]*)";

        /**
         * Lista de métodos.
         */
        private ArrayList<Method> methodList;
        /**
         * Lista de membros.
         */
        private ArrayList<Member> memberList;
        /**
         * Lista de classes.
         */
        private ArrayList<ClassDeclaration> classList;

        /**
         * Lista de linhas do arquivo de código-fonte.
         */
        private ArrayList<String> lineList;

        /**
         * Construtor que inicializa todas as listas e obtém o local do arquivo
a ser lido.
         * @param filePath o local do arquivo de código-fonte a processar.
         */
        public SimpleParser(String filePath) {
```

```
                this.filePath = filePath;
                methodList = new ArrayList<Method>();
                memberList = new ArrayList<Member>();
                classList = new ArrayList<ClassDeclaration>();
        }

        /**
         * Lê e identifica os símbolos presentes no código, linha a linha,
preenchendo
         * as listas de declarações, e a lista de linhas, dados que servirão de
parâmetros para
         * Obfuscator.
         */
        public void parse() {
                lineList = new ArrayList<String>();
                String line = "";

                FileReader reader = null;
                BufferedReader bReader = null;

                try {
                        File input = new File(this.filePath);

                        reader = new FileReader(input);
                        bReader = new BufferedReader(reader);

                        while((line=bReader.readLine()) != null) {
                                lineList.add(line);

                                Method method = this.hasMethod(line);
                                if(method != null)
                                        methodList.add(method);

                                Member member = this.hasMember(line);
                                if(member != null)
                                        memberList.add(member);

                                ClassDeclaration classDec = this.hasClass(line);
                                if(classDec != null) {
                                        classList.add(classDec);
                                }

                        }
                } catch (FileNotFoundException e) {
```

```
                        e.printStackTrace();
                } catch (IOException e) {
                        e.printStackTrace();
                } finally {
                        try {
                                reader.close();
                                bReader.close();
                        } catch (IOException e) {
                                e.printStackTrace();
                        }

                }

        }

        /**
         * Identifica se em uma dada linha existe uma declaração de método.
         * @param line a linha atual lida por parser().
         * @return um objeto em formato Method contendo os dados da declaração
encontrada, ou null
         *         se nenhuma declaração de método for encontrado.
         */
        private Method hasMethod(String line) {
                Pattern p = Pattern.compile(this.METHOD_PATTERN);
                Matcher m = p.matcher(line);
                Method method = null;
                String methodName = "";
                if(m.find()) {
                        methodName = m.group(5).trim();
                        method = new Method();
                        method.setName(methodName);
                        if(method.getName().equals("main")) {
                                method.setMain(true);
                        }
                        System.out.println("Método: " + method.getName());
                }
                return method;
        }

        /**
         * Identifica se em uma dada linha existe uma declaração de membro.
         * @param line a linha atual lida por parser().
         * @return um objeto em formato Member contendo os dados da declaração
encontrada, ou null
```

```
         *          se nenhuma declaração de membro for encontrado.
         */
        private Member hasMember(String line) {
                Pattern p = Pattern.compile(this.MEMBER_PATTERN);
                Matcher m = p.matcher(line);
                Member member = null;
                String memberName = "";
                if(m.find()) {
                        memberName = m.group(5).trim();
                        member = new Member();
                        member.setName(memberName);
                        System.out.println("Membro: " + member.getName());
                }
                return member;
        }

        /**
         * Identifica se em uma dada linha existe uma declaração de classe.
         * @param line a linha atual lida por parser().
         * @return um objeto em formato ClassDeclaration contendo os dados da
declaração encontrada, ou null
         *          se nenhuma declaração de classe for encontrado.
         */
        private ClassDeclaration hasClass(String line) {
                Pattern p = Pattern.compile(this.CLASS_PATTERN);
                Matcher m = p.matcher(line);
                ClassDeclaration classDec = null;
                String className = "";
                if(m.find()) {
                        className = m.group(5).trim();
                        classDec = new ClassDeclaration();
                        classDec.setName(className);
                        if(line.contains("public")) {
                                classDec.setPublic(true);
                        }
                        System.out.println("Classe: " + classDec.getName());
                }
                return classDec;
        }

        //getters para acesso externo às listas por Obfuscator.
        public ArrayList<Method> getMethodList() {
                return methodList;
        }
```

```
        public ArrayList<Member> getMemberList() {
                return memberList;
        }

        public ArrayList<ClassDeclaration> getClassList() {
                return classList;
        }

        public ArrayList<String> getLineList() {
                return lineList;
        }
}
```

Classe ClassDeclaration

```
package parser;

/**
 * Representa uma declaração de classe.
 * @author Daniel
 */
public class ClassDeclaration extends Declaration {

        private boolean publicClass;

        public boolean isPublic() {
                return publicClass;
        }

        public void setPublic(boolean publicClass) {
                this.publicClass = publicClass;
        }

}
```

Classe Declaration

```
package parser;

/**
 * Representa uma declaração.
 * @author Daniel
 */
```

```
public abstract class Declaration {

	private String name;

	public String getName() {
		return name;
	}
	public void setName(String name) {
		this.name = name;
	}
}
```

Classe Member

```
package parser;

/**
 * Representa uma declaração de membro.
 * @author Daniel
 */
public class Member extends Declaration {

}
```

Classe Method

```
package parser;

/**
 * Representa uma declaração de método.
 * @author Daniel
 */
public class Method extends Declaration {

	private boolean mainMethod;

	public boolean isMain() {
		return mainMethod;
	}
```

```
        public void setMain(boolean mainMethod) {
                this.mainMethod = mainMethod;
        }
}
```