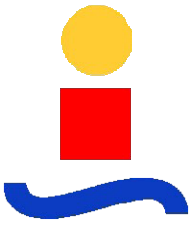
Estudio y Prueba de una Métrica para los Ofuscadores de Java

Proyecto Fin de Carrera

Departamento de Sistemas y Automática, Área de Telemática
Universidad de Sevilla



Autor del proyecto:
Ángel Miralles Arévalo
Tutor del proyecto:
Antonio Jesús Sierra Collado
Sevilla, Mayo 2005

ÍNDICE

1. OBJETIVOS

Internet ha ayudado a Java a situarse como líder de los lenguajes de programación y, por su lado, Java ha tenido un profundo efecto sobre Internet. La razón de esto es bastante simple: Java amplia el universo de objetos que pueden moverse libremente por el ciberespacio. Antes de Java, Internet estaba cerrada a la mitad de las entidades que ahora viven en ella debido a cuestiones de seguridad y portabilidad.

La llave que permite a Java resolver los problemas descritos anteriormente es que la salida de un compilador Java no es código ejecutable, sino que es código binario (*bytecode*). Este código binario es un conjunto de instrucciones altamente optimizado diseñado para ser ejecutado por una máquina virtual que emula el intérprete de Java. Es decir, el intérprete de Java es un intérprete de código binario. Esto hace mucho más fácil ejecutar programas en gran variedad de entornos, solo es necesario implementar para cada plataforma el intérprete de Java.

Es bien conocido el hecho de que los archivos Java "*.class*" pueden ser fácilmente descompilados debido a que el *bytecode* contiene gran parte de la información del código fuente original. Esto constituye una brecha en cuanto a la mencionada seguridad proporcionada por Java. La flexibilidad de Java ofrece una gran cantidad de potenciales ventajas en un entorno distribuido como es Internet, sin embargo, se está bajo amenazas de varios tipos: ingeniería inversa (*reverse engineering*), saboteos (*tampering*) y piratería de software. Es necesaria por tanto una herramienta que impida la fácil descompilación del código fuente. La herramienta a la que hacemos referencia se denomina ofuscador.

Los ofuscadores hacen que el código generado sea prácticamente imposible de descompilar. Tras el tratamiento realizado por un ofuscador, los descompiladores siguen generando código fuente a partir de *bytecode*, pero este código no es como el original, este será incomprensible, haciendo la labor de la ingeniería inversa muy difícil. No existe ningún proceso de ingeniería inversa capaz de deshacer los cambios realizados por un ofuscador, de manera automática.

Nuevos usos de la ofuscación del código se hayan en las aplicaciones para dispositivos móviles. La baja potencia de cálculo y la limitación de recursos buscan en la ofuscación del código una optimización reduciendo el tamaño de las aplicaciones. Este hecho también mejora en la carga más rápida de los *applets* ya que se realiza en un tiempo menor.

El propósito de este proyecto es primeramente el estudio del proceso de ofuscación, la comprensión de las técnicas empleadas. En segundo lugar realizar un análisis de los ofuscadores disponibles en la red, así como una comparativa de la efectividad de cada uno de ellos. Todo esto centrándonos en ofuscación para lenguaje de programación Java.

2. OFUSCACIÓN

2.1. DESCOMPILACIÓN DE CÓDIGO

El Java *bytecode* generado durante la compilación del código fuente del fichero "*.java*" contiene toda la información necesaria para poder regenerar el código fuente original mediante la aplicación de una herramienta denominada descompilador. Hay multitud de descompiladores gratuitos disponibles en Internet, muchos de ellos basados en el proyecto JODE (Entorno de Descompilación y Optimización Java). El propio JDK también incorpora su propio descompilador de Java, el *javap*, si bien es cierto que éste solo es capaz de dar información sobre las cabeceras de los métodos y los atributos de las clases, sin regenerar cuerpo alguno de los métodos, de forma similar a como lo hacen multitud de entornos de desarrollo Java cuando abren un fichero "*.class*".

Vamos a comprobar la potencia de los descompiladores mediante una prueba sencilla. Supongamos el siguiente código fuente "*generadorAleatorios.java* ":

Programa 1.1. Código Fuente generadorAleatorios.java.

```
import java.util.Random;

/**
* <p>Title: </p>
* <p>Description: </p>
* <p>Copyright: Copyright (c) 2004</p>
* <p>Company: </p>
* @author not attributable
* @version 1.0
*/

public class generadorAleatorios {
       int semilla;
       Random random;
       public generadorAleatorios(int s) {
              semilla=s;
              long l=new Long(s).longValue();
              random=new Random(l);
       }
       public int generaEnteroAleatorio(){
              return random.nextInt();
       }
       public long generaLongAleatorio(){
              long aux=random.nextLong();
              return aux;
       }
       public static void main(String[] args) {
       generadorAleatorios generadorAleatorios1 = new generadorAleatorios(33);
       }
}
```

Tras establecer correctamente el *classpath* del entorno (no del sistema), para indicar al descompilador la localización de los "*.class*" a descompilar, el JODE regenera el código fuente original de la clase de forma totalmente automática, como puede verse en programa 1.2, faltando únicamente los comentarios.

Programa 1.2. Código Descompilado mediante JODE.

```
/* generadorAleatorios - Decompiled by JODE
 * Visit http://jode.sourceforge.net/
 */
import java.util.Random;

public class generadorAleatorios
{
    int semilla;
    Random random;

    public generadorAleatorios(int i) {
       ((generadorAleatorios) this).semilla = i;
       long l = new Long((long) i).longValue();
       ((generadorAleatorios) this).random = new Random(l);
    }

    public int generaEnteroAleatorio() {
       return ((generadorAleatorios) this).random.nextInt();
    }

    public long generaLongAleatorio() {
       long l = ((generadorAleatorios) this).random.nextLong();
       return l;
    }

    public static void main(String[] strings) {
       generadorAleatorios var_generadorAleatorios
           = new generadorAleatorios(33);
    }
}
```

Comprobamos por tanto la necesidad de emplear una herramienta que impida la fácil descompilación del código fuente. En otras palabras, si no se toman las precauciones debidas, cualquier aplicación en Java puede ser víctima de la ingeniería inversa.

2.2. DEFINICIÓN DE OFUSCACIÓN

La Ofuscación del código, como técnica de seguridad, es actualmente una de las mejores herramientas de protección frente al "*reverse engineering*". Proporciona un software ininteligible pero con la misma funcionalidad que el código fuente original.

Hay gran cantidad de ofuscadores disponibles en Internet, debido a la creciente demanda, que proporcionan diferentes niveles de seguridad para proteger programas Java.

En la figura 2.1 podemos observar el proceso de ofuscación. Un conjunto de archivos "*.class*" P, al pasar por un ofuscador, se convierten en otro conjunto de archivos "*.class*" P'; siendo el código de ambos conjuntos distinto pero su funcionalidad la misma.

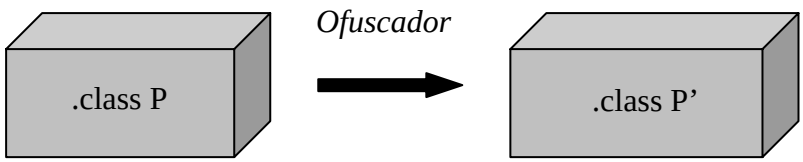


Figura 2.1. Proceso de Ofuscación.

Como hemos comentado anteriormente otra de las posibles utilidades de la ofuscación del código la encontramos en aplicaciones para dispositivos móviles. Estos dispositivos tienen una potencia de cálculo baja, interfaces de usuario pobres y memoria limitada. Mediante la ofuscación del código estamos optimizando y reduciendo el tamaño de las aplicaciones.

2.3. MEDIDA DE LA OFUSCACIÓN

Para tener una idea de la efectividad de los ofuscadores es necesario estudiar el proceso de ofuscación de acuerdo a una serie de medidas. En general la ofuscación puede ser medida como suma de lo siguiente:

- Carga de procesado (*Potency*): $\left(E(P)/E(P'\right)-1$.
- Flexibilidad (*Resilience*) para recuperar el código original (trivial, débil, fuerte, completa, en un sentido (*one way*)).
- Coste de la ofuscación.
- Ocultación de la ofuscación (*Stealth*).

Las medidas de ofuscación (*obfuscation metrics*) son diseñadas de acuerdo a las transformaciones y técnicas usadas por los ofuscadores disponibles en Internet. También atendiendo al uso que actualmente se da a los ofuscadores. Por ejemplo: ¿puede un ofuscador oscurecer el código con éxito?, pero también: ¿puede el código ofuscado ser descompilado?

Generalmente se suele acudir para el análisis de un ofuscador a métricas de software que tienen en cuenta el incremento de la complejidad en: longitud del programa (número de operandos y operadores), flujo de datos, número de predicados, anidamiento (en estructuras condicionales), estructura de datos (variables, *arrays*, etc.) y orientación a objetos (nivel de herencia, emparejamiento entre objetos, número de métodos desencadenados por otro, etc.).

2.4. TÉCNICAS DE OFUSCACIÓN

Un ofuscador es un programa que aplica transformaciones al código fuente de una aplicación en Java. Esta transformación de ofuscación podemos clasificarla en tres técnicas: Ofuscación de estructura, de control y de datos.

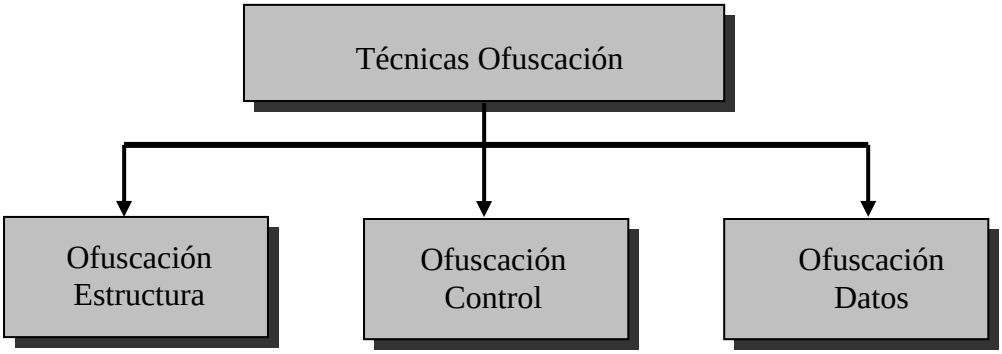


Figura 2.2. Técnicas de Ofuscación.

2.4.1. OFUSCACIÓN DE ESTRUCTURA

La ofuscación de estructura modifica la estructura del código mediante dos métodos básicos: renombrar identificadores y borrar información redundante. Esto hace que el código contenga menor información con el objetivo de evitar la ingeniería inversa. La mayoría de las ofuscaciones de estructura no pueden deshacerse porque usan una función de un solo sentido que renombra identificadores mediante símbolos aleatorios y borra comentarios, métodos sin uso e información redundante. Pese a esto, la ofuscación de estructura no evita los ataques de ingeniería inversa ya que el código aún puede ser estudiado y comprendido. Este tipo de ofuscación es la más empleada y extendida. La mayoría de los ofuscadores Java utilizan esta técnica.

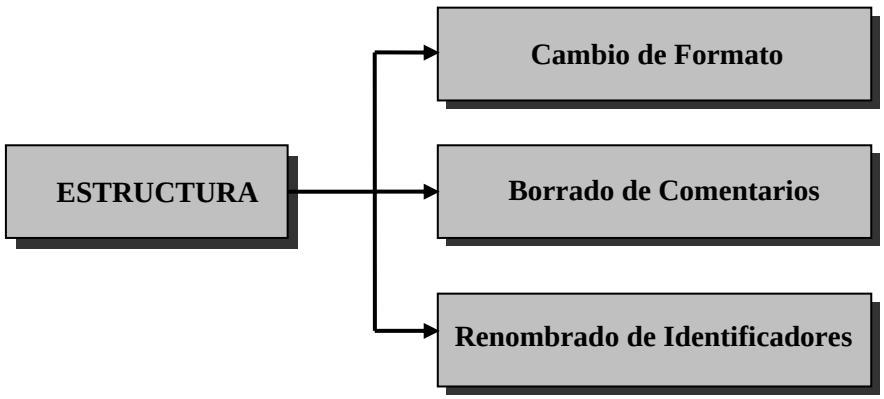


Figura 2.3. Ofuscación de Estructura.

2.4.2. OFUSCACIÓN DE CONTROL

La ofuscación de control cambia el flujo de control del código fuente del programa. Esta técnica de ofuscación podemos clasificarla en:

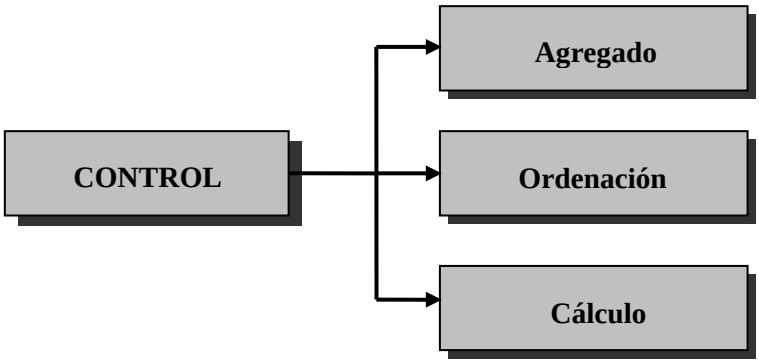


Figura 2.4. Ofuscación de Control.

2.4.2.1. Agregado

- Inserción de predicados opacos. Un predicado P se considera opaco si su valor es conocido durante el proceso de ofuscación. P^T (siempre evaluado como verdad), P^F (falso), $P^?$ (evaluado como cierto algunas veces o falso en otras ocasiones). Estos predicados opacos pueden introducirse en la estructura de control de flujo mediante los tres métodos expuestos anteriormente. En la figura 2.5 (a) el bloque [A;B] es dividido mediante la inserción de un predicado cierto P^T que hace que parezca que B solamente se ejecuta solamente en algunos casos. En la figura 2.5 (b) B se divide en dos versiones ofuscadas diferentes: B y B'. El predicado opaco $P^?$ selecciona uno de los dos en ejecución. En la figura 2.5 (c) P^T siempre selecciona B sobre B_{Bug}, una versión *bug* (errónea) de B.

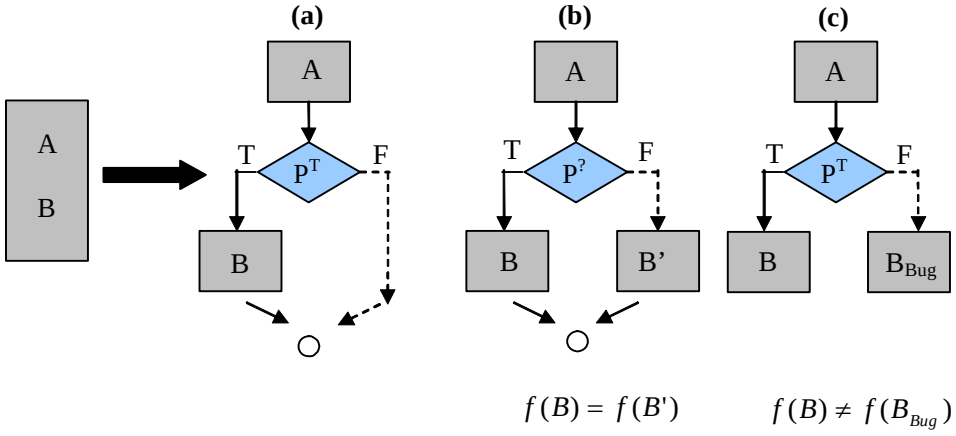


Figura 2.5. Introducción de predicados opacos.

- *Inlining*, que borra toda huella de abstracción de un método. *Outlining*, que crea métodos mediante secciones de código no relacionadas, las cuales añaden un nivel de fingida abstracción.

- Interpolación de métodos. Útil cuando algunos parámetros son del mismo tipo.

- Clonación de métodos. Se realizan múltiples pero diferentes versiones ofuscadas de de un método.

- Transformaciones en bucles.

2.4.2.2. Ordenación

- Ofuscación mediante control del orden, altera el orden de ejecución de sentencias.

2.4.2.3. Cálculo

- Inserción de código irrelevante.

- Inserción de códigos con diferente ofuscación.

- Inserción de código erróneo (*buggy code*). Estos segmentos introducidos no afectan a la ejecución de las clases Java, pero si rompen el proceso de descompilación. Esta técnica es muy peligrosa ya que hay MVJ muy estrictas en la interpretación del código y detecten estos segmentos como puntos corruptos y no continúen la interpretación del resto del código.

- Utilización de *Goto*. El código fuente no lo admite pero el bytecode si. Se modifica el bytecode mediante la utilización de Goto en los bucles.

- Borrado de llamadas a librerías comunes.

2.4.3. OFUSCACIÓN DE DATOS

Tiene por objetivo el oscurecimiento de datos y estructuras de datos que pueda haber en el código de nuestra aplicación. La ofuscación de datos rompe las estructuras de datos usadas en el código y las encripta literalmente. Tenemos diferentes métodos para ofuscar las estructuras. Ver figura 2.6.

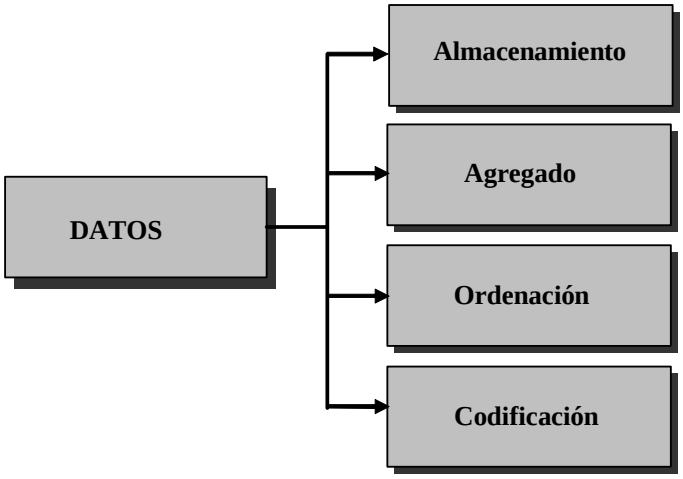


Figura 2.6. Ofuscación de Datos.

2.4.3.1. Almacenamiento y Codificación

- División de variables. Por ejemplo, separar una variable *boolean* en dos o más variables: $v = [(v_1, v_2, ..., v_k)]$. Esto incrementa el procesado de la aplicación. Ver tabla 2.1.

- Convertir especificadores de acceso *static* en *procedural*. Aumenta la carga de procesado y la flexibilidad. Ver tabla 2.2.

Original	***Ofuscado***
bool a,b,c	short a1, a2, a3, b1, b2, c1, c2
a = true, b = false, c = true	a1 = a2 = a3 = true, ……
c= a & b	c1 = ((a1^a2) ^a3) & (b1^b2)) c1=c2

Tabla 2.1. Ejemplo de división de variables.

Original	***Ofuscado***
String t = "Red"	String retStr (int i) { String s; s[1]="R"; s[2]="e"; s[3]="d";}

Tabla 2.2. Conversión Static en Procedural.

- Cambiar Codificación.

- Cambiar el periodo de vida de variables.

2.4.3.2. Agregado y Reordenación

- Redimensionar arrays. Por ejemplo, incrementando (*folding*) o reduciendo (*flattening*) las dimensiones del *array*. Aumenta o disminuye la carga de procesamiento respectivamente. Ver tabla 2.3.

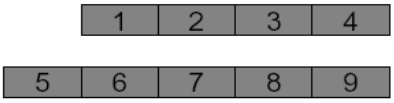
Original	***Ofuscado***
	
	

Tabla 2.3. Redimensionado de arrays.

- Clonar métodos. Por ejemplo, crear diferentes versiones de un método aplicando distintos tipos de ofuscaciones al código original.

- Modificar relaciones de herencia. Por ejemplo, dividir una clase (figura 2.7) o generar más clases extendiendo el árbol jerárquico de herencia (figura 2.8).

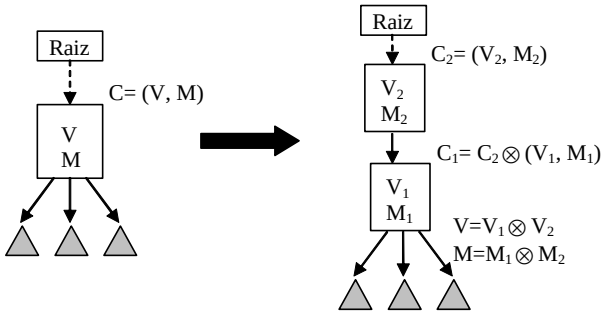


Figura 2.7. División de una clase.

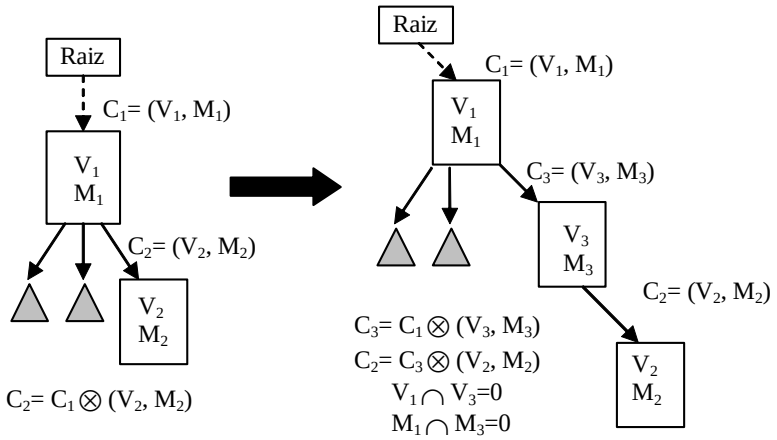


Figura 2.8. Generación de clase.

2.4.4. CONCLUSIONES

Todo esto hace que los códigos ofuscados sean prácticamente imposibles de descompilar.

Tras el tratamiento llevado a cabo por un ofuscador, los descompiladores siguen generando código fuente a partir del Java *bytecode*, pero este código no es como el código fuente original, siendo totalmente incomprensible y haciendo la labor de la ingeniería inversa muy difícil.

Además no existe ningún proceso de ingeniería inversa capaz de deshacer los cambios realizados por un ofuscador, de manera automática.

Otro beneficio que se obtiene al aplicar la ofuscación sobre nuestro código Java, es que se reduce considerablemente el tamaño de las clases, ya que la ofuscación elimina, como se ha comentado anteriormente, toda la información innecesaria y reemplaza los nombres largos puestos por los programadores para una mejor comprensión por otros mucho más cortos (*class a*, *int c*, etc.).

Esta reducción en el tamaño de las clases se traduce en una carga más rápida de los *applets*, ya que la transmisión de estos vía Internet, se realiza en un tiempo menor.

Esto también supone una gran ventaja para su utilización en programas o aplicaciones para dispositivos móviles, para los cuales tenemos unos requerimientos mas estrictos en cuanto a me memoria y capacidad.

2.5. PROCESO DE OFUSCACIÓN

Los ofuscadores pueden trabajar de dos maneras según realicen el proceso de ofuscación:

1) Una manera es empezar la ofuscación partiendo de la clase principal (o su equivalente en los *applets* y realizar un árbol de las clases accesibles desde la clase principal y aplicar la ofuscación en este orden. Esto se conoce como ofuscación con entrada de punto único. Este método es bastante rudimentario, y muy limitado, ya que código con múltiples entradas como pueden ser aplicaciones un tanto complejas, o *Applets*, o *JavaBeans,* o si el código es estructurado para trabajar como librería Java, la ofuscación no se realiza en todo el código.

2) Otro método es aplicar la ofuscación sobre todo el código java, permitiendo la selección de las clases sobre las que se aplicara ofuscación y en que grado se va a llevar a cabo. Es lo que se conoce como ofuscación con múltiples puntos de entrada.

2.6. MÉTODO DE ANALISIS PARA OFUSCACIÓN

2.6.1. DISEÑO DE MÉTRICA DE OFUSCACIÓN

Como hemos estudiado en anteriores apartados tenemos tres técnicas de ofuscación: ofuscación de Estructura, ofuscación de Control y ofuscación de Datos. Con objeto de comparar la efectividad de los ofuscadores encontrados en la red vamos a diseñar una métrica que nos permita comprobar dicha efectividad, desde el punto de vista de las técnicas mencionadas. Relacionamos así teoría con el trabajo de medida de ofuscación llevada a cabo por cada ofuscador.

Nuestra métrica de ofuscación será de la forma:

OBJETOS	CONTENIDO		PUNTUACIÓN
OFUSCACIÓN DE LAYOUT	Renombrar Identificadores	Public	1
		Protected	1
		Private	1
	Borrado de Número de Líneas		1
OFUSCACIÓN DE DATOS	Encriptado de String		1
	Desestructuracón de Datos		1
OFUSCACIÓN DE CONTROL	Transformaciones en Sentencias y Bucles		1
OFUSCACIÓN DE PROGRAMA			1
DESCOMPILACIÓN DE CÓDIGO OFUSCADO			1
DESCOMPILACIÓN DE PROGRAMA OFUSCADO			1
PUNTUACIÓN TOTAL			10

Tabla 2.4. Métrica para evaluación de ofuscadores.

La primera parte, ofuscación de Layout, incluye “Renombrar Identificadores” que contiene la modificación de identificadores *public*, *protected* y *private*; y “Borrado de Número de Líneas”.

La segunda parte, ofuscación de Datos, tiene en cuenta “Encriptado de *String*” y “Desestructuración de Datos” como el redimensionado de un *array*.

En la tercera parte, ofuscación de Control, tenemos en cuenta transformaciones en el flujo de control de el código tales como transformaciones en la estructura de sentencias condicionales “*if….else*” y bucles “*while, for*”.

El cuarto apartado “Ofuscación de Programa” mide si un ofuscador puede oscurecer con éxito un programa complicado.

La quinta parte “Descompilación de Código Ofuscado” muestra si el *bytecode* ofuscado puede ser descompilado nuevamente.

Por último, en el apartado sexto “Descompilación de Programa Ofuscado” comprobamos si el *bytecode* de programas ofuscados puede ser descompilado.

Hemos escogido este diseño porque nos permitirá puntuar por separado los distintos tipos de ofuscación de manera sencilla. La puntuación máxima de la métrica es de “10”, un punto por cada apartado individual. La puntuación se otorga de la siguiente manera: “0” si para un apartado individual no se observa comportamiento de ofuscación en la salida. Por ejemplo, en el apartado de “Renombrar Identificadores” si tuviéramos un método *private* de nombre “*getArray*” y después del proceso de ofuscación, en la salida aparece aun como “*getArray*” se le asignaría un 0 a esa métrica individual.

En el apartado “Borrado de Número de Líneas” dentro de ofuscación de Datos, si encontramos que no hay en el *bytecode* ofuscado, puntuaremos con un “1”.

En la métrica individual “Encriptado de *String*” dentro de ofuscación de Datos, si el nombre del *String* ha sido encriptado en el código ofuscado se asignará un “1”.

Para el apartado “Desestructuración de Datos” tendremos en cuenta que, si un *array* de una dimensión en el código ofuscado aparece como un *array* de dos o más dimensiones se puntuará con un “1”.

En lo referente a ofuscación de control puntuaremos con un 1 en el caso de que el código ofuscado presente modificaciones en las sentencias “*if…else*” y bucles “*for* y *while*”.

Para el cuarto apartado “Ofuscación de Programa”, si el ofuscador puede oscurecer el código del programa (aplicación gran tamaño) con éxito asignaremos un 1. En caso contrario un “0”.

En “Descompilación de Código Ofuscado”, para ejemplos de código ofuscado si no podemos descompilarlo o no realiza la misma función entonces puntuaremos esta métrica individual con un “1”.

Por último, para el apartado “Descompilación de Programa Ofuscado” si no se consigue descompilar el programa ofuscado previamente, se asignará un “1”.

2.6.2. DISEÑO DE EJEMPLOS

Entorno
Windows XP.
Ordenador Personal.

Herramientas
J2SDK1.4.2_07.
Descompilador de Java, Descompilador JODE.

Los ejemplos han sido diseñados de acuerdo a la métrica de ofuscación. Tenemos dos ejemplos, Test1 y Test2. Son usados para medir las diferentes métricas individuales estudiadas anteriormente. Test1 mide el comportamiento de un ofuscador en lo referente a ofuscación de Estructura y ofuscación de Datos. El ejemplo Test2 mide el comportamiento en cuanto a ofuscación de Control.

Con el objeto de simplificar y ver como puntuar cada métrica individual hemos listado los objetos de cada ejemplo de acuerdo a la métrica diseñada en la sección anterior (2.6.1. Diseño de Métrica de Ofuscación). Ver tabla 2.5.

MÉTRICA		**EJEMPLO 1**		**EJEMPLO 2**
		TEST1.CLASS	*HOLA.CLASS*	*TEST2.CLASS*
Public	**Clase**			
	Métodos	Main	GetHola, Hola	
	Campos	hello		
Protected	**Métodos**	pruebaOfus		
	Campos	aux		
Private	**Métodos**	getVector		
	Campos	vector		
Borrado Numero Linea		Numero de Linea		
Encriptado de String		PRUEBA DE OFUSCACION		
Desestructuracion de Datos		vector[]		
Transformacion Bucles y Sentencias Condicionales				if…else/ for…/ while...

Tabla 2.5. Identificadores originales de los objetos bajo observación para la métrica de ofuscación.

El proceso que se ha llevado a cabo para el diseño de los ejemplos ha sido el siguiente:

- En primer lugar escribir el código Java de los ejemplos.
- En segundo lugar compilación y ejecución de los archivos "*.class*" resultantes para asegurarnos de que funcionan apropiadamente. Hemos imprimido el *bytecode* ejecutando *javap* con las siguientes opciones: c (descompilar), p (incluir campos privados) y l (numero de línea y tablas locales).
- A continuación hemos descompilado los archivos "*.class*" mediante el descompilador JODE y comparado el código obtenido con el original.

- El siguiente paso ha sido la compilación del código descompilado para cerciorarnos de que realiza la misma funcionalidad.

- Finalmente hemos comparado los resultados de cada paso para comprender como aparecen los ejemplos antes de ser ofuscados.

2.6.2.1. EJEMPLO 1

El código fuente del ejemplo 1 viene dado en el fichero "*Test1.java*" (794 *bytes*). Tiene dos clases: "*Test1.class*" (1.102 *bytes*) y "*Hola.class*" (284 *bytes*). "*Test1.class*" es la clase principal.

Código Fuente

A continuación se detalla el código fuente de nuestro primer ejemplo "*Test1.java*".

Programa 2.1. Código Fuente Test1.java.

```
//Ejemplo1
//Proposito: Comprobar el comportamiento de un ofuscador
//con respecto a ofuscacion de layout y datos.
//Fecha realización: 11/3/2005.
//Autor: Angel Miralles Arevalo.

public class Test1{
	public Hola hello=new Hola();
	protected String aux="";
	private int[] vector = new int[10];
	public static void main(String args[]){
		new Test1();
	}
	public Test1(){
		System.out.println(hello.getHola("HOLA MUNDO"));
		System.out.println(pruebaOfus());
		System.out.println("Ejemplo="+getVector());
	}
	protected String pruebaOfus(){
		aux = "PRUEBA DE OFUSCACION";
		return aux;
	}
	private int getVector(){
		vector[0]=1;
		return vector[0];
	}
}

class Hola{
	public Hola(){
		int num=1;
	}
	public String getHola(String cadena){
		return cadena;
	}
}
```

Al compilar mediante J2SDK1.4.2_07 obtenemos las dos clases mencionadas anteriormente: "*Test1.class*" y "*Hola.class*". Ejecutando comprobamos el correcto funcionamiento de nuestra aplicación:

```
D:\Temp\prueba>java Test1
HOLA MUNDO
PRUEBA DE OFUSCACION
Ejemplo=1
```

Bytecode

Para obtener el bytecode de nuestro ejemplo 1 ejecutamos la aplicación *javap* con las opciones oportunas. Observar lo obtenido en programa 2.2 y programa 2.3.

Programa 2.2. Bytecode Test1.class.

```
D:\Temp\prueba>javap -c -p -l Test1
Compiled from "Test1.java"
public class Test1 extends java.lang.Object{
public Hola hello;

protected java.lang.String aux;

private int[] vector;

public static void main(java.lang.String[]);
  Code:
   0:   new     #1; //class Test1
   3:   dup
   4:   invokespecial   #2; //Method "<init>":()V
   7:   pop
   8:   return

  LineNumberTable:
   line 12: 0
   line 13: 8

public Test1();
  Code:
   0:   aload_0
   1:   invokespecial   #3; //Method java/lang/Object."<init>":()V
   4:   aload_0
   5:   new     #4; //class Hola
   8:   dup
   9:   invokespecial   #5; //Method Hola."<init>":()V
   12:  putfield        #6; //Field hello:LHola;
   15:  aload_0
   16:  ldc     #7; //String
   18:  putfield        #8; //Field aux:Ljava/lang/String;
   21:  aload_0
   22:  bipush  10
   24:  newarray int
   26:  putfield  #9; //Field vector:[I
   29:  getstatic #10; //Field java/lang/System.out:Ljava/io/PrintStream;
   32:  aload_0
   33:  getfield        #6; //Field hello:LHola;
   36:  ldc     #11; //String HOLA MUNDO
   38:  invokevirtual#12; //Method
Hola.getHola:(Ljava/lang/String;)Ljava/lang/String;
   41:  invokevirtual   #13; //Method
java/io/PrintStream.println:(Ljava/lang/String;)V
   44:  getstatic #10; //Field java/lang/System.out:Ljava/io/PrintStream;
```

```
   47:  aload_0
   48:  invokevirtual   #14; //Method pruebaOfus:()Ljava/lang/String;
   51:invokevirtual #13; //Method
java/io/PrintStream.println:(Ljava/lang/String;)V
   54:  getstatic #10; //Field java/lang/System.out:Ljava/io/PrintStream;
   57:  new     #15; //class StringBuffer
   60:  dup
   61:  invokespecial   #16; //Method java/lang/StringBuffer."<init>":()V
   64:  ldc     #17; //String Ejemplo=
   66:  invokevirtual   #18; //Method
java/lang/StringBuffer.append:(Ljava/lang/String;)Ljava/lang/StringBuffer;
   69:  aload_0
   70:  invokespecial   #19; //Method getVector:()I
   73:  invokevirtual   #20; //Method
java/lang/StringBuffer.append:(I)Ljava/lang/StringBuffer;
   76:  invokevirtual   #21; //Method
java/lang/StringBuffer.toString:()Ljava/lang/String;
   79:  invokevirtual   #13; //Method
java/io/PrintStream.println:(Ljava/lang/String;)V
   82:  return

  LineNumberTable:
   line 14: 0
   line 8: 4
   line 9: 15
   line 10: 21
   line 15: 29
   line 16: 44
   line 17: 54
   line 18: 82

protected java.lang.String pruebaOfus();
  Code:
   0:   aload_0
   1:   ldc     #22; //String PRUEBA DE OFUSCACION
   3:   putfield        #8; //Field aux:Ljava/lang/String;
   6:   aload_0
   7:   getfield        #8; //Field aux:Ljava/lang/String;
   10:  areturn

  LineNumberTable:
   line 20: 0
   line 21: 6

private int getVector();
  Code:
   0:   aload_0
   1:   getfield        #9; //Field vector:[I
   4:   iconst_0
   5:   iconst_1
   6:   iastore
   7:   aload_0
   8:   getfield        #9; //Field vector:[I
   11:  iconst_0
   12:  iaload
   13:  ireturn
  LineNumberTable:
   line 24: 0
   line 25: 7
}
```

Programa 2.3. Bytecode Hola.class.

```
D:\Temp\prueba>javap -c -p -l Hola
Compiled from "Test1.java"
class Hola extends java.lang.Object{
```

```
public Hola();
  Code:
   0:   aload_0
   1:   invokespecial   #1; //Method java/lang/Object."<init>":()V
   4:   iconst_1
   5:   istore_1
   6:   return



  LineNumberTable:
   line 30: 0
   line 31: 4
   line 32: 6

public java.lang.String getHola(java.lang.String);
  Code:
   0:   aload_1
   1:   areturn

  LineNumberTable:
   line 34: 0

}
```

De los *bytecodes* generados podemos concluir que los nombres de las clases, métodos y campos son los mismos que en el código fuente original. Además el *bytecode* contiene el número de línea generada por el compilador.

Descompilación

Utilizando el descompilador JODE podemos comprobar como, mediante ingeniería inversa, es fácil obtener el código fuente original de nuestras aplicaciones a partir de los *bytecodes*. Ver programa 2.4 y programa 2.5.

Programa 2.4. Decompilación de Test1.class mediante JODE.

```
/* Test1 - Decompiled by JODE
 * Visit http://jode.sourceforge.net/
 */

public class Test1
{
    public Hola hello = new Hola();
    protected String aux = "";
    private int[] vector = new int[10];

    public static void main(String[] strings) {
       new Test1();
    }

    public Test1() {
       System.out.println(hello.getHola("HOLA MUNDO"));
       System.out.println(pruebaOfus());
       System.out.println("Ejemplo=" + getVector());
    }

    protected String pruebaOfus() {
       aux = "PRUEBA DE OFUSCACION";
       return aux;
    }
```

```
    private int getVector() {
       vector[0] = 1;
       return vector[0];
    }
}
```

Programa 2.5. Decompilación de Hola.class mediante JODE.

```
/* Hola - Decompiled by JODE
 * Visit http://jode.sourceforge.net/
 */

class Hola
{
    public Hola() {
       boolean bool = true;
    }

    public String getHola(String string) {
       return string;
    }
}
```

Comparando con los códigos fuentes originales observamos que lo único que no se recupera son los comentarios.

Renombrando los archivos descompilados (notar que el archivo "*.java"* ha de tener el mismo nombre que la clase principal del código fuente) y volviendo a compilarlos comprobamos que siguen realizando la misma función:

```
D:\Temp\prueba>java Test1
HOLA MUNDO
PRUEBA DE OFUSCACION
Ejemplo=1
```

Conclusiones

- Antes de ser ofuscado el código del ejemplo Test1 puede ser descompilado con éxito.
- Comparando código fuente y *bytecode*, ambos tienen los mismos identificadores.

- Al comparar código fuente original y descompilado observamos que son iguales pero no aparecen comentarios en el último.

- El código descompilado realiza la misma función que el original.

2.6.2.2. EJEMPLO 2

El código fuente del ejemplo 2 es "*Test2.java*" (634 *bytes*). Al compilar obtenemos un solo archivo de *bytecode* "*Test2.class*" (780 *bytes*).

Código Fuente

A continuación detallamos el código fuente de nuestro segundo ejemplo. No debemos olvidar que esta aplicación la diseñamos para comprobar el comportamiento de los ofuscadores con respecto a la ofuscación de control. Ver programa 2.6.

Programa 2.6. Código fuente de aplicación Test2.java.

```
//Ejemplo2: Test2.
//Proposito: Comprobar el comportamiento de un ofuscador
//con respecto a ofuscacion de control.
//Fecha realización: 11/3/2005.
//Autor: Angel Miralles Arevalo.

public class Test2{
	public static void main( String args[] ){
		int bucleNum=0;

		// test bucle for
		for ( int i=0; i<2; i++ )
			bucleNum++;

		// test sentencia condicional
		if ( bucleNum>1 )
			System.out.print("Numero de bucle > 1");
		else
			System.out.println("Numero de bucle <= 1");

		// test bucle while
		while ( bucleNum>1 )
			bucleNum=bucleNum-1;

		System.out.println();
		System.out.println("Numero de bucle =" +bucleNum );
		}
}
```

Al compilar mediante J2SDK1.4.2_07 obtenemos el bytecode "*Test2.class*". Ejecutando comprobamos el correcto funcionamiento de nuestra aplicación:

```
D:\Temp\prueba>java Test2
Numero de bucle > 1
Numero de bucle=1
```

Bytecode

Ejecutando la aplicación *javap* con las opciones adecuadas obtenemos el *bytecode* correspondiente al segundo ejemplo. Ver programa 2.7.

Programa 2.7. Bytecode Test2.class.

```
D:\Temp\prueba>javap -c -p -l Test2
Compiled from "Test2.java"
public class Test2 extends java.lang.Object{
public Test2();
  Code:
   0:   aload_0
   1:   invokespecial   #1; //Method java/lang/Object."<init>":()V
   4:   return

  LineNumberTable:
   line 7: 0

public static void main(java.lang.String[]);
  Code:
   0:   iconst_0
   1:   istore_1
   2:   iconst_0
   3:   istore_2
   4:   iload_2
   5:   iconst_2
   6:   if_icmpge       18
   9:   iinc    1, 1
   12:  iinc    2, 1
   15:  goto    4
   18:  iload_1
   19:  iconst_1
   20:  if_icmple       34
   23:  getstatic       #2; //Field java/lang/System.out:Ljava/io/PrintStream;
   26:  ldc     #3; //String Numero de bucle > 1
   28:  invokevirtual#4; //Method
java/io/PrintStream.print:(Ljava/lang/String;)V
   31:  goto    42
   34:  getstatic       #2; //Field java/lang/System.out:Ljava/io/PrintStream;
   37:  ldc     #5; //String Numero de bucle <= 1
   39:  invokevirtual   #6; //Method
java/io/PrintStream.println:(Ljava/lang/String;)V
   42:  iload_1
   43:  iconst_1
   44:  if_icmple       54
   47:  iload_1
   48:  iconst_1
   49:  isub
   50:  istore_1
   51:  goto    42
   54:  getstatic       #2; //Field java/lang/System.out:Ljava/io/PrintStream;
   57:  invokevirtual   #7; //Method java/io/PrintStream.println:()V
   60:  getstatic       #2; //Field java/lang/System.out:Ljava/io/PrintStream;
   63:  new     #8; //class StringBuffer
   66:  dup
   67:  invokespecial   #9; //Method java/lang/StringBuffer."<init>":()V
   70:  ldc     #10; //String Numero de bucle=
   72:  invokevirtual   #11; //Method
java/lang/StringBuffer.append:(Ljava/lang/String;)Ljava/lang/StringBuffer;
   75:  iload_1
   76:  invokevirtual   #12; //Method
java/lang/StringBuffer.append:(I)Ljava/lang/StringBuffer;
   79:  invokevirtual   #13; //Method
java/lang/StringBuffer.toString:()Ljava/lang/String;
   82:  invokevirtual   #6; //Method
java/io/PrintStream.println:(Ljava/lang/String;)V
   85:  return

  LineNumberTable:
   line 9: 0
   line 12: 2
   line 13: 9
   line 12: 12
```

```
   line 16: 18
   line 17: 23
   line 19: 34
   line 22: 42
   line 23: 47
   line 25: 54
   line 26: 60
   line 27: 85

}
```

Del *bytecode* generado podemos concluir que los nombres de las clases, métodos y campos son los mismos que en el código fuente original. Además el *bytecode* contiene el número de línea generada por el compilador.

Descompilación

Utilizando la herramienta de descompilación JODE obtenemos el siguiente código descompilado:

Programa 2.8. Decompilación de Test2.class mediante JODE.

```
/* Test2 - Decompiled by JODE
 * Visit http://jode.sourceforge.net/
 */

public class Test2
{
    public static void main(String[] strings) {
       int i = 0;
       for (int i_0_ = 0; i_0_ < 2; i_0_++)
           i++;
       if (i > 1)
           System.out.print("Numero de bucle > 1");
       else
           System.out.println("Numero de bucle <= 1");
       for (/**/; i > 1; i--) {
           /* empty */
       }
       System.out.println();
       System.out.println("Numero de bucle=" + i);
    }
}
```

A excepción de los comentarios, todo el código recuperado es igual al del código fuente original.

Renombrando los archivos descompilados (notar que el archivo "*.java*" ha de tener el mismo nombre que la clase principal del código fuente) y volviendo a compilarlos comprobamos que siguen realizando la misma función:

```
D:\Temp\prueba>java Test2
Numero de bucle > 1
Numero de bucle=1
```

Conclusiones

- Antes de ser ofuscado el código del ejemplo Test2 puede ser descompilado con éxito.
- Comparando código fuente y *bytecode*, ambos tienen los mismos identificadores.
- Al comparar código fuente original y el descompilado observamos que los bucles y sentencias condicionales aparecen en el mismo orden.
- El código descompilado realiza la misma función que el original.

3. PRUEBAS Y RESULTADOS

El trabajo que vamos a realizar en esta sección se basará en evaluar la efectividad de los ofuscadores en base a la métrica diseñada con anterioridad. Para cada ofuscador hemos llevado a cabo los siguientes pasos: Instalación, Procesado, Programa Lógico, Descompilación y Evaluación de la Métrica. En la figura 3.1 describimos mediante un diagrama de flujo el diseño experimental que vamos a seguir.

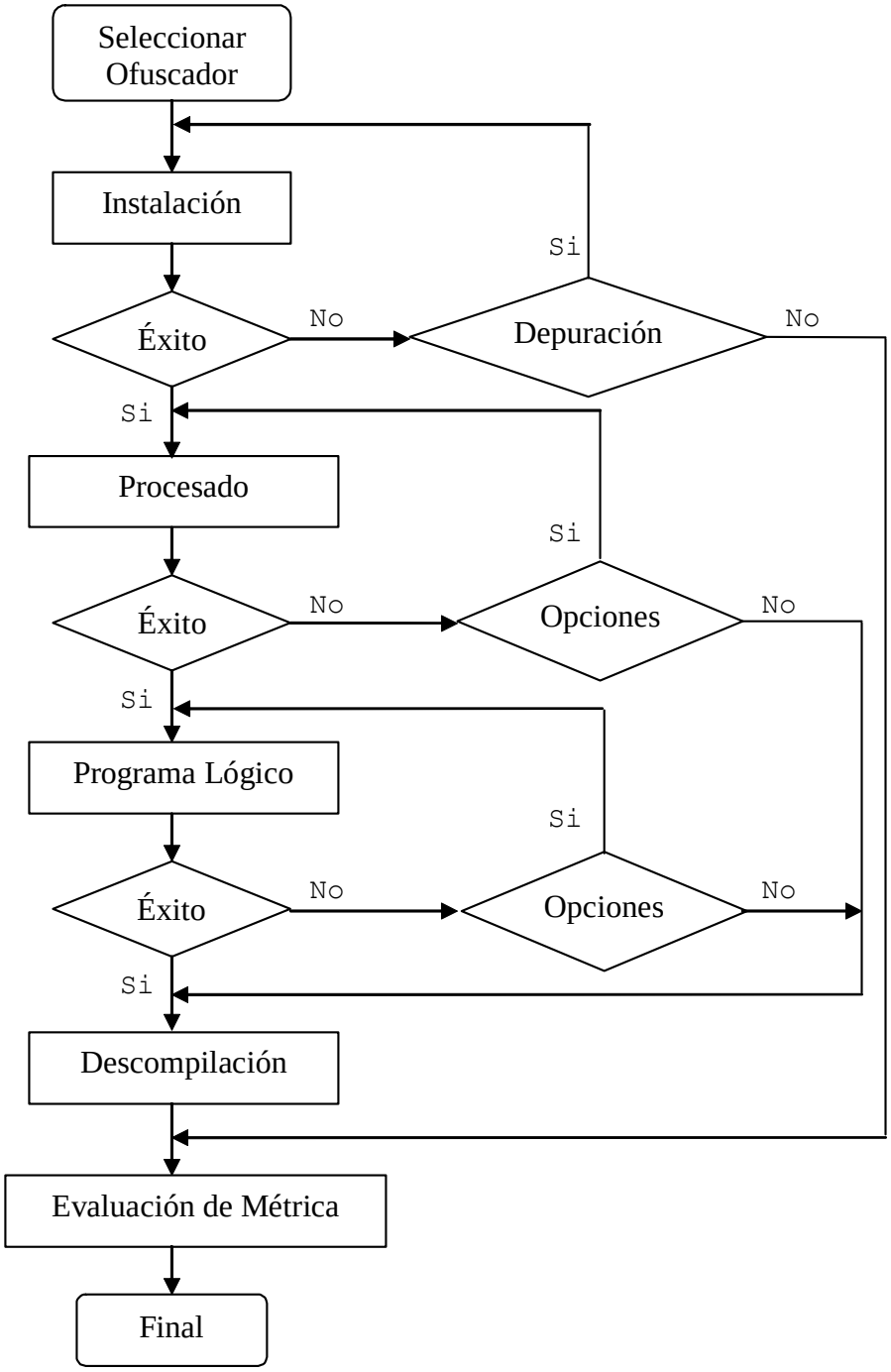


Figura 3.1. Diagrama de flujo del proceso de evaluación.

3.1. INSTALACIÓN

En este punto lo que se pretende es preparar las aplicaciones para su posterior testéo. El proceso lo vamos a dividir en tres pasos: Recopilación de Ofuscadores, Descarga e Instalación.

3.1.1. RECOPILACIÓN DE OFUSCADORES

En la siguiente tabla hemos recopilado la información que hemos encontrado sobre los ofuscadores que vamos a evaluar, así como sus características más destacables:

OFUSCADOR	VERSION	LICENCIA	CARACTERÍSTICAS
Proguard	v 3.2	Open Source	Optimiza y Ofusca mediante una plantilla de configuración. Proporciona soporte para Ant y J2ME.
Javaguard	v 1.0 Beta 4	Open Source	Ofuscador. Versión actualizada de Retroguard. Puede ejecutarse desde línea de comandos o mediante GUI.
Retroguard	v 2.0.1	Open Source	Ofuscador altamente configurable para bytecode Java con soporte para Java 2, reflexión y encriptado.
Jobfuscate	v 3.0	Shareware	Ofuscador de archivos .class Java. Herramienta en modo comando.
Jshrink	v 2.0.1	Comercial.	Borra código y datos no usados, ofusca identificadores en bytecode de Java.
Jzipper	v 1.0.9	Comercial.	Ofusca y empaqueta archivos .class.
Marvin	v 1.2b	Freeware	Borra clases innecesarias y ofusca aplicaciones, applets y servlets.Soporta encriptado de String.
Smokescreen	v 3.41	Shareware	Ofuscador Java. Soporta renombramiento selectivo, borrado de métodos y campos sin uso. Puede usrase con o sin GUI.
Yguard	v 1.3.2	Freeware, Open Source	Ofuscador de bytecode configurable para ejecutarse mediante Ant scripts
Zelix KlassMaster	v 4.3	Comercial, versión de evaluación.	Ofuscador de Java con ofuscación de control de flujo, encriptado, integración con Ant, J2ME pluggin y capacidad de decompilación.
CafeBabe	v 1.2.7a	Open Source	Desamblador de código Java que además permite compactar y ofuscar.
JoGa	v 0.1(alpha)	Open Source	Es básicamente un optimizador de código Java. Permite reducir el tamaño de aplicaciones, applets y apis.

Tabla 3.1. Recopilación de ofuscadores disponibles en internet.

Comprobamos la gran variedad de aplicaciones que permiten optimizar y ofuscar el código Java disponibles en internet. La mayoría aplicaciones comerciales, de las que sólamente hemos podido conseguir una versión de evaluación, pero también muchas otras open source.

De las principales ventajas de cada aplicación, cabe destacar que la mayoría de los ofuscadores realizan ofuscación de datos, tal como renombrado de identificadores, borrado de información redundante, etc…Algunos ofrecen la posibilidad de ofuscación de flujo de control, así como encriptado. Pero todo esto lo veremos con más detalle cuando evaluemos cada aplicación por separado.

3.1.2. DESCARGA

En la siguiente tabla se pone de manifiesto los aspectos más relevantes de la descarga de las aplicaciones detalladas con anterioridad: Formato de descarga (tipo archivo), Tamaño de aplicación y Dirección de bajada (*URL*).

OFUSCADOR	FORMATO DE DESCARGA	TAMAÑO	URL
Proguard	proguard3.2.zip	1.489 bytes	http://proguard.sourceforge.net/
Javaguard	javaguard.rar	1.198 bytes	http://sourceforge.net/projects/javaguard/
Retroguard	retroguard-v2.0.1.zip	241 bytes	http://www.retrologic.com/retroguard-main.html
Jobfuscate	jobfuscate.exe	88 bytes	http://www.duckware.com/jobfuscate/index.html
Jshrink	jshrink.exe	197 bytes	http://www.e-t.com/jshrink.html
Jzipper	jzip_install_demo.zip	315 bytes	http://www.vegatech.net/jzipper
Marvin	marvinobfuscator1_2b.zip	125 bytes	http://www.drjava.de/obfuscator/
Smokescreen	SmokescreenSetup341_Eval.zip	341 bytes	http://www.leesw.com/
Yguard	yguard-1.3.2.zip	146 bytes	http://www.yworks.com/en/products_yguard_about.htm
Zelix KlassMaster	ZKMEval.zip	1.032 bytes	http://www.zelix.com/klassmaster/
CafeBabe	CafeBabe-sources.zip	294 bytes	http://www.geocities.com/CapeCanaveral/Hall/2334/Programs/cafebabe.html
JoGa	joga.zip	342 bytes	http://www.nq4.de/

Tabla 3.2. Detalles de descarga de aplicaciones.

Destacar que no hemos tenido problemas con la descarga de ninguna de las aplicaciones. Para la descarga de algunas de las aplicaciones (las comerciales) tuvimos que registrarnos para poder recibir una versión de evaluación.

3.1.3. INSTALACIÓN

Entorno
Windows XP
Ordenador Personal. Pentium IV 2Ghz, 512Mb RAM.
J2SDK1.4.2_07.

Destacar que no tuvimos problemas a la hora de instalar todas las aplicaciones recopiladas. Para la instalación de cada aplicación se siguieron los manuales de usuario facilitados en las páginas web asociadas.

3.2. PROCESADO

El proceso de evaluación del comportamiento de los ofuscadores lo hemos denominado Procesado. Esta evaluación se realizará en base a pequeñas aplicaciones que hemos diseñado de acuerdo a la métrica especificada en la sección 2.6.1. Diseño de Métrica de Ofuscación, ejemplo 1 y 2. Recordemos que el ejemplo 1 trata de testear la efectividad de un ofuscador en lo referente a ofuscación de Estructura y Datos. El ejemplo 2 hace lo propio, referido a ofuscación de Control.

Para los 12 ofuscadores recopilados intentaremos un procesado de testeo y evaluación reflejado en el siguiente esquema:

Esquema 3.1. Proceso de evaluación.

```
Para cada ofuscador o {
       Para cada ejemplo i {
              Si i=2 y ofuscador no proporciona ofuscación de Control {
                     Especificar "No Disponible" N/A
              }
       Intentar ofuscar ejemplo i con ofuscador o con las opciones a de
       seguridad más elevadas
       Verficar que el programa aun realiza la misma función
       Si fallo {
              Probar con nivel de seguridad menor
              Repetir (o,a,i)
       }
       Si fallo persiste
              Devolver fallo
       Devolver éxito (o,a,i)
}
```

3.2.1. PROGUARD

3.2.1.1. Introducción

Es un compactador, optimizador y ofuscador para archivos "*.class*" Java. Puede detectar y borrar clases, miembros, métodos y atributos sin uso. Puede por tanto optimizar *bytecode* y borrar instrucciones no usadas.

Finalmente puede renombrar los identificadores de clases, miembros y métodos usando nombres cortos sin ningún sentido. El resultado "*.jar*" será de menor tamaño y más difícil de descompilar. Ver figura 3.2.

La principal ventaja con respecto a otros ofuscadores Java es probablemente su configuración compacta y sencilla. Mediante opciones en línea de comandos o con un archivo de configuración pueden realizarse fácilmente las utilidades expuestas anteriormente. Proguard es una herramienta de línea de comandos con una interfaz gráfica opcional. También se proporcionan *plugins* para Ant y J2ME Wireless Toolkit.

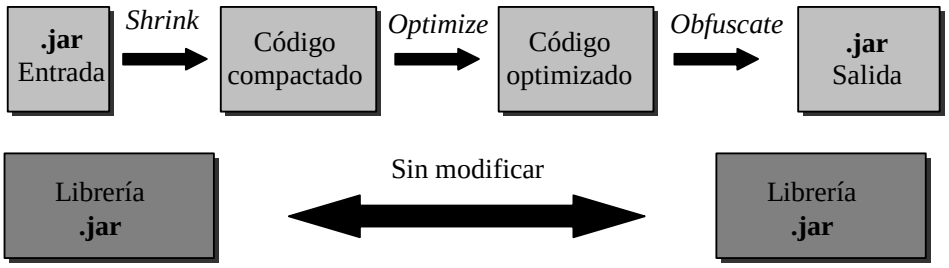


Figura 3.2. Proceso de compactación, optimizado y ofuscación.

Como entrada generalmente se le pasa un archivo "*.jar*" (o "*.wars*", "*.ears*", "*.zips*" o directorios). Se compacta, se optimiza y se ofusca dando como resultado o salida un archivo "*.jar*" (o "*.wars*", "*.ears*", "*.zips*" o directorios). Además requiere que se especifiquen librerías externas para resolver posibles dependencias de clases y herencia. Estas librerías permanecen inalteradas como puede observarse en la anterior figura.

Con objeto de determinar qué código tiene que preservarse y cuál ha de ser descartado u ofuscado, tenemos que especificar uno o más puntos de entrada en nuestro código. Estos puntos de entrada son típicamente clases que contienen el método principal, *applets*, *midlets*, etc.

En el paso de compactación Proguard empieza por estas semillas o puntos de entrada y determina recursivamente qué clases y qué miembros de clases son usados. Lo demás es descartado.

En el paso de optimización Proguard realiza acciones como por ejemplo: clases y métodos que no son puntos de entrada son convertidos a *final* y algunos métodos sufren *inlined*.

Por último en el proceso de ofuscación se renombran identificadores de clases y miembros de clases que no son puntos de entrada. Manteniéndose los identificadores de éstos últimos.

3.2.1.2. Ofuscación de Ejemplo 1

Los pasos que hemos seguido para comprobar la efectividad de la aplicación Proguard 3.2 en el proceso de ofuscación del ejemplo 1, han sido los siguientes:

1) Compilar el ejemplo 1 mediante J2SDK1.4.2_07, generándose los archivos: "*Test1.class*" (1.102 *bytes*) y "*Hola.class*" (284 *bytes*).

```
D:\Temp\prueba\javac Test1.java
```

2) Meter estos archivos en un contenedor "*Test1.jar*", que será el archivo de entrada para nuestra aplicación:

D:\Temp\prueba\jar cvf Test1.jar Tes1.class Hola.class

3) Ejecutar Proguard 3.2. mediante la siguiente instrucción:

D:\Temp\prueba>java -jar D:\Temp\proguard3.2\lib\proguard.jar @config.pro –verbose

Notar que hemos utilizado un archivo de configuración donde hemos indicado las opciones con las que debía operar Proguard "*config.pro*". La configuración se detalla a continuación en programa 3.1.

Programa 3.9. Listado de opciones.

```
-injars      Test1.jar
-outjars     Test1res.jar

-libraryjars    "C:\Archivos de programa\j2sdk1.4.2_07\jre\lib\rt.jar"

-printmapping   result.map

-overloadaggressively

-keep  public class Test1{
       public static void main(java.lang.String[]);
       }
```

4) Comprobamos que no ha ocurrido ninguna incidencia. Proguard no genera una *logfile* sino que muestra por salida estándar los pasos llevados a cabo y si ha ocurrido algún error. La salida se lista en el programa 3.2.

Programa 3.2. Salida por pantalla de las operaciones llevadas a cabo por Proguard.

```
ProGuard, version 3.2
Reading jars...
Reading program jar [Test1.jar]
Reading library jar [C:\Archivos de programa\j2sdk1.4.2_07\jre\lib\rt.jar]
Removing unused library classes...
    Original number of library classes: 5742
    Final number of library classes:    10
Shrinking...
Removing unused program classes and class elements...
    Original number of program classes: 2
    Final number of program classes:    2
Optimizing...
Shrinking...
Removing unused program classes and class elements...
    Original number of program classes: 2
    Final number of program classes:    2
Obfuscating...
Renaming program classes and class elements...
Printing mapping to [result.map]
Writing jars...
Preparing output jar [Test1res.jar]
Copying resources from program jar [Test1.jar]
```

Destacar los nombres del archivo de salida "*Test1res.jar*" y el archivo donde se imprime el proceso de renombramiento de clases y miembros de una clase "*result.map*".

5) Comprobamos que el código ofuscado sigue realizando la misma función. Para ello primeramente extraemos los *bytecodes* del contenedor de salida y posteriormente los ejecutamos:

```
D:\Temp\prueba\jar xf Test1res.jar
```

Destacar que obtenemos dos clases "*Test1.class*" (896 *bytes*) y "*a.class*" (187 bytes). Comprobamos que el tamaño de nuestros *bytecodes* se ha reducido. La clase Test1 no ha sufrido renombramiento porque Proguard requiere de un punto de entrada para ir ofuscando recursivamente desde esta semilla como ese ha explicado en la introducción del programa.

```
D:\Temp\prueba\java Test1
HOLA MUNDO
PRUEBA DE OFUSCACION
Ejemplo=1
```

Comprobamos que seguimos teniendo la misma funcionalidad.

6) A continuación obtenemos los bytecodes de las dos clases ofuscadas para poder compararlas con sus homólogas sin ofuscar y así poder evaluar nuestra métrica. Ver programa 3.3 y 3.4.

```
D:\Temp\prueba\javap –c –p –l Test1
```

Programa 3.3. Bytecode Test1.class.

```
Compiled from null
public class Test1 extends java.lang.Object{
public a a;

protected java.lang.String a;

private int[] a;

public static void main(java.lang.String[]);
  Code:
   0:   new     #1; //class Test1
   3:   invokespecial   #11; //Method "<init>":()V
   6:   return


public Test1();
  Code:
   0:   aload_0
   1:   invokespecial   #12; //Method java/lang/Object."<init>":()V
   4:   aload_0
   5:   new     #2; //class a
   8:   dup
   9:   invokespecial   #13; //Method a."<init>":()V
   12:  putfield        #7; //Field a:La;
   15:  aload_0
   16:  ldc     #22; //String
   18:  putfield        #8; //Field a:Ljava/lang/String;
```

```
   21:  aload_0
   22:  bipush  10
   24:  newarray int
   26:  putfield        #9; //Field a:[I
   29:  getstatic     #10; //Field java/lang/System.out:Ljava/io/PrintStream;
   32:  aload_0
   33:  getfield        #7; //Field a:La;
   36:  ldc     #24; //String HOLA MUNDO
   38:  invokevirtual #15; //Method a.a:(Ljava/lang/String;)Ljava/lang/String;
   41:  invokevirtual #20;//Method java/io/PrintStream.println:(Ljava/lang/St
ring;)V
   44:  getstatic  #10; //Field java/lang/System.out:Ljava/io/PrintStream;
   47:  aload_0
   48:  invokevirtual   #16; //Method a:()Ljava/lang/String;
   51:  invokevirtual #20; //Method java/io/PrintStream.println:(Ljava/lang/St
ring;)V
   54:  getstatic       #10; //Field
java/lang/System.out:Ljava/io/PrintStream;
   57:  new     #5; //class StringBuffer
   60:  dup
   61:  invokespecial   #14; //Method java/lang/StringBuffer."<init>":()V
   64:  ldc     #23; //String Ejemplo=
   66:  invokevirtual #18; //Method java/lang/StringBuffer.append:(Ljava/lang/
String;)Ljava/lang/StringBuffer;
   69:  aload_0
   70:  invokespecial   #17; //Method a:()I
   73:  invokevirtual #19; //Method java/lang/StringBuffer.append:(I)Ljava/lan
g/StringBuffer;
   76:  invokevirtual #21; //Method java/lang/StringBuffer.toString:()Ljava/la
ng/String;
   79:  invokevirtual #20; //Method java/io/PrintStream.println:(Ljava/lang/St
ring;)V
   82:  return



protected final java.lang.String a();
  Code:
   0:   aload_0
   1:   ldc     #25; //String PRUEBA DE OFUSCACION
   3:   putfield        #8; //Field a:Ljava/lang/String;
   6:   aload_0
   7:   getfield        #8; //Field a:Ljava/lang/String;
   10:  areturn



private int a();
  Code:
   0:   aload_0
   1:   getfield        #9; //Field a:[I
   4:   iconst_0
   5:   iconst_1
   6:   iastore
   7:   aload_0
   8:   getfield        #9; //Field a:[I
   11:  iconst_0
   12:  iaload
   13:  ireturn


}
```

Progarama 3.4. Bytecode de a.class.

```
Compiled from null
final class a extends java.lang.Object{
public a();
  Code:
   0:   aload_0
   1:   invokespecial   #3; //Method java/lang/Object."<init>":()V
   4:   iconst_0
   5:   istore_1
   6:   return


public final java.lang.String a(java.lang.String);
  Code:
   0:   aload_1
   1:   areturn
}
```

Comparando con los *bytecodes* originales podemos evaluar la métrica diseñada. Ver tabla 3.5.

MÉTRICA		EJEMPLO 1 (sin)		EJEMPLO 1(ofuscado)		PUNTUACIÓN
		Test1class	*Hola.class*	*Test1.class*	*a.class*	
Public	**Clase**					1
	Métodos	Main	GetHola, Hola	Main	a,a	
	Campos	hello		a		
Protected	**Métodos**	pruebaOfus		a		1
	Campos	aux		a		
Private	**Métodos**	getVector		a		1
	Campos	vector		a		
Borrado Numero Linea		Numero de Linea		Borrado		1
Encriptado de String		PRUEBA DE OFUSCACION		PRUEBA DE OFUSCACION		0
Desestructuracion de Datos		vector[]		a[]		0
Puntuación para Ofuscación de Layout y Datos						4

Tabla 3.5. Evaluación de ofuscación de Estructura y Datos para el ejemplo 1.

3.2.1.3. Ofuscación de Ejemplo 2

Proguard 3.2 no realiza ofuscación de Control de Flujo, no está especificado ni como opción en línea de comandos, ni tampoco mediante el uso de la interfaz gráfica que proporciona. Por tanto puntuamos con un “0” la métrica individual correspondiente.

3.2.1.4. Conclusiones

- Proguard puede realizar ofuscación de Estructura, renombrando identificadores de clases y miembros de clases *Public*, *Protected* y *Private*. Además borra toda información redundante, como comentarios, miembros si uso, etc.

- No es capaz de realizar ofuscación de Datos como puede verse en la tabla 3.5, es decir, no proporciona opciones de encriptado o desestructuración de datos.

- Después de ser ofuscado con las opciones más severas, nuestro código seguía realizando la misma función.

- No proporciona mecanismos para ofuscación de Control de Flujo.

- La interfaz gráfica que ofrece es bastante intuitiva y fácil de manejar. Hemos comprobado que se llega al mismo resultado.

3.2.2. RETROGUARD

3.2.2.1. Introducción

Retroguard es un ofuscador de *bytecode*, una herramienta diseñada para renombrar los identificadores comprensibles asignados por el programador en las clases Java, mediante *strings* sin sentido alguno, haciendo la labor de ingeniería inversa casi imposible. El resultado es un código más compacto, es decir, de menor tamaño y la confianza de que el código está a salvo de ingeniería inversa.

Las características que proporciona se detallan a continuación:

- Reduce el tamaño del *bytecode* (generalmente reducciones del 20-30%).

- Diseñado para ajustarse sin problemas a la construcción de aplicaciones Java.

- Permite una configuración completa del proceso de ofuscación.

- Soporta múltiples puntos de entrada en el código Java, permite el acceso a tantas aplicaciones, *applets*, *javabeans* y librerías como sean requeridas.

- Trabaja sobre archivos "*.jar*", el estándar de Java para el empaquetado de clases.

- El proceso de ofuscación es controlado por un lenguaje *script* flexible. Se proporciona una interfaz sencilla para la creación de *scrpits* (*Wizard*).

- Emplea *overloading* para nombres de métodos y miembros para conseguir una mayor seguridad.

3.2.2.2. Ofuscación de Ejemplo 1

Los pasos que hemos seguido para comprobar la efectividad de la aplicación Retroguard han sido los siguientes:

1) Generar el script de configuración mediante *Wizard*.

```
D:\Temp\retroguard-v2.0.1\src-dist\javac RGgui.java
D:\Temp\retroguard-v2.0.1\src-dist\java Rggui
```

Al ejecutar el *Wizard*, buscando siempre una configuración para realizar la ofuscación más agresiva posible, obtenemos el siguiente script. Ver programa 3.5.

Programa 3.5. Script de configuración para Retroguard.

```
# Automatically generated script for RetroGuard bytecode obfuscator.
# To be used with Java JAR-file: D:\Temp\prueba\Test1.jar
# 18-mar-2005 9:41:45
#
.class Test1
.method Test1/main ([Ljava/lang/String;)V
```

Notar que hemos preservado el nombre de la clase que contiene el método principal. Esto no puede evitarse porque Retroguard requiere al igual que Proguard de una semilla o punto de entrada desde el cual ir aplicando la ofuscación recursivamente.

2) Ejecutar Retroguard.

```
java -jar D:\Temp\retroguard-v2.0.1\retroguard.jar Test1.jar Test1res.jar script.rgs retro.log
```

No se producen incidencias a destacar. Para comprobarlo editamos el *logfile*. Además en este archivo de incidencias se puede observar el proceso de renombrado llevado a cabo. Ver programa 3.6.

Programa 3.6. Logfile que refleja actuaciones durante el proceso de ofuscación.

```
# If this log is to be used for incremental obfuscation / patch generation,
# add any '.class', '.method', '.field' and '.attribute' restrictions here:

#-DO-NOT-EDIT-BELOW-THIS-LINE-----------------DO-NOT-EDIT-BELOW-THIS-LINE--
#
# RetroGuard Bytecode Obfuscator, v2.0.1, a product of Retrologic Systems -
www.retrologic.com
#
# Logfile created on Fri Mar 18 11:12:19 CET 2005
#
# Jar file to be obfuscated:           Test1.jar
# Target Jar file for obfuscated code: Test1res.jar
# RetroGuard Script file used:         script.rgs
#
# Memory in use after class data structure built: 283232 bytes
# Total memory available                        : 2031616 bytes
#
# Obfuscated name overloading frequency:
```

```
#  'a'       used 4 times (57%)
#  'if'      used 2 times (28%)
#  'do'      used 1 times (14%)
#  Other names (each used in <1% of mappings) used a total of 0 times (1%)
#
# Names reserved from obfuscation:
#
.class Test1
.method Test1/main ([Ljava/lang/String;)V
#
# Obfuscated name mappings (some of these may be unchanged due to polymorphism
constraints):
#
.field_map Test1/aux a
.field_map Test1/vector if
.field_map Test1/hello do
.method_map Test1/pruebaOfus ()Ljava/lang/String; a
.method_map Test1/getVector ()I if
.class_map Hola a
.method_map Hola/getHola (Ljava/lang/String;)Ljava/lang/String; a
```

3) Comprobamos que el código ofuscado sigue realizando la misma función. Para ello primeramente extraemos los bytecodes del contenedor de salida y posteriormente los ejecutamos:

D:\Temp\prueba\jar xf Test1res.jar

Destacar que obtenemos dos clases "*Test1.class*" (896 *bytes*) y "*a.class*" (187 *bytes*). Comprobamos que el tamaño de nuestros *bytecodes* se ha reducido. La clase Test1 no ha sufrido renombramiento porque Proguard requiere de un punto de entrada para ir ofuscando recursivamente desde esta semilla como ese ha explicado en la introducción del programa.

D:\Temp\prueba\java Test1
HOLA MUNDO
PRUEBA DE OFUSCACION
Ejemplo=1

4) A continuación comparamos los bytecodes ofuscados con sus homólogos sin ofuscar, obteniendo los programas 3.7 y 3.8. Para ello ejecutamos:

D:\Temp\prueba\javap –c –p –l Test1
D:\Temp\prueba\javap –c –p –l a

Programa 3.7. Bytecode Test1.class.

```
Compiled from null
public class Test1 extends java.lang.Object{
public a do;

protected java.lang.String a;

private int[] if;

public static void main(java.lang.String[]);
  Code:
```

```
   0:   new     #1; //class Test1
   3:   dup
   4:   invokespecial   #2; //Method "<init>":()V
   7:   pop
   8:   return


public Test1();
  Code:
   0:   aload_0
   1:   invokespecial   #3; //Method java/lang/Object."<init>":()V
   4:   aload_0
   5:   new     #4; //class a
   8:   dup
   9:   invokespecial   #5; //Method a."<init>":()V
   12:  putfield        #6; //Field do:La;
   15:  aload_0
   16:  ldc     #7; //String
   18:  putfield        #8; //Field a:Ljava/lang/String;
   21:  aload_0
   22:  bipush  10
   24:  newarray int
   26:  putfield        #9; //Field if:[I
   29:  getstatic   #10; //Field java/lang/System.out:Ljava/io/PrintStream;
   32:  aload_0
   33:  getfield        #6; //Field do:La;
   36:  ldc     #11; //String HOLA MUNDO
   38:  invokevirtual #12; //Method a.a:(Ljava/lang/String;)Ljava/lang/String;
   41:  invokevirtual #13; //Method java/io/PrintStream.println:(Ljava/lang/St
ring;)V
   44:  getstatic  #10; //Field java/lang/System.out:Ljava/io/PrintStream;
   47:  aload_0
   48:  invokevirtual   #14; //Method a:()Ljava/lang/String;
   51:  invokevirtual #13; //Method java/io/PrintStream.println:(Ljava/lang/St
ring;)V
   54:  getstatic   #10; //Field java/lang/System.out:Ljava/io/PrintStream;
   57:  new     #15; //class StringBuffer
   60:  dup
   61:  invokespecial   #16; //Method java/lang/StringBuffer."<init>":()V
   64:  ldc     #17; //String Ejemplo=
   66:  invokevirtual #18; //Method java/lang/StringBuffer.append:(Ljava/lang/
String;)Ljava/lang/StringBuffer;
   69:  aload_0
   70:  invokespecial   #19; //Method if:()I
   73:  invokevirtual #20; //Method java/lang/StringBuffer.append:(I)Ljava/lan
g/StringBuffer;
   76:  invokevirtual #21; //Method java/lang/StringBuffer.toString:()Ljava/la
ng/String;
   79:  invokevirtual #13; //Method java/io/PrintStream.println:(Ljava/lang/St
ring;)V
   82:  return


protected java.lang.String a();
  Code:
   0:   aload_0
   1:   ldc     #22; //String PRUEBA DE OFUSCACION
   3:   putfield        #8; //Field a:Ljava/lang/String;
   6:   aload_0
   7:   getfield        #8; //Field a:Ljava/lang/String;
   10:  areturn


private int if();
  Code:
   0:   aload_0
   1:   getfield        #9; //Field if:[I
   4:   iconst_0
```

```
   5:   iconst_1
   6:   iastore
   7:   aload_0
   8:   getfield        #9; //Field if:[I
   11:  iconst_0
   12:  iaload
   13:  ireturn
}
```

Programa 3.8. Bytecode a.class.

```
Compiled from null
class a extends java.lang.Object{
public a();
  Code:
   0:   aload_0
   1:   invokespecial   #1; //Method java/lang/Object."<init>":()V
   4:   iconst_1
   5:   istore_1
   6:   return


public java.lang.String a(java.lang.String);
  Code:
   0:   aload_1
   1:   areturn
}
```

Comprando los bytecodes ofuscados obtenidos con los códigos originales podemos evaluar la métrica diseñada. Ver tabla 3.6.

MÉTRICA		EJEMPLO 1 (sin)		EJEMPLO 1(ofuscado)		PUNTUACIÓN
		Test1class	*Hola.class*	*Test1.class*	*a.class*	
Public	**Clase**					1
	Métodos	Main	GetHola, Hola	Main	a,a	
	Campos	hello		do		
Protected	**Métodos**	pruebaOfus		a		1
	Campos	aux		a		
Private	**Métodos**	getVector		if		1
	Campos	vector		if		
Borrado Numero Linea		Numero de Linea		Borrado		1
Encriptado de String		PRUEBA DE OFUSCACION		PRUEBA DE OFUSCACION		0
Desestructuracion de Datos		vector[]		if[]		0
Puntuación para Ofuscación de Layout y Datos						4

Tabla 3.6. Evaluación de ofuscación de Layout y Datos para el ejemplo 1.

3.2.2.3. Ofuscación de Ejemplo 2

Retroguard v2.0.1 no realiza ofuscación de Control de Flujo. Es una opción no disponible.

3.2.2.4. Conclusiones

- Retroguard puede realizar ofuscación de Estructura, renombrando identificadores de clases y miembros de clases *Public*, *Protected* y *Private*. Además borra toda información redundante, como comentarios, miembros si uso, etc.

- No es capaz de realizar ofuscación de Datos como puede verse en la tabla 3.6, es decir, no proporciona opciones de encriptado o desestructuración de datos.

- Después de ser ofuscado con las opciones más severas, nuestro código seguía realizando la misma función.

- No proporciona mecanismos para ofuscación de Control de Flujo.

- Aunque ha obtenido una puntuación igual a la de Proguard, si comparamos los *bytecodes* obtenidos en ambos casos, comprobamos que la reducción en el tamaño es mayor al utilizar Proguard.

- Además hemos de destacar que Retroguard se ejecuta en línea de comandos si ninguna opción permitida. No proporciona interfaz gráfica (*GUI*).

3.2.3. JAVAGUARD

3.2.3.1. Introducción

Ofuscador muy similar a la aplicación anterior (Retroguard). Herramienta de ofuscación desde línea de comandos. No muy intuitivo ya que no se proporciona manual de usuario en la página asociada.

3.2.3.2. Ofuscación de Ejemplo 1

Con objeto de evaluar los apartados de la métrica diseña referentes a ofuscación de Estructura y Datos, hemos llevado a cabo los siguientes pasos:

1) Partimos del archivo "*Test1.jar*" (1.425 *bytes*), que contiene las clases de nuestro ejemplo 1. Ya vimos en casos anteriores cómo generarlo. Desde línea de comandos ejecutamos JavaGuard:

```
java JavaGuard –i Test1.jar -o Test1res.jar –l logfile.log –v –s script
```

Obtenemos el archivo de salida "*Test1res.jar*" (1.414 *bytes*), así como un archivo donde se reflejan las incidencias del proceso de ofuscación (*logfile*). Ver programa 3.9.

Programa 3.9. Archivo Logfile generado por JavaGuard.

```
# If this log is to be used for incremental obfuscation / patch generation,
# add any '.class', '.method', '.field' and '.attribute' restrictions here:

#-DO-NOT-EDIT-BELOW-THIS-LINE-----------------DO-NOT-EDIT-BELOW-THIS-LINE--
#
# JavaGuard Bytecode Obfuscator, version 1.0beta4
#
# Logfile created on Wed Apr 06 16:19:11 CEST 2005
#
# Input taken for obfuscation: Test1.jar
# Output Jar file:             Test1res.jar
# JavaGuard script file used:          script

# Memory in use after class data structure built: 213280 bytes
# Total memory available                        : 2031616 bytes

# Obfuscated name overloading frequency:
#
#  'a'       used 4 times (50%)
#  'if'      used 2 times (25%)
#  'b'       used 1 times (12%)
#  'do'      used 1 times (12%)
#  Other names (each used in <1% of mappings) used a total of 0 times (1%)

# The following fields and methods are used in serializable classes
# and are checked whether they should be marked for retention or not:

# Full list of names reserved from obfuscation:
#
.method Test1/main ([Ljava/lang/String;)V
#
# Obfuscated name mappings (some of these may be unchanged due to polymorphism
# constraints):
#
.class_map   Hola a
.method_map  Hola/getHola (Ljava/lang/String;)Ljava/lang/String; a
.class_map   Test1 b
.field_map   Test1/aux a
.field_map   Test1/hello if
.field_map   Test1/vector do
.method_map  Test1/getVector ()I a
.method_map  Test1/pruebaOfus ()Ljava/lang/String; if
```

Destacar que hemos empleado un *script* para especificar la exclusión del método principal del proceso de ofuscación. En caso contrario obtendríamos un *bytecode* que daría error al interpretarse ya que no se encontraría el método principal (*main*).

2) A continuación comprobamos que el código ofuscado sigue realizando la misma función. Para ello, en primer lugar extraemos los bytecodes del archivo de salida:

```
D:\Temp\prueba\jar xf Test1res.jar
```

Obtenemos dos clases: "*a.class*" (199 *bytes*) y "*b.class*" (928 *bytes*). Comprobamos que hemos reducido el tamaño de los bytecodes con respecto a los originales.

```
D:\Temp\prueba\java b
HOLA MUNDO
PRUEBA DE OFUSCACION
Ejemplo=1
```

Efectivamente el código ofuscado sigue realizando la misma función.

3) El siguiente paso sería la evaluación del proceso de ofuscación llevado a cabo por Javaguard. Para esto comparamos los bytecodes obtenidos tras el proceso de ofuscación con los originales. Ver programas 3.10 y 3.11.

```
D:\Temp\prueba\javap –c –p –l b
D:\Temp\prueba\javap –c –p –l a
```

Programa 3.10. Bytecode b.class.

```
public class b extends java.lang.Object{
public a if;

protected java.lang.String a;

private int[] do;

public static void main(java.lang.String[]);
  Code:
   0:   new     #1; //class b
   3:   dup
   4:   invokespecial   #2; //Method "<init>":()V
   7:   pop
   8:   return


public b();
  Code:
   0:   aload_0
   1:   invokespecial   #3; //Method java/lang/Object."<init>":()V
   4:   aload_0
   5:   new     #4; //class a
   8:   dup
   9:   invokespecial   #5; //Method a."<init>":()V
   12:  putfield        #6; //Field if:La;
   15:  aload_0
   16:  ldc     #7; //String
   18:  putfield        #8; //Field a:Ljava/lang/String;
   21:  aload_0
   22:  bipush  10
   24:  newarray int
   26:  putfield        #9; //Field do:[I
   29:  getstatic      #10; //Field java/lang/System.out:Ljava/io/PrintStream;
   32:  aload_0
   33:  getfield        #6; //Field if:La;
   36:  ldc     #11; //String HOLA MUNDO
   38:  invokevirtual #12; //Method a.a:(Ljava/lang/String;)Ljava/lang/String;
   41:  invokevirtual #13; //Method java/io/PrintStream.println:(Ljava/lang/St
ring;)V
   44:  getstatic      #10; //Field java/lang/System.out:Ljava/io/PrintStream;
   47:  aload_0
   48:  invokevirtual   #14; //Method if:()Ljava/lang/String;
   51:  invokevirtual #13; //Method java/io/PrintStream.println:(Ljava/lang/St
ring;)V
   54:  getstatic      #10; //Field java/lang/System.out:Ljava/io/PrintStream;
   57:  new     #15; //class StringBuffer
```

```
   60:  dup
   61:  invokespecial   #16; //Method java/lang/StringBuffer."<init>":()V
   64:  ldc     #17; //String Ejemplo=
   66:  invokevirtual #18; //Method java/lang/StringBuffer.append:(Ljava/lang/
String;)Ljava/lang/StringBuffer;
   69:  aload_0
   70:  invokespecial   #19; //Method a:()I
   73:  invokevirtual #20; //Method java/lang/StringBuffer.append:(I)Ljava/lan
g/StringBuffer;
   76:  invokevirtual #21; //Method java/lang/StringBuffer.toString:()Ljava/la
ng/String;
   79:  invokevirtual #13; //Method java/io/PrintStream.println:(Ljava/lang/St
ring;)V
   82:  return


protected java.lang.String if();
  Code:
   0:   aload_0
   1:   ldc     #22; //String PRUEBA DE OFUSCACION
   3:   putfield        #8; //Field a:Ljava/lang/String;
   6:   aload_0
   7:   getfield        #8; //Field a:Ljava/lang/String;
   10:  areturn

private int a();
  Code:
   0:   aload_0
   1:   getfield        #9; //Field do:[I
   4:   iconst_0
   5:   iconst_1
   6:   iastore
   7:   aload_0
   8:   getfield        #9; //Field do:[I
   11:  iconst_0
   12:  iaload
   13:  ireturn
}
```

Programa 3.11. Bytecode a.class.

```
class a extends java.lang.Object{
public a();
  Code:
   0:   aload_0
   1:   invokespecial   #1; //Method java/lang/Object."<init>":()V
   4:   iconst_1
   5:   istore_1
   6:   return


public java.lang.String a(java.lang.String);
  Code:
   0:   aload_1
   1:   areturn
}
```

Después de comparar podemos hacer la evaluación de la métrica diseñada. Ver tabla 3.7.

MÉTRICA		EJEMPLO 1 (sin)		EJEMPLO 1(ofuscado)		PUNTUACIÓN
		Test1class	*Hola.class*	*b.class*	*a.class*	
Public	**Clase**					1
	Métodos	main	GetHola, Hola	main	a,a	
	Campos	hello		if		
Protected	**Métodos**	pruebaOfus		if		1
	Campos	aux		a		
Private	**Métodos**	getVector		a		1
	Campos	vector		do		
Borrado Numero Linea		Numero de Linea		Borrado		1
Encriptado de String		PRUEBA DE OFUSCACION		PRUEBA DE OFUSCACION		0
Desestructuracion de Datos		vector[]		do[]		0
Puntuación para Ofuscación de Layout y Datos						4

Tabla 3.7. Evaluación de ofuscación de Estructura y Datos para el ejemplo 1.

3.2.3.3. Ofuscación de Ejemplo 2

Javaguard no realiza ofuscación de Control de Flujo. Es una opción no disponible. Por tanto puntuamos con un 0 la métrica individual correspondiente.

3.2.3.4. Conclusiones

- Javaguard puede realizar ofuscación de Estructura, renombrando identificadores de clases y miembros de clases *Public*, *Protected* y *Private*. Además borra toda información redundante, como comentarios, miembros si uso, etc.

- No es capaz de realizar ofuscación de Datos como puede verse en la tabla 3.7, es decir, no proporciona opciones de encriptado o desestructuración de datos.

- Después de ser ofuscado con las opciones mas severas que permite, nuestro código seguía realizando la misma función.

- No proporciona mecanismos para ofuscación de Control de Flujo.

- Como punto negativo de la aplicación podríamos destacar que no se proporciona manual de usuario. Aunque su utilización es similar al de otras aplicaciones como por ejemplo Retroguard.

- Es una herramienta sin interfaz gráfica.

3.2.4. JSHRINK

3.2.4.1. Introducción

Jshrink extrae el mínimo conjunto de clases necesarias para una aplicación, borra datos y código que no se usa, ofusca nombre simbólicos, finaliza el código para una ejecución óptima y almacena el resultado en un archivo "*.jar*".

Jshrink reduce el tamaño de la aplicación entorno al 30-40%. El código ofuscado es mucho más difícil de comprender al descompilar. Lo que a primera vista parecen nombres sin sentido en el código ofuscado son a menudo nombres reutilizados de sistemas, una técnica conocida como reciclado semántico.

Algunas características que ofrece esta aplicación se detallan a continuación:

- Renombrado (o eliminación) de campos y métodos *Public* (además de en *Private*, *Package*, *Protected*).
- Eliminación de clases sin uso.
- Renombrado de clases.
- Reciclado semántico de nombres compuestos.
- Incremento de opciones sobre el proceso de ofuscación.
- Una interfaz gráfica mejorada y de fácil uso.
- Descompilador integrado.
- Proporciona integración para J2ME MIDP.
- Manejo automático de clases *ClassforName*().
- Automatización mediante *scriptfile*.
- Encriptado de cadenas.
- Salida como archivo "*.jar*" o "*.exe*".

3.2.4.2. Ofuscación de Ejemplo 1

Para la ofuscación de nuestro ejemplo hemos llevado a cabo los siguientes pasos:

1) Generar el *script* con las opciones adecuadas para llevar a cabo la ofuscación más agresiva posible. Consultando el manual de usuario comprobamos que este *script* podemos crearlo mediante la interfaz gráfica que proporciona la aplicación o escribirlo directamente, ver el programa 3.12:

Programa 3.12. Script de configuración para Jshrink.

```
-classpath C:\Archivos de programa\Java\j2re1.4.2_07\lib\rt.jar
-l
-noForName
-noGetMethod
-noNativePreserve
-stringEncrypt
D:\Temp\prueba\Test1.jar
```

2) Ejecutar Jshrink mediante la instrucción:

```
D:\Temp\prueba>java -jar D:\Temp\jshrink\jshrink.exe -script Jshrink -o  Test12res.jar
```

No se producen incidencias a destacar, ni errores ni *warnings*. Hemos especificado la opción "*-l*" mediante la cual obtenemos por salida estándar una tabla donde se muestra el renombramiento de clases y miembros de clases llevado a cabo, así como una estadística de la reducción en el tamaño de los archivos "*.class*". Ver programa 3.13.

Programa 3.13. Incidencias del proceso de ofuscación.

```
Jshrink 2.33 Copyright 1997-2004 Eastridge Technology www.e-t.com
Rename  I       <=  Hola
Rename  I.I(String)     <=  Hola.getHola
Rename  I.I.getClass    <=  I.I.NGAJ
Rename  I.I.getResourceAsStream <=  I.I.append
Rename  I.I.intern      <=  I.I.close
Rename  Test1.append    <=  Test1.hello
Rename  Test1.append()  <=  Test1.pruebaOfus
Rename  Test1.out       <=  Test1.aux
Rename  Test1.out()     <=  Test1.getVector
Rename  Test1.println   <=  Test1.vector
Class Hola 208 / 284 bytes = 26.8% reduction
Class I.I 1,014 / 1,021 bytes = 0.7% reduction
Class Test1 914 / 1,102 bytes = 17.1% reduction
Output 3 out of 3 class files in 0.08 seconds
Output class file size reduction 2,136 / 2,407 bytes = 11.3%
```

Obtenemos como salida un archivo "*.jar*", aunque podíamos haber elegido otro formato de salida: "*Test12res.jar*" (2.018 *bytes*).

3) Comprobamos que el código ofuscado sigue realizando la misma función. Para ello primeramente extraemos los *bytecodes* del contenedor de salida y posteriormente los ejecutamos:

```
D:\Temp\prueba\jar xf Test12res.jar
```

Obtenemos los siguientes archivos: "*Test1.class*" (914 *bytes*), "*I.class*" (208 *bytes*), más un directorio que contiene una nueva clase también denominada "*I.class*" (1.014 *bytes*). Esta nueva clase aparece como consecuencia del proceso de encriptación. Jshrink utiliza para ello llamadas a métodos que devuelven *strings* encriptados. Al ejecutar nuestro ejemplo comprobamos que sigue realizando la misma función:

```
D:\Temp\prueba\java Test1
HOLA MUNDO
PRUEBA DE OFUSCACION
Ejemplo=1
```

4) A continuación obtenemos los bytecodes de las clases ofuscadas para poder compararlas con sus homólogas sin ofuscar y así poder evaluar nuestra métrica.

```
D:\Temp\prueba\javap –c –p –l Test1
D:\Temp\prueba\javap –c –p –l I
```

Programa 3.14. Bytecode Test1.class.

```
Compiled from Test1
public class Test1 extends java.lang.Object{
public I append;

protected java.lang.String out;

private int[] println;

public static final void main(java.lang.String[]);
  Code:
   0:   new     #1; //class Test1
   3:   dup
   4:   invokespecial   #2; //Method "<init>":()V
   7:   pop
   8:   return


public Test1();
  Code:
   0:   aload_0
   1:   invokespecial   #3; //Method java/lang/Object."<init>":()V
   4:   aload_0
   5:   new     #4; //class I
   8:   dup
   9:   invokespecial   #5; //Method I."<init>":()V
   12:  putfield        #6; //Field append:LI;
   15:  aload_0
   16:  ldc     #7; //String
   18:  putfield        #8; //Field out:Ljava/lang/String;
   21:  aload_0
   22:  bipush  10
   24:  newarray int
   26:  putfield        #9; //Field println:[I
   29:  getstatic      #10; //Field java/lang/System.out:Ljava/io/PrintStream;
   32:  aload_0
   33:  getfield        #6; //Field append:LI;
   36:  iconst_1
   37:  invokestatic    #65; //Method I/I.I:(I)Ljava/lang/String;
   40:  invokevirtual #11; //Method I.I:(Ljava/lang/String;)Ljava/lang/String;
   43:  invokevirtual #12; //Method java/io/PrintStream.println:(Ljava/lang/St
ring;)V
   46:  getstatic    #10; //Field java/lang/System.out:Ljava/io/PrintStream;
   49:  aload_0
```

```
   50:  invokevirtual   #13; //Method append:()Ljava/lang/String;
   53:  invokevirtual #12; //Method java/io/PrintStream.println:(Ljava/lang/St
ring;)V
   56:  getstatic     #10; //Field java/lang/System.out:Ljava/io/PrintStream;
   59:  new     #14; //class StringBuffer
   62:  dup
   63:  invokespecial   #15; //Method java/lang/StringBuffer."<init>":()V
   66:  bipush  12
   68:  invokestatic    #65; //Method I/I.I:(I)Ljava/lang/String;
   71:  invokevirtual #16; //Method java/lang/StringBuffer.append:(Ljava/lang/
String;)Ljava/lang/StringBuffer;
   74:  aload_0
   75:  invokespecial   #17; //Method out:()I
   78:  invokevirtual #18; //Method java/lang/StringBuffer.append:(I)Ljava/lan
g/StringBuffer;
   81:  invokevirtual #19; //Method java/lang/StringBuffer.toString:()Ljava/la
ng/String;
   84:  invokevirtual #12; //Method java/io/PrintStream.println:(Ljava/lang/St
ring;)V
   87:  return

protected final java.lang.String append();
  Code:
   0:   aload_0
   1:   bipush  21
   3:   invokestatic    #65; //Method I/I.I:(I)Ljava/lang/String;
   6:   putfield        #8; //Field out:Ljava/lang/String;
   9:   aload_0
   10:  getfield        #8; //Field out:Ljava/lang/String;
   13:  areturn


private int out();
  Code:
   0:   aload_0
   1:   getfield        #9; //Field println:[I
   4:   iconst_0
   5:   iconst_1
   6:   iastore
   7:   aload_0
   8:   getfield        #9; //Field println:[I
   11:  iconst_0
   12:  iaload
   13:  ireturn
}
```

Programa 3.15. Bytecode I.class.

```
Compiled from I
class I extends java.lang.Object{
public I();
  Code:
   0:   aload_0
   1:   invokespecial   #1; //Method java/lang/Object."<init>":()V
   4:   iconst_1
   5:   istore_1
   6:   return


public final java.lang.String I(java.lang.String);
  Code:
   0:   aload_1
   1:   areturn
}
```

Programa 3.16. Bytecode clase generada para encriptación, I.class.

```
Compiled from I.I
public class I.I extends java.lang.Object{
static byte[] getClass;

static java.lang.String[] getResourceAsStream;

static int[] intern;

public I.I();
  Code:
   0:   aload_0
   1:   invokespecial   #1; //Method java/lang/Object."<init>":()V
   4:   return


public static final synchronized java.lang.String I(int);
  Code:

   0:   iload_0
   1:   sipush  255
   4:   iand
   5:   istore_1
   6:   getstatic       #2; //Field intern:[I
   9:   iload_1
   10:  iaload
   11:  iload_0
   12:  if_icmpeq       62
   15:  getstatic       #2; //Field intern:[I
   18:  iload_1
   19:  iload_0
   20:  iastore
   21:  iload_0
   22:  ifge    30
   25:  iload_0
   26:  ldc     #3; //int 65535
   28:  iand
   29:  istore_0
   30:  new     #4; //class String
   33:  dup
   34:  getstatic       #5; //Field getClass:[B
   37:  iload_0
   38:  getstatic       #5; //Field getClass:[B
   41:  iload_0
   42:  iconst_1
   43:  isub
   44:  baload
   45:  sipush  255
   48:  iand
   49:  invokespecial   #6; //Method java/lang/String."<init>":([BII)V
   52:  invokevirtual#7; //Method java/lang/String.intern:()Ljava/lang/String;
   55:  astore_2
   56:  getstatic       #8; //Field getResourceAsStream:[Ljava/lang/String;
   59:  iload_1
   60:  aload_2
   61:  aastore
   62:  getstatic       #8; //Field getResourceAsStream:[Ljava/lang/String;
   65:  iload_1
   66:  aaload
   67:  areturn


static {};
  Code:
   0:   sipush  256
   3:   anewarray       #4; //class String
   6:   putstatic       #8; //Field getResourceAsStream:[Ljava/lang/String;
   9:   sipush  256
   12:  newarray int
```

```
   14:  putstatic       #2; //Field intern:[I
   17:  new     #9; //class I
   20:  dup
   21:  invokespecial   #10; //Method "<init>":()V
   24:  invokevirtual#11; //Method java/lang/Object.getClass:()Ljava/lang/Cla
ss;
   27:  new     #12; //class StringBuffer
   30:  dup
   31:  invokespecial   #13; //Method java/lang/StringBuffer."<init>":()V
   34:  bipush  73
   36:  invokevirtual #14; //Method java/lang/StringBuffer.append:(C)Ljava/lan
g/StringBuffer;
   39:  bipush  46
   41:  invokevirtual #14; //Method java/lang/StringBuffer.append:(C)Ljava/lan
g/StringBuffer;
   44:  bipush  103
   46:  invokevirtual #14; //Method java/lang/StringBuffer.append:(C)Ljava/lan
g/StringBuffer;
   49:  bipush  105
   51:  invokevirtual #14; //Method java/lang/StringBuffer.append:(C)Ljava/lan
g/StringBuffer;
   54:  bipush  102
   56:  invokevirtual #14; //Method java/lang/StringBuffer.append:(C)Ljava/lan
g/StringBuffer;
   59:  invokevirtual #15; //Method java/lang/StringBuffer.toString:()Ljava/la
ng/String;
   62:  invokevirtual #16; //Method java/lang/Class.getResourceAsStream:(Ljava
/lang/String;)Ljava/io/InputStream;
   65:  astore_0
   66:  aload_0
   67:  ifnull  169
   70:  aload_0
   71:  invokevirtual   #17; //Method java/io/InputStream.read:()I
   74:  bipush  16
   76:  ishl
   77:  aload_0
   78:  invokevirtual   #17; //Method java/io/InputStream.read:()I
   81:  bipush  8
   83:  ishl
   84:  ior
   85:  aload_0
   86:  invokevirtual   #17; //Method java/io/InputStream.read:()I
   89:  ior
   90:  istore_1
   91:  iload_1
   92:  newarray byte
   94:  putstatic       #5; //Field getClass:[B
   97:  iconst_0
   98:  istore_2
   99:  iload_1
   100: i2b
   101: istore_3
   102: getstatic       #5; //Field getClass:[B
   105: astore  4
   107: goto    161
   110: aload_0
   111: aload   4
   113: iload_2
   114: iload_1
   115: invokevirtual   #18; //Method java/io/InputStream.read:([BII)I
   118: istore  5
   120: iload   5
   122: iconst_m1
   123: if_icmpne       129
   126: goto    165
   129: iload_1
   130: iload   5
   132: isub
```

```
   133: istore_1
   134: iload   5
   136: iload_2
   137: iadd
   138: istore  5
   140: goto    155
   143: aload   4
   145: iload_2
   146: dup2
   147: baload
   148: iload_3
   149: ixor
   150: i2b
   151: bastore
   152: iinc    2, 1
   155: iload_2
   156: iload   5
   158: if_icmplt       143
   161: iload_1
   162: ifne    110
   165: aload_0
   166: invokevirtual   #19; //Method java/io/InputStream.close:()V
   169: goto    173
   172: astore_0
   173: return
  Exception table:
   from   to  target type
    17   169   172   Class java/lang/Exception

}
```

Comprando los bytecodes ofuscados con los originales podemos evaluar la métrica diseñada. Ver la tabla 3.8.

3.2.4.3. Ofuscación de Ejemplo 2

Jshrink no realiza ofuscación de Control de Flujo, no está especificado ni como opción en línea de comandos, ni tampoco mediante el uso de la interfaz gráfica que proporciona. Por tanto puntuamos con un 0 la métrica individual correspondiente.

MÉTRICA		EJEMPLO 1 (sin)		EJEMPLO 1(ofuscado)		PUNTUACIÓN
		Test1class	*Hola.class*	*Test1.class*	*a.class*	
Public	**Clase**					1
	Métodos	main	GetHola, Hola	main	I,I	
	Campos	hello		append		
Protected	**Métodos**	pruebaOfus		append		1
	Campos	aux		out		
Private	**Métodos**	getVector		out		1
	Campos	vector		println		
Borrado Numero Linea		Numero de Linea		Borrado		1
Encriptado de String		PRUEBA DE OFUSCACION				1
Desestructuracion de Datos		vector[]		println[]		0
Puntuación para Ofuscación de Layout y Datos						5

Tabla 3.8. Evaluación de ofuscación de Estructura y Datos para el ejemplo 1.

3.2.4.4. Conclusión

- Jshrink puede realizar ofuscación de Estructura, renombrando identificadores de clases y miembros de clases *Public*, *Protected* y *Private*. Además borra toda información redundante, como comentarios, miembros si uso, etc.

- Es capaz de realizar ofuscación de Datos como puede verse en la tabla 3.8, proporciona opciones de encriptado. Sin embargo no realiza desestructuración de datos.

- Después de ser ofuscado con las opciones más severas, nuestro código seguía realizando la misma función.

- Debemos destacar que finaliza el código, de manera que borra toda información de *throws*, marca todos los métodos no dominantes como *final* de manera que puedan ser *inlined* durante la ejecución y puede que no se preserven los miembros generados en la compilación usados por clases internas e invocación de métodos remotos (*RMI*).

- No proporciona mecanismos para ofuscación de Control de Flujo.

- Como hemos comentado anteriormente el proceso de encriptado supone una cierta sobrecarga que se ve reflejada en un aumento del tamaño de nuestra aplicación: "*Test1.jar*" (1.425 *bytes*) produce como resultado "*Test12res.jar*" (2.018 *bytes*). A costa de obtener una mayor seguridad para la prevención de la ingeniería inversa.

3.2.5. CAFEBABE

3.2.5.1. Introducción

CafeBabe más que un ofuscador es un completo desensamblador de código Java que además permite ofuscar y compactar el código. Muestra visualmente toda la información contenida dentro del Java bytecode: atributos, métodos, argumentos, excepciones, cuerpo de los métodos, información adicional incluida dentro de las clases, etc.

En cuanto a la tarea de ofuscación, proporciona la posibilidad de proteger el código de aplicaciones contra ataques de ingeniería inversa. CafeBabe analiza las clases cargadas, encuentra dependencias entre ellas y resuelve la tarea de ofuscación. Podemos seleccionar los objetos a ofuscar: paquetes, clases o miembros de clases. Además permite mover clases de diferentes paquetes hacia un paquete anónimo. Este método consigue que se reduzca en gran medida el tamaño de nuestro proyecto.

Algunas de las características de esta aplicación son:

- Representa el *bytecode* tal como es dentro del formato de los archivos de las clases compiladas.
- Puede leer las clases directamente desde un archivo "*.jar*" o "*.zip*", e interpretar toda la estructura de los mismos.
- Permite buscar una cadena dentro del *bytecode*.
- Permite mostrar la información mostrada (en formato UTF8-string).
- Permite eliminar información opcional contenida en el código.
- Permite la ofuscación del *bytecode*.

3.2.5.2. Ofuscación de Ejemplo 1

Una vez estudiadas las breves notas sobre la utilización de nuestra aplicación en la página web asociada, hemos seguido los siguientes pasos para la evaluación de la misma.

1) Abrir CafeBabe mediante la instrucción desde línea de comandos:

```
D:\Temp\CafeBabe\CafeBabe>java -jar CafeBabe.jar
```

Obtenemos una interfaz gráfica (*GUI*) de fácil manejo y muy intuitiva, para la cual solo tenemos que cargar el archivo a tratar. Este archivo debe ser un "*.class*". No debemos olvidar que CafeBabe es ante todo un desensamblador. Además mediante la opción "*Class-Hound Service*" podemos cargar el contenedor "*Test1.jar*" que contiene los bytecodes de nuestro ejemplo 1.

2) El siguiente paso ha sido la optimización de nuestro código fuente. Para ello, en la pestaña de tareas hemos seleccionado "*strip*". Aparece una especie de navegador que nos permite explorar el archivo o conjunto de archivos a optimizar. Seleccionamos entrada, salida y opciones de optimización: borrar información redundante en el cuerpo de los métodos, atributos de clases desconocidos y atributos de código fuente. El resultado es un archivo de salida "*Test1res.jar*" aun sin ofuscar pero para el cual hemos reducido el tamaño de los bytecodes: "*Hola.class*" (224 *bytes*) y "*Test1.class*" (986 *bytes*). Para esta comprobación previamente hemos tenido que extraer el contenido del archivo de salida mediante la instrucción:

```
D:\Temp\prueba\Test1res.jar>jar xf Test1res.jar
```

3) A continuación hemos procedido a ofuscar nuestro código optimizado. Para esto al igual que antes nos hemos ido a la pestaña de tareas y hemos seleccionado la opción "Obfuscate". Al igual que para la tarea de optimización, obtenemos un navegador que nos permite seleccionar la entrada, la salida y las opciones para el proceso de ofuscación. Como entrada hemos tomado el contenedor optimizado en el paso anterior "*Test1res.jar*". Las opciones de configuración para el proceso de ofuscación que hemos decidido aplicar son las siguientes: modo anónimo de paquetes, ofuscación de paquetes, clases y miembros de clases. Por otro lado pueden especificarse clases y miembros de clases que no queremos que sean ofuscados. No permite más opciones. Es necesario escribir un punto de entrada desde el cual CafeBabe aplicará ofuscación recursivamente, este punto de entrada debe ser necesariamente la clase principal (Test1, en nuestro caso). A la salida obtenemos otro contenedor "*Test12res.jar*", esta vez con el código ofuscado.

4) Comprobamos que el código ofuscado sigue realizando la misma función. Para ello primeramente extraemos los bytecodes del contenedor de salida y posteriormente los ejecutamos:

```
D:\Temp\prueba\jar xf Test12res.jar
```

Obtenemos los siguientes archivos: "*Test1.class*" (958 *bytes*), "*A.class*" (217 *bytes*). Para comprobar que siguen realizando la misma función ejecutamos:

```
D:\Temp\prueba\java Test1
HOLA MUNDO
PRUEBA DE OFUSCACION
Ejemplo=1
```

5) Con objeto de comparar los *bytecodes* obtenidos (ver programa 3.17 y 3.18) con los del código fuente original y poder evaluar así el proceso de ofuscación llevado a cabo por la aplicación CafeBabe ejecutamos:

```
D:\Temp\prueba\javap –c –p –l Test1
D:\Temp\prueba\javap –c –p –l A
```

Programa 3.17. Bytecode Test1.class.

```
Compiled from null
public class Test1 extends java.lang.Object{
public A ajb;

protected java.lang.String ajc;

private int[] ajd;

public static void main(java.lang.String[]);
  Code:
   0:   new     #1; //class Test1
   3:   dup
   4:   invokespecial   #2; //Method "<init>":()V
   7:   pop
   8:   return

public Test1();
  Code:
   0:   aload_0
   1:   invokespecial   #3; //Method java/lang/Object."<init>":()V
   4:   aload_0
   5:   new     #4; //class A
   8:   dup
   9:   invokespecial   #5; //Method A."<init>":()V
   12:  putfield        #6; //Field ajb:LA;
   15:  aload_0
   16:  ldc     #7; //String
   18:  putfield        #8; //Field ajc:Ljava/lang/String;
   21:  aload_0
   22:  bipush  10
   24:  newarray int
   26:  putfield        #9; //Field ajd:[I
   29:  getstatic     #10; //Field java/lang/System.out:Ljava/io/PrintStream;
   32:  aload_0
   33:  getfield        #6; //Field ajb:LA;
   36:  ldc     #11; //String HOLA MUNDO
   38:  invokevirtual#12;//Method A.aja:(Ljava/lang/String;)Ljava/lang/String;
   41:  invokevirtual #13; //Method java/io/PrintStream.println:(Ljava/lang/St
ring;)V
   44:  getstatic   #10; //Field java/lang/System.out:Ljava/io/PrintStream;
   47:  aload_0
   48:  invokevirtual   #14; //Method aje:()Ljava/lang/String;
   51:  invokevirtual #13; //Method java/io/PrintStream.println:(Ljava/lang/St
ring;)V
   54:  getstatic  #10; //Field java/lang/System.out:Ljava/io/PrintStream;
   57:  new     #15; //class StringBuffer
   60:  dup
   61:  invokespecial   #16; //Method java/lang/StringBuffer."<init>":()V
   64:  ldc     #17; //String Ejemplo=
   66:  invokevirtual#18; //Method java/lang/StringBuffer.append:(Ljava/lang/
```

```
String;)Ljava/lang/StringBuffer;
   69:  aload_0
   70:  invokespecial   #19; //Method ajf:()I
   73:  invokevirtual #20; //Method java/lang/StringBuffer.append:(I)Ljava/lan
g/StringBuffer;
   76:  invokevirtual#21; //Method java/lang/StringBuffer.toString:()Ljava/la
ng/String;
   79:  invokevirtual   #13; //Method
java/io/PrintStream.println:(Ljava/lang/St
ring;)V
   82:  return

protected java.lang.String aje();
  Code:
   0:   aload_0
   1:   ldc     #22; //String PRUEBA DE OFUSCACION
   3:   putfield        #8; //Field ajc:Ljava/lang/String;
   6:   aload_0
   7:   getfield        #8; //Field ajc:Ljava/lang/String;
   10:  areturn

private int ajf();
  Code:
   0:   aload_0
   1:   getfield        #9; //Field ajd:[I
   4:   iconst_0
   5:   iconst_1
   6:   iastore
   7:   aload_0
   8:   getfield        #9; //Field ajd:[I
   11:  iconst_0
   12:  iaload
   13:  ireturn
}
```

Programa 3.18. Bytecode A.class.

```
Compiled from null
class A extends java.lang.Object{
public A();
  Code:
   0:   aload_0
   1:   invokespecial   #1; //Method java/lang/Object."<init>":()V
   4:   iconst_1
   5:   istore_1
   6:   return


public java.lang.String aja(java.lang.String);
  Code:
   0:   aload_1
   1:   areturn


}
```

En base a estos *bytecodes* podemos evaluar la métrica diseñada para la comparación de las diferentes aplicaciones de ofuscación. En base a esta comparación hemos construido la tabla 3.9.

MÉTRICA		EJEMPLO 1 (sin)		EJEMPLO 1(ofuscado)		PUNTUACIÓN
		Test1class	*Hola.class*	*Test1.class*	*A.class*	
Public	**Clase**					1
	Métodos	main	GetHola, Hola	main	A, aja	
	Campos	hello		ajb		
Protected	**Métodos**	pruebaOfus		aje		1
	Campos	aux		ajc		
Private	**Métodos**	getVector		ajf		1
	Campos	vector		ajd		
Borrado Numero Linea		Numero de Linea		Borrado		1
Encriptado de String		PRUEBA DE OFUSCACION		PRUEBA DE OFUSCACION		0
Desestructuracion de Datos		vector[]		ajd[]		0
Puntuación para Ofuscación de Layout y Datos						4

Tabla 3.9. Evaluación de ofuscación de Layout y Datos para el ejemplo 1.

3.2.5.3. Ofuscación de Ejemplo 2

CafeBabe no realiza ofuscación de Control de Flujo. Por tanto puntuamos con un 0 la métrica individual correspondiente.

3.2.5.4. Conclusión

- CafeBabe puede realizar ofuscación de Estructura, renombrando identificadores de clases y miembros de clases *Public*, *Protected* y *Private*. Además borra toda información redundante, como comentarios, miembros si uso, etc.

- No es capaz de realizar ofuscación de Datos como puede verse en la tabla 3.9, es decir, no proporciona opciones de encriptado o desestructuración de datos.

- Después de ser ofuscado con las opciones más severas, nuestro código seguía realizando la misma función.

- No proporciona mecanismos para ofuscación de Control de Flujo.

- Aunque obtiene una puntuación igual a la de otras aplicaciones analizadas con anterioridad, podemos destacar que CafeBabe proporciona un número mucho menor de opciones. Esto puede ser debido a que es un proyecto abandonado, pues desde 1999 no se han realizado actualizaciones de la aplicación.

- No podemos trabajar desde línea de comandos.

- No se proporciona un manual de usuario que explique en detalle como trabajar con la aplicación. Sin embargo, es fácil de manejar y muy intuitivo si se ha trabajado con herramientas de este tipo con anterioridad.

3.2.6. ZELIX KLASSMASTER

3.2.6.1. Introducción

Zelix KlassMaster puede ofuscar el *bytecode* de manera que código obtenido mediante un descompilador sea ininteligible. Como ventaja adicional, al ofuscar nuestras aplicaciones obtendremos una reducción en el tamaño de las mismas, de manera que podremos enviarlas por la red más rápidamente.

Esta aplicación presenta las siguientes características para el proceso de ofuscación:

- Proporciona ofuscación de Estructura, realizando renombrado de paquetes, clases, campos y métodos.

- Proporciona ofuscación de Control, que cambia la mayoría de las sentencias condicionales y bucles, de manera que estas construcciones no tengan un equivalente directo en Java.

- Puede encriptar *strings* para hacer el código descompilado menos leíble.

- Puede comprimir la estructura de paquetes con el objetivo de reducir el tamaño del *bytecode* de nuestra aplicación en conjunto.

- Proporciona la función de exclusión de nombres, que nos da un buen control sobre los nombres que son ofuscados.

- Puede producir un archivo *logfile* muy detallado en el que se graban todos los cambios de nombres realizados y además proporciona una herramienta *Stack Trace Translation Tool* para ayudarnos a interpretar el proceso llevado a cabo.

- Borra el número de línea para no dar a descompiladores ninguna información extra.

- Podemos cargar un archivo *logfile* existente de manera que obtenemos un renombramiento y una ofuscación de control más consistente. Esto es importante cuando utilizamos RMI .

- Maneja RMI y *JavaBeans* por defecto.

- Además permite la integración en J2ME Wireless Toolkit.

Debemos destacar que la opción de excluir métodos específicos de la ofuscación de control solamente está disponible desde el interfaz ZKM Script.

La herramienta de ofuscación trabaja sobre todas las clases abiertas. No puede ofuscar clases que no hayan sido cargadas previamente. Por tanto, es necesario que carguemos nuestra aplicación completa, de manera que la dependencia entre clases pueda ser actualizada apropiadamente.

3.2.6.2. Ofuscación de Ejemplo 1

Los pasos que hemos seguido para la evaluación del proceso de ofuscación llevado a cabo por la aplicación Zelix KlassMaster sobre nuestro ejemplo 1 han sido los siguientes:

1) Abrir la aplicación Zelix. Para ello, desde una consola ejecutamos:

```
D:\Temp\ZKM\java –jar ZKM.jar
```

2) Cargar nuestro ejemplo. Previamente hemos almacenado nuestros bytecodes en un contenedor "*Test1.jar*". Como hemos observado al principio, podemos abrir clases desde un directorio o desde un contenedor como es nuestro caso. Para ello solo tenemos que pulsar *File* y *Open*. Obtendremos un explorador que nos permitirá cargar el archivo deseado.

3) Utilizar la herramienta de ofuscación. Para ello, seleccionamos *Tools* y *Obfuscate*. A continuación seleccionamos las opciones de ofuscación que deseemos. En nuestro caso hemos seleccionado las más agresivas. Ver el apartado de utilización de la aplicación.

4) Ejecutar el proceso de ofuscación. Obtenemos como salida un contenedor "*Test1res.jar*" que hemos seleccionado previamente en *File>Save All*. Debemos destacar que se ha generado un archivo *log* donde se especifican todas las incidencias del proceso de ofuscación. Ver programa 3.19.

Programa 3.19. Archivo ChangeLog donde se reflejan las incidencias de la ofuscación.

```
// ["D:\Temp\prueba\log.txt" version=4.3.6d 2005.03.29 16:18:21]
// DO NOT EDIT THIS FILE. You need it to interpret exception stack traces.


Class: Hola  =>     a
       Source: "Test1.java"
       FieldsOf: Hola
              public static int z =>     a
       MethodsOf: Hola
              public java.lang.String getHola(java.lang.String)    =>     a

Class: public Test1 NameNotChanged
       Source: "Test1.java"
       FieldsOf: Test1
              protected java.lang.String aux   =>     b
              public Hola hello   =>     a
              private int[] vector      =>     c
              public static boolean z   =>     d
              private static java.lang.String[] z     NameNotChanged
       MethodsOf: Test1
              private int getVector()   =>     b
              public static void main(java.lang.String[])   NameNotChanged
              protected java.lang.String pruebaOfus() =>     a


TraceBackClass: Test1     Data: 1637
ForwardClass: Hola  Data: 2269
MemberClass: Hola   Data: 98
MemberClass: Test1  Data: 466
```

Los *bytecodes* ofuscados los obtenemos extrayéndolos del contenedor de salida mediante la instrucción:

D:\Temp\prueba\jar xf Test1res.jar

Obtenemos: "*Test1.class*" (1.339 *bytes*) y "*a.class*" (242 *bytes*). Comprobamos que hemos obtenido bytecodes de mayor tamaño que los originales. Esto se debe al proceso de encriptado y ofuscación de control como hemos explicado anteriormente. Comprobamos que siguen realizando la misma función, para ello:

D:\Temp\prueba\java Test1
HOLA MUNDO
PRUEBA DE OFUSCACION
Ejemplo=1

5) A continuación comparamos los *bytecodes* ofuscados obtenidos con los originales para poder así evaluar el proceso de ofuscación llevado a cabo por la aplicación Zelix KlasMaster. Ver programas 3.20 y 3.21.

D:\Temp\prueba\javap –c –p –l Test1
D:\Temp\prueba\javap –c –p –l A

Programa 3.20. Bytecode Test1.class.

```
Compiled from "Test1.java"
public class Test1 extends java.lang.Object{
```

```
public a a;

protected java.lang.String b;

private int[] c;

public static boolean d;

private static java.lang.String[] z;

public static void main(java.lang.String[]);
  Code:
   0:   new     #1; //class Test1
   3:   dup
   4:   invokespecial   #2; //Method "<init>":()V
   7:   pop
   8:   return


public Test1();
  Code:
   0:   getstatic       #76; //Field a.a:I
   3:   istore_1
   4:   aload_0
   5:   invokespecial   #3; //Method java/lang/Object."<init>":()V
   8:   aload_0
   9:   new     #4; //class a
   12:  dup
   13:  invokespecial   #5; //Method a."<init>":()V
   16:  putfield        #6; //Field a:La;
   19:  aload_0
   20:  ldc     #7; //String
   22:  putfield        #8; //Field b:Ljava/lang/String;
   25:  aload_0
   26:  bipush  10
   28:  newarray int
   30:  putfield        #9; //Field c:[I
   33:  getstatic     #10; //Field java/lang/System.out:Ljava/io/PrintStream;
   36:  aload_0
   37:  getfield        #6; //Field a:La;
   40:  getstatic       #98; //Field z:[Ljava/lang/String;
   43:  iconst_1
   44:  aaload
   45:  invokevirtual #12; //Method a.a:(Ljava/lang/String;)Ljava/lang/String;
   48:  invokevirtual #13; //Method java/io/PrintStream.println:(Ljava/lang/St
ring;)V
   51:  getstatic    #10; //Field java/lang/System.out:Ljava/io/PrintStream;
   54:  aload_0
   55:  invokevirtual   #14; //Method a:()Ljava/lang/String;
   58:  invokevirtual #13; //Method java/io/PrintStream.println:(Ljava/lang/St
ring;)V
   61:  getstatic     #10; //Field java/lang/System.out:Ljava/io/PrintStream;
   64:  new     #15; //class StringBuffer
   67:  dup
   68:  invokespecial   #16; //Method java/lang/StringBuffer."<init>":()V
   71:  getstatic       #98; //Field z:[Ljava/lang/String;
   74:  iconst_0
   75:  aaload
   76:  invokevirtual #18; //Method java/lang/StringBuffer.append:(Ljava/lang/
String;)Ljava/lang/StringBuffer;
   79:  aload_0
   80:  invokespecial   #19; //Method b:()I
   83:  invokevirtual #20; //Method java/lang/StringBuffer.append:(I)Ljava/lan
g/StringBuffer;
   86:  invokevirtual #21; //Method java/lang/StringBuffer.toString:()Ljava/la
ng/String;
   89:  invokevirtual #13; //Method java/io/PrintStream.println:(Ljava/lang/St
ring;)V
```

```
   92:  iload_1
   93:  ifeq    110
   96:  getstatic       #78; //Field d:Z
   99:  ifeq    106
   102: iconst_0
   103: goto    107
   106: iconst_1
   107: putstatic       #78; //Field d:Z
   110: return


protected java.lang.String a();
  Code:
   0:   aload_0
   1:   getstatic       #98; //Field z:[Ljava/lang/String;
   4:   iconst_2
   5:   aaload
   6:   putfield        #8; //Field b:Ljava/lang/String;
   9:   aload_0
   10:  getfield        #8; //Field b:Ljava/lang/String;
   13:  areturn


private int b();
  Code:
   0:   aload_0
   1:   getfield        #9; //Field c:[I
   4:   iconst_0
   5:   iconst_1
   6:   iastore
   7:   aload_0
   8:   getfield        #9; //Field c:[I
   11:  iconst_0
   12:  iaload
   13:  ireturn


static {};
  Code:
   0:   iconst_3
   1:   anewarray       #82; //class String
   4:   dup
   5:   iconst_0
   6:   ldc     #17; //String M→)↕Md▼q
   8:   jsr     34
   11:  aastore
   12:  dup
   13:  iconst_1
   14:  ldc     #11; //String @? >↔E%☻;r
   16:  jsr     34
   19:  aastore
   20:  dup
   21:  iconst_2
   22:  ldc     #22; //String X\"↓:⌂I:↔G6↓,~I3♣0s
   24:  jsr     34
   27:  aastore
   28:  putstatic       #98; //Field z:[Ljava/lang/String;
   31:  goto    133
   34:  astore_0
   35:  invokevirtual   #88; //Method java/lang/String.toCharArray:()[C
   38:  dup
   39:  arraylength
   40:  swap
   41:  iconst_0
   42:  istore_1
   43:  goto    112
   46:  dup
   47:  iload_1
```

```
   48:  dup2
   49:  caload
   50:  iload_1
   51:  iconst_5
   52:  irem
   53:  tableswitch{ //0 to 3
                0: 84;
                1: 89;
                2: 94;
                3: 99;
                default: 104 }
   84:  bipush  8
   86:  goto    106
   89:  bipush  112
   91:  goto    106
   94:  bipush  76
   96:  goto    106
   99:  bipush  127
   101: goto    106
   104: bipush  61
   106: ixor
   107: i2c
   108: castore
   109: iinc    1, 1
   112: swap
   113: dup_x1
   114: iload_1
   115: if_icmpgt       46
   118: new     #82; //class String
   121: dup_x1
   122: swap
   123: invokespecial   #92; //Method java/lang/String."<init>":([C)V
   126: invokevirtual#96;//Method java/lang/String.intern:()Ljava/lang/String;
   129: swap
   130: pop
   131: ret     0
   133: return
}
```

Programa 3.21. Bytecode a.class.

```
Compiled from "a.java"
class a extends java.lang.Object{
public static int a;

public a();
  Code:
   0:   aload_0
   1:   invokespecial   #1; //Method java/lang/Object."<init>":()V
   4:   iconst_1
   5:   istore_1
   6:   return


public java.lang.String a(java.lang.String);
  Code:
   0:   aload_1
   1:   areturn
}
```

Una vez obtenidos los bytecodes ofuscados podemos evaluar el proceso de ofuscación llevado a cabo por esta aplicación mediante la métrica diseñada.

MÉTRICA		EJEMPLO 1 (sin)		EJEMPLO 1(ofuscado)		PUNTUACIÓN
		Test1class	*Hola.class*	*Test1.class*	*a.class*	
Public	**Clase**					1
	Métodos	main	GetHola, Hola	main	a,a	
	Campos	hello		a		
Protected	**Métodos**	pruebaOfus		a		1
	Campos	aux		b		
Private	**Métodos**	getVector		b		1
	Campos	vector		c		
Borrado Numero Linea		Numero de Linea		Borrado		1
Encriptado de String		PRUEBA DE OFUSCACION		Encriptado		1
Desestructuracion de Datos		vector[]		c[]		0
Puntuación para Ofuscación de Layout y Datos						5

Tabla 3.10. Evaluación de ofuscación de Layout y Datos para el ejemplo 1.

3.2.6.3. Ofuscación de Ejemplo 2

Zelix KlassMaster si proporciona ofuscación de Control a diferencia de la mayoría de las aplicaciones estudiadas hasta ahora. Por tanto lo convierte en uno de los ofuscadores más completos estudiados. Para evaluar este proceso de ofuscación de control habíamos diseñado el ejemplo 2 (ver apartado 2.6.2. Diseño de Ejemplos). Los pasos que hemos realizado para esta evaluación son los siguientes:

1) Hemos abierto la aplicación Zelix KlassMaster como se ha explicado anteriormente:

D:\Temp\ZKM\java –jar ZKM.jar

2) Hemos cargado el bytecode de nuestro ejemplo 2 "*Test2.class*". En este caso no ha sido necesario crear un contenedor porque solo tenemos una clase.

3) A continuación hemos utilizado la herramienta de ofuscación. Para ello, seleccionamos Tools y *Obfuscate*. Seleccionando las opciones de ofuscación que deseemos. En nuestro caso hemos seleccionado las más agresivas. Ver el apartado de utilización de la aplicación.

4) Ejecutamos el proceso de ofuscación. Obtenemos como salida un archivo "*a.class*" que hemos seleccionado previamente en *File>Save All*. Debemos destacar que se ha generado un archivo *log* donde se especifican todas las incidencias del proceso de ofuscación. Ver programa 3.22.

Programa 3.22. Archivo ChangeLog donde se reflejan incidencias de ofuscaión.

```
// ["D:\Temp\prueba\log.txt" version=4.3.6d 2005.03.29 17:15:13]
// DO NOT EDIT THIS FILE. You need it to interpret exception stack traces.


Class: public Test2 NameNotChanged
       Source: "Test2.java"
       FieldsOf: Test2
              public static boolean A    NameNotChanged
              private static java.lang.String[] B     NameNotChanged
              public static boolean z    NameNotChanged
       MethodsOf: Test2
              public static void main(java.lang.String[])   NameNotChanged


TraceBackClass: Test2      Data: 1638
ForwardClass: Test2 Data: 1567
MemberClass: Test2  Data: 467
```

El archivo de salida "*a.class*" (1.118 *bytes*) seguía realizando la misma función, para comprobarlo ejecutamos la instrucción:

D:\Temp\prueba>java a
Numero de bucle > 1
Numero de bucle=1

5) A continuación comparamos el bytecode ofuscado obtenido con el original para poder así evaluar el proceso de ofuscación llevado a cabo por la aplicación Zelix KlasMaster. Ver programa 3.23.

D:\Temp\prueba\javap –c –p –l a

Programa 3.23. Bytecode a.class.

```
Compiled from "a.java"
public class a extends java.lang.Object{
public static boolean z;

public static boolean A;

private static java.lang.String[] B;

public a();

Code:
   0:   aload_0
   1:   invokespecial   #1; //Method java/lang/Object."<init>":()V
   4:   return


public static void main(java.lang.String[]);
  Code:
   0:   getstatic       #56; //Field A:Z
   3:   istore_3
   4:   iconst_0
   5:   istore_1
   6:   iconst_0
   7:   istore_2
   8:   iload_2
   9:   iconst_2
   10:  if_icmpge       41
```

```
   13:  iinc    1, 1
   16:  iinc    2, 1
   19:  iload_3
   20:  ifne    61
   23:  iload_3
   24:  ifeq    8
   27:  getstatic       #58; //Field z:Z
   30:  ifeq    37
   33:  iconst_0
   34:  goto    38
   37:  iconst_1
   38:  putstatic       #58; //Field z:Z
   41:  iload_1
   42:  iconst_1
   43:  if_icmple       61
   46:  getstatic       #2; //Field java/lang/System.out:Ljava/io/PrintStream;
   49:  getstatic       #78; //Field B:[Ljava/lang/String;
   52:  iconst_2
   53:  aaload
   54:  invokevirtual #4;//Method java/io/PrintStream.print:(Ljava/lang/Strin
g;)V
   57:  iload_3
   58:  ifeq    72
   61:  getstatic       #2; //Field java/lang/System.out:Ljava/io/PrintStream;
   64:  getstatic       #78; //Field B:[Ljava/lang/String;
   67:  iconst_1
   68:  aaload
   69:  invokevirtual #6; //Method java/io/PrintStream.println:(Ljava/lang/Str
ing;)V
   72:  iload_1
   73:  iconst_1
   74:  if_icmple       89
   77:  iload_1
   78:  iconst_1
   79:  isub
   80:  istore_1
   81:  iload_3
   82:  ifne    123
   85:  iload_3
   86:  ifeq    72
   89:  getstatic       #2; //Field java/lang/System.out:Ljava/io/PrintStream;
   92:  invokevirtual   #7; //Method java/io/PrintStream.println:()V
   95:  getstatic       #2; //Field java/lang/System.out:Ljava/io/PrintStream;
   98:  new     #8; //class StringBuffer
   101: dup
   102: invokespecial   #9; //Method java/lang/StringBuffer."<init>":()V
   105: getstatic       #78; //Field B:[Ljava/lang/String;
   108: iconst_0
   109: aaload
   110: invokevirtual #11; //Method java/lang/StringBuffer.append:(Ljava/lang/
String;)Ljava/lang/StringBuffer;
   113: iload_1
   114: invokevirtual #12; //Method java/lang/StringBuffer.append:(I)Ljava/lan
g/StringBuffer;
   117: invokevirtual #13; //Method java/lang/StringBuffer.toString:()Ljava/la
ng/String;
   120: invokevirtual #6; //Method java/io/PrintStream.println:(Ljava/lang/Str
ing;)V
   123: return
static {};
  Code:
   0:   iconst_3
   1:   anewarray       #62; //class String
   4:   dup
   5:   iconst_0
   6:   ldc     #10; //String +L_pqD\nIyq¬L~xSX
   8:   jsr     34
   11:  aastore
```

```
   12:  dup
   13:  iconst_1
   14:  ldc     #5; //String +L_pqD\nIyq⊣L_~xSEU 4
   16:  jsr     34
   19:  aastore
   20:  dup
   21:  iconst_2
   22:  ldc     #3; //String +L_pqD\nIyq⊣L_~xSEW=%
   24:  jsr     34
   27:  aastore
   28:  putstatic       #78; //Field B:[Ljava/lang/String;
   31:  goto    133
   34:  astore_0
   35:  invokevirtual   #68; //Method java/lang/String.toCharArray:()[C
   38:  dup
   39:  arraylength
   40:  swap
   41:  iconst_0
   42:  istore_1
   43:  goto    112
   46:  dup
   47:  iload_1
   48:  dup2
   49:  caload
   50:  iload_1
   51:  iconst_5
   52:  irem
   53:  tableswitch{ //0 to 3
                0: 84;
                1: 89;
                2: 94;
                3: 99;
                default: 104 }
   84:  bipush  101
   86:  goto    106
   89:  bipush  105
   91:  goto    106
   94:  bipush  29
   96:  goto    106
   99:  bipush  20
  101:  goto    106
  104:  bipush  54
  106:  ixor
  107:  i2c
  108:  castore
  109:  iinc    1, 1
  112:  swap
  113:  dup_x1
  114:  iload_1
  115:  if_icmpgt       46
  118:  new     #62; //class String
  121:  dup_x1
  122:  swap
  123:  invokespecial   #72; //Method java/lang/String."<init>":([C)V
  126:  invokevirtual#76;//Method java/lang/String.intern:()Ljava/lang/String;
  129:  swap
  130:  pop
  131:  ret     0
  133:  return
}
```

Comprando el *bytecode* ofuscado obtenido con el original podemos comprobar que es mucho más complicado y que las sentencias condicionales y los bucles han cambiado.

Las desventajas de la ofuscación de control es que la ejecución del bytecode obtenido es más lenta y el tamaño del mismo es ligeramente mayor: "*Test2.class*" (799 *bytes*) en comparación con "*a.class*" (1.118 *bytes*).

3.2.6.4. Conclusión

- Zelix KlassMaster puede realizar ofuscación de Estructura, renombrando identificadores de clases y miembros de clases *Public*, *Protected* y *Private*. Además borra toda información redundante, como comentarios, miembros si uso, etc.

- Es capaz de realizar ofuscación de Datos como puede verse en la tabla 3.15, proporciona opciones de encriptado. Sin embargo no realiza desestructuración de datos.

- Después de ser ofuscado con las opciones más severas, nuestro código seguía realizando la misma función. Los dos ejemplos seguían funcionando.

- Proporciona mecanismos para ofuscación de Control de Flujo. Es la principal ventaja con respecto a otras aplicaciones.

- Se proporciona un manual de usuario muy detallado en la página web asociada a la aplicación.

- Se ejecuta mediante una interfaz gráfica.

- Proporciona una herramienta llamada: *Stack Trace Translate*, que nos permite traducir el código ofuscado. Útil para el caso de trabajar con RMI o serialización.

3.2.7. SMOKESCREEN

3.2.7.1. Introducción

Smokescreen es un ofuscador Java. Más allá de ser capaz de modificar nombres simbólicos, también puede modificar instrucciones de métodos en el *bytecode*, proporciona por tanto ofuscación de Control. Esto hace que las clases ofuscadas resultantes sean mucho más difíciles de descompilar.

Smokescreen trabaja con archivos "*.class*" en directorios pero también con archivos "*.zip*" o "*.jar*". Permite una ofuscación selectiva de los nombres simbólicos dependiendo del nivel de acceso de la clase, método o campo.

La eliminación de métodos y campos sin uso también puede realizarse. También proporciona, como antes se ha mencionado, ofuscación de Control mediante la modificación de instrucciones del *bytecode* en los métodos de las clases.

Las principales características que presenta la aplicación son:

- El programa proporciona una fácil interfaz gráfica (*GUI*), así como la opción de ejecutar desde línea de comandos.
- Permite ofuscación de Layout, renombrando clases, métodos y campos.
- Da opciones de exclusión para el anterior proceso.
- Borra métodos, campos y constantes si uso.
- Proporciona ofuscación de Control.
- Encriptación de Strings.
- Permite distintos niveles de ofuscación.

3.2.7.2. Ofuscación de Ejemplo 1

Con objeto de evaluar el proceso de ofuscación sobre el ejemplo 1 hemos realizado los siguientes pasos:

1) En primer lugar como hemos optado por ejecutar la aplicación desde línea de comandos, es necesario crear un archivo con las directivas oportunas para indicar a la aplicación las opciones deseadas en el proceso de ofuscación. De acuerdo al manual de usuario y buscando una ofuscación lo más agresiva posible generamos un script de configuración. Ver programa 3.24.

Programa 3.33. Directivas de configuración de Smokescreen, myDirectives.txt.

```
* source_directory D:\Temp\prueba\Test1.jar
* destination_directory D:\Temp\prueba\Test1res.jar
* superclass_path C:\Archivos de programa\j2sdk1.4.2_07\jre\lib\rt.jar

* use_class_loader_for_superclasses
* log_changes
* overwrite_classfiles

* classes all_classes
* methods all_methods
* fields all_fields

* bytecode add_fake_exceptions
* bytecode shuffle_stack_operations
* bytecode change_switch_statements
* bytecode encrypt_strings

-method Test1.main
```

2) Una vez tenemos el *script* de directivas podemos ejecutar la aplicación desde línea de comandos tecleando:

D:\Temp\SmokescreenSetup341_Eval\java –jar Smokescreen30_Eval.jar –nogui –directives myDirectives.txt

Notar que es necesario especificar la opción "*-nogui*" para indicar que no se ejecute la interfaz gráfica. Además, como se ha mencionado anteriormente, debemos especificar un archivo de directivas que contiene las opciones adecuadas de ofuscación. Esto lo hacemos mediante la opción "*-directives*". Las opciones elegidas se especifican en el apartado de utilización.

3) Al ejecutar la aplicación se genera un archivo "*ChangeLog.txt*" donde se reflejan las incidencias del proceso de ofuscación. Destacar que en este archivo se detalla el proceso de renombrado llevado a cabo. Ver programa 3.25. Comprobamos que no ha ocurrido ningún error e igualmente comprobamos que nuestro ejemplo sigue realizando la misma función.

Programa 3.25. Archivo ChangeLog.txt.

```
JVM info: Sun Microsystems Inc. 1.4.2_07
OS info: Windows XP x86 5.1

---------------------- Beginning Obfuscation ----------------------
Wednesday, March 30, 2005 10:56:35 AM CEST

Using Directives file 'D:\Temp\SmokescreenSetup341_Eval\mydirectives.txt'
Source and destination are jar files
Source is 'D:\Temp\prueba\Test1.jar'
Destination is 'D:\Temp\prueba\Test1res.jar'

Will use class loader to find super classes

Changes and removals will be recorded
Destination Class, Jar or Zip files may be overwritten
Unused methods will not be removed
Unused fields will not be removed
All class names will be changed
All method names will be changed
All field names will be changed
Stack operations will be shuffled
Fake exceptions will be added
Switch statements will be obfuscated
Strings will be encrypted
Number of selected classes is 2
Total number of classes scanned is 3

Class: 'Test1' changed to 'A'
Method: 'Test1.main' '([Ljava/lang/String;)V' not changed
Method: 'Test1.<init>' '()V' not changed
Method: 'Test1.pruebaOfus' '()Ljava/lang/String;' changed to 'A.A'
Method: 'Test1.getVector' '()I' changed to 'A.B'
Field: 'Test1.hello' 'LHola;' changed to 'A.A'
Field: 'Test1.aux' 'Ljava/lang/String;' changed to 'A.B'
Field: 'Test1.vector' '[I' changed to 'A.C'
Class: 'Hola' changed to 'B'
Method: 'Hola.<init>' '()V' not changed
Method: 'Hola.getHola' '(Ljava/lang/String;)Ljava/lang/String;' changed to
'B.A'
---------------------- Ending Obfuscation ----------------------
```

En el archivo de directivas hemos seleccionado obtener como salida un contenedor "*Test1res.jar*". Para comprobar que nuestro ejemplo ofuscado sigue realizando la misma función extraemos el contenido de este contenedor y compilamos las clases extraídas, para ello:

```
D:\Temp\prueba\jar xf Test1res.jar
```

Obtenemos los *bytecodes* de las dos clases originales ofuscadas: "*A.class*" (1.295 *bytes*) y "*B.class*" (338 *bytes*). Comprobamos que tienen un mayor tamaño que los originales. Esto es debido al proceso de ofuscación de control y a la encriptación.

```
D:\Temp\prueba\java A
HOLA MUNDO
PRUEBA DE OFUSCACION
Ejemplo=1
```

Sigue realizando la misma función a pesar de haber ofuscado con las opciones más severas.

4) A continuación comparamos los bytecodes ofuscados obtenidos con los originales para poder así evaluar el proceso de ofuscación llevado a cabo por la aplicación Smokescreen. Ver programas 3.26 y 3.27.

```
D:\Temp\prueba\javap –c –p –l A
D:\Temp\prueba\javap –c –p –l B
```

Programa 3.26. Bytecode A.class.

```
public class A extends java.lang.Object{
public B A;

protected java.lang.String B;

private int[] C;

static java.lang.String _0;

static {};
  Code:
   0:   ldc     #1; //String r]RZG[X\n∩x{v↨zbysxgebruv↨sr↨xqbdtvt~xy
   2:   invokevirtual   #86; //Method java/lang/String.toCharArray:()[C
   5:   iconst_0
   6:   dup2
   7:   dup2
   8:   caload
   9:   bipush  55
   11:  ixor
   12:  i2c
   13:  castore
   14:  iconst_1
   15:  iadd
   16:  dup
   17:  bipush  38
   19:  if_icmplt       6
   22:  pop
   23:  invokestatic  #90; //Method java/lang/String.copyValueOf:([C)Ljava/lan
g/String;
   26:  putstatic       #95; //Field _0:Ljava/lang/String;
   29:  invokestatic    #75; //Method java/lang/System.currentTimeMillis:()J
   32:  dup2
   33:  ldc2_w  #68; //long 11134643087191
   36:  lcmp
   37:  dup
   38:  ifgt    37
   41:  pop
   42:  ldc2_w  #70; //long 11121683087071
   45:  lcmp
```

```
   46:  dup
   47:  iflt    46
   50:  pop
   51:  return
   52:  athrow
  Exception table:
   from   to  target type
    38    45    52   any


public static void main(java.lang.String[]);
  Code:
   0:   new     #2; //class A
   3:   dup
   4:   invokespecial   #16; //Method "<init>":()V
   7:   pop
   8:   return


public A();
  Code:
   0:   aload_0
   1:   invokespecial   #17; //Method java/lang/Object."<init>":()V
   4:   aload_0
   5:   new     #18; //class B
   8:   dup
   9:   invokespecial   #19; //Method B."<init>":()V
   12:  putfield        #21; //Field A:LB;
   15:  aload_0
   16:  getstatic       #95; //Field _0:Ljava/lang/String;
   19:  iconst_0
   20:  iconst_0
   21:  invokevirtual #82; //Method java/lang/String.substring:(II)Ljava/lang/
String;
   24:  putfield        #25; //Field B:Ljava/lang/String;
   27:  aload_0
   28:  bipush  10
   30:  newarray int
   32:  putfield        #27; //Field C:[I
   35:  getstatic    #33; //Field java/lang/System.out:Ljava/io/PrintStream;
   38:  aload_0
   39:  getfield        #21; //Field A:LB;
   42:  getstatic       #95; //Field _0:Ljava/lang/String;
   45:  bipush  8
   47:  bipush  18
   49:  invokevirtual #82; //Method java/lang/String.substring:(II)Ljava/lang/
String;
   52:  invokevirtual #38; //Method B.A:(Ljava/lang/String;)Ljava/lang/String;
   55:  invokevirtual #44; //Method java/io/PrintStream.println:(Ljava/lang/St
ring;)V
   58:  getstatic     #33; //Field java/lang/System.out:Ljava/io/PrintStream;
   61:  aload_0
   62:  invokevirtual   #47; //Method A:()Ljava/lang/String;
   65:  invokevirtual #44; //Method java/io/PrintStream.println:(Ljava/lang/St
ring;)V
   68:  getstatic     #33; //Field java/lang/System.out:Ljava/io/PrintStream;
   71:  new     #49; //class StringBuffer
   74:  dup
   75:  invokespecial   #50; //Method java/lang/StringBuffer."<init>":()V
   78:  getstatic       #95; //Field _0:Ljava/lang/String;
   81:  iconst_0
   82:  bipush  8
   84:  invokevirtual #82; //Method java/lang/String.substring:(II)Ljava/lang/
String;
   87:  invokevirtual #56; //Method java/lang/StringBuffer.append:(Ljava/lang/
String;)Ljava/lang/StringBuffer;
   90:  aload_0
   91:  invokespecial   #59; //Method B:()I
```

```
   94:  invokevirtual #62; //Method java/lang/StringBuffer.append:(I)Ljava/lan
g/StringBuffer;
   97:  invokevirtual #65; //Method java/lang/StringBuffer.toString:()Ljava/la
ng/String;
   100: invokevirtual #44; //Method java/io/PrintStream.println:(Ljava/lang/St
ring;)V
   103: return


protected java.lang.String A();
  Code:
   0:   aload_0
   1:   getstatic       #95; //Field _0:Ljava/lang/String;
   4:   bipush  18
   6:   bipush  38
   8:   invokevirtual #82; //Method java/lang/String.substring:(II)Ljava/lang/
String;
   11:  putfield        #25; //Field B:Ljava/lang/String;
   14:  aload_0
   15:  getfield        #25; //Field B:Ljava/lang/String;
   18:  areturn


private int B();
  Code:
   0:   aload_0
   1:   getfield        #27; //Field C:[I
   4:   iconst_0
   5:   iconst_1
   6:   iastore
   7:   aload_0
   8:   getfield        #27; //Field C:[I
   11:  iconst_0
   12:  iaload
   13:  ireturn
}
```

Programa 3.27. Bytecode B.class.

```
class B extends java.lang.Object{
static {};
  Code:
   0:   ldc2_w  #12; //long 11134643087131
   3:   invokestatic    #21; //Method java/lang/System.currentTimeMillis:()J
   6:   lcmp
   7:   ldc2_w  #14; //long 11121683087131
   10:  invokestatic    #21; //Method java/lang/System.currentTimeMillis:()J
   13:  lcmp
   14:  dup
   15:  ifgt    14
   18:  pop
   19:  dup
   20:  iflt    19
   23:  pop
   24:  return
   25:  athrow
  Exception table:
   from   to  target type
    15    18    25   any

public B();
  Code:
   0:   iconst_1
   1:   aload_0
   2:   invokespecial   #9; //Method java/lang/Object."<init>":()V
   5:   istore_1
```

```
    6:   return

public java.lang.String A(java.lang.String);
  Code:
   0:   aload_1
   1:   areturn
}
```

En base a la comparación de estos bytecodes ofuscados y los originales podemos evaluar la métrica. Ver tabla 3.11.

Hemos de destacar que a pesar de especificar la opción de encriptado, los *strings* de nuestro ejemplo 1 seguían apareciendo sin encriptar, por tanto en el aparatado individual de la métrica referente a encriptación hemos puntuado con un 0.

Como parte de la ofuscación de control se han añadido nuevos campos que si aparecen encriptados. Sin embargo, este no era el propósito que perseguíamos.

MÉTRICA		**EJEMPLO 1 (sin)**		**EJEMPLO 1(ofuscado)**		**PUNTUACIÓN**
		Test1class	*Hola.class*	*A.class*	*B.class*	
Public	**Clase**					1
	Métodos	main	GetHola, Hola	main, A	A,B	
	Campos	hello		A		
Protected	**Métodos**	pruebaOfus		A		1
	Campos	aux		B		
Private	**Métodos**	getVector		B		1
	Campos	vector		C		
Borrado Numero Linea		Numero de Linea		Borrado		1
Encriptado de String		PRUEBA DE OFUSCACION		PRUEBA DE OFUSCACION		0
Desestructuracion de Datos		vector[]		C[]		0
Puntuación para Ofuscación de Layout y Datos						4

Tabla 3.11. Evaluación de ofuscación de Estructura y Datos para el ejemplo 1.

3.2.7.3. Ofuscación de Ejemplo 2

Smokescreen si proporciona ofuscación de Control. Este proceso de ofuscación se basa en : "*Shuffle Stack Operations*", que realiza cambios en el orden de ejecución de instrucciones; "*Add Fake Exceptions*", introduce en el *bytecode* bloques de excepciones que se superponen a los bloques de control existentes; y "*Change Switch Statements*", cambia el flujo de control en sentencias de tipo *switch*.

Para evaluar esta ofuscación de control hemos diseñado el ejemplo 2. Los pasos que hemos seguido han sido los siguientes:

1) Generamos el archivo de directivas. Destacar que en este caso como entrada hemos considerado un directorio que contiene el bytecode del ejemplo 2 "*Test2.class*". Por lo demás el archivo de opciones es análogo al empleado para la ofuscación del ejemplo 1. Ver programa 3.28.

Programa 3.28. Directivas de control para Smokescreen, myDirectivas.txt.

```
* source_directory D:\Temp\prueba\
* destination_directory D:\Temp\res\
* superclass_path C:\Archivos de programa\j2sdk1.4.2_07\jre\lib\rt.jar

* use_class_loader_for_superclasses
* log_changes
* overwrite_classfiles

* classes all_classes
* methods all_methods
* fields all_fields

* bytecode add_fake_exceptions
* bytecode shuffle_stack_operations
* bytecode change_switch_statements
* bytecode encrypt_strings

-method Test2.main
```

Notar que hemos preservado el método *main* del proceso de ofuscación.

2) Ejecutamos la aplicación desde línea de comandos:

D:\Temp\SmokescreenSetup341_Eval\java –jar Smokescreen30_Eval.jar –nogui –directives myDirectives.txt

3) Comprobamos que no han ocurrido errores examinando el archivo "*ChangeLog.txt*". Ver programa 3.29.

Programa 3.29. .Archivo ChangeLog.txt.

```
Smokescreen 3.41 (Evaluation edition)

Using command line

JVM info: Sun Microsystems Inc. 1.4.2_07
OS info: Windows XP x86 5.1

---------------------- Beginning Obfuscation ----------------------
Wednesday, March 30, 2005 12:15:47 PM CEST

Using Directives file 'D:\Temp\SmokescreenSetup341_Eval\mydirectives.txt'
Source is 'D:\Temp\prueba'
Destination is 'D:\Temp\res'

Will use class loader to find super classes

Changes and removals will be recorded
Destination Class, Jar or Zip files may be overwritten
Unused methods will not be removed
```

```
Unused fields will not be removed
All class names will be changed
All method names will be changed
All field names will be changed
Stack operations will be shuffled
Fake exceptions will be added
Switch statements will be obfuscated
Strings will be encrypted
Number of selected classes is 1
Total number of classes scanned is 2

Class: 'Test2' changed to 'A'
Method: 'Test2.<init>' '()V' not changed
Method: 'Test2.main' '([Ljava/lang/String;)V' not changed
---------------------- Ending Obfuscation ----------------------
```

Para este segundo ejemplo, al tener solamente una clase de entrada obtuvimos el bytecode de salida directamente: "*A.class*" (1.124 *bytes*). Comprobamos que sigue realizando la misma función:

D:\Temp\res\java A
Numero de bucle > 1
Numero de bucle=1

4) Por último, compramos el bytecode ofuscado mediante Smokescreen con el código original, para ello tecleamos en consola:

D:\Temp\res\javap –c –p –l A

Programa 3.30. Bytecode A.class.

```
public class A extends java.lang.Object{
static java.lang.String _0;

public static void main(java.lang.String[]);
  Code:
   0:   iconst_0
   1:   iconst_0
   2:   istore_1
   3:   istore_2
   4:   iload_2
   5:   iconst_2
   6:   if_icmpge       88
   9:   iinc    2, 1
   12:  iinc    1, 1
   15:  goto    4
   18:  getstatic      #17; //Field java/lang/System.out:Ljava/io/PrintStream;
   21:  invokevirtual   #32; //Method java/io/PrintStream.println:()V
   24:  getstatic      #17; //Field java/lang/System.out:Ljava/io/PrintStream;
   27:  new     #34; //class StringBuffer
   30:  dup
   31:  invokespecial   #35; //Method java/lang/StringBuffer."<init>":()V
   34:  getstatic       #77; //Field _0:Ljava/lang/String;
   37:  bipush  39
   39:  bipush  55
   41:  invokevirtual #63; //Method java/lang/String.substring:(II)Ljava/lang/
String;
   44:  invokevirtual #41; //Method java/lang/StringBuffer.append:(Ljava/lang/
String;)Ljava/lang/StringBuffer;
   47:  iload_1
   48:  invokevirtual #44; //Method java/lang/StringBuffer.append:(I)Ljava/lan
g/StringBuffer;
```

```
   51:  invokevirtual #48; //Method java/lang/StringBuffer.toString:()Ljava/la
ng/String;
   54:  invokevirtual #30; //Method java/io/PrintStream.println:(Ljava/lang/St
ring;)V
   57:  return
   58:  athrow
   59:  getstatic      #17; //Field java/lang/System.out:Ljava/io/PrintStream;
   62:  getstatic       #77; //Field _0:Ljava/lang/String;
   65:  bipush  19
   67:  bipush  39
   69:  invokevirtual #63; //Method java/lang/String.substring:(II)Ljava/lang/
String;
   72:  invokevirtual #30; //Method java/io/PrintStream.println:(Ljava/lang/St
ring;)V
   75:  iload_1
   76:  iconst_1
   77:  if_icmple       18
   80:  iload_1
   81:  iconst_1
   82:  isub
   83:  istore_1
   84:  goto    75
   87:  athrow
   88:  iload_1
   89:  iconst_1
   90:  if_icmple       59
   93:  getstatic      #17; //Field java/lang/System.out:Ljava/io/PrintStream;
   96:  getstatic       #77; //Field _0:Ljava/lang/String;
   99:  iconst_0
   100: bipush  19
   102: invokevirtual #63; //Method java/lang/String.substring:(II)Ljava/lang/
String;
   105: invokevirtual #25; //Method java/io/PrintStream.print:(Ljava/lang/Stri
ng;)V
   108: goto    75
  Exception table:
   from   to  target type
     5     9    87   any
    76    83    58   any


static {};
  Code:
   0:   ldc #1; //String vMU]JW↑\]↑ZM[T]↑♠↑\tvMU]JW↑\]↑ZM[T]↑♦♣↑\tvMU]JW↑\]↑
ZM[T]♣
   2:   invokevirtual   #67; //Method java/lang/String.toCharArray:()[C
   5:   iconst_0
   6:   dup2
   7:   dup2
   8:   caload
   9:   bipush  56
   11:  ixor
   12:  i2c
   13:  castore
   14:  iconst_1
   15:  iadd
   16:  dup
   17:  bipush  55
   19:  if_icmplt       6
   22:  pop
   23:  invokestatic  #71; //Method java/lang/String.copyValueOf:([C)Ljava/lan
g/String;
   26:  putstatic       #77; //Field _0:Ljava/lang/String;
   29:  ldc2_w  #49; //long 11134643087131
   32:  invokestatic    #56; //Method java/lang/System.currentTimeMillis:()J
   35:  lcmp
   36:  ldc2_w  #51; //long 11121683087131
   39:  invokestatic    #56; //Method java/lang/System.currentTimeMillis:()J
```

```
   42:  lcmp
   43:  dup
   44:  ifgt    43
   47:  pop
   48:  dup
   49:  iflt    48
   52:  pop
   53:  return
   54:  athrow
  Exception table:
   from   to  target type
    44    47    54   any


public A();
  Code:
   0:   aload_0
   1:   invokespecial   #9; //Method java/lang/Object."<init>":()V
   4:   return
}
```

Observamos que el bytecode ofuscado es mucho más complejo que el original, lo que nos da una idea de que la ofuscación de control de flujo ha sido efectiva. El coste de esta ofuscación es obtener un bytecode cuya ejecución es más lenta y el tamaño del mismo es ligeramente mayor: "*Test2.class*" (799 *bytes*) en comparación con "*A.class*" (1.124 *bytes*).

3.2.7.4. Conclusión

- Smokescreen puede realizar ofuscación de Estructura, renombrando identificadores de clases y miembros de clases *Public*, *Protected* y *Private*. Además borra toda información redundante, como comentarios, miembros si uso, etc.

- Es capaz de realizar ofuscación de Datos como puede verse en la tabla 3.11, proporciona opciones de encriptado, aunque en este caso hemos comprobado que siguen sin encriptar los strings de nuestro bytecode. Por tanto hemos puntuado con un cero la métrica individual. No realiza desestructuración de datos.

- Después de ser ofuscado con las opciones más severas, nuestro código seguía realizando la misma función. Los dos ejemplos seguían funcionando.

- Proporciona mecanismos para ofuscación de Control de Flujo. Es la principal ventaja con respecto a otras aplicaciones.

- Se proporciona un manual de usuario muy detallado en la página web asociada a la aplicación.

- Destacar que la ofuscación de Control y la encriptación producen *bytecodes* de mayor peso que los originales.

3.2.8. JOGA

3.2.8.1. Introducción

JoGa principalmente es un optimizador de código que permite reducir el tamaño de aplicaciones, *applets* y *API's* Java. Analiza el programa completo y determina clases, métodos y campos sin uso que pueden ser borrados. Además proporciona técnicas de optimización con el objetivo final de reducir el tamaño de nuestra aplicación, consiguiéndose reducciones del 80%.

- Actualmente implementa las siguientes técnicas de optimizado:
- Borra clases, métodos y campos sin uso.
- Borra bytecode superfluo de métodos.
- Asocia clases en una sola.
- Proporciona inlined para métodos.
- Ofuscación mediante renombramiento de paquetes, clases y miembros de clases.
- Trabaja con archivos ".jar" o ".zip".
- Borra información redundante.

3.2.8.2. Ofuscación de Ejemplo 1

Los pasos que hemos seguido para la ofuscación de este ejemplo y la posterior evaluación del proceso han sido los siguientes:

1) En primer lugar tenemos que especificar la aplicación a tratar. Para ello, en el menú de nuestra GUI nos vamos a *Project > add classes*; y cargamos el archivo en cuestión, en nuestro caso "*Test1.jar*". Debemos destacar que JoGa permite tratar un directorio, clases o archivos "*.jar*", "*.zip*".

2) A continuación configuramos las opciones de optimización, compactación y ofuscación; para ello navegamos por el menú *Optimazation*, seleccionando las opciones que deseemos. Ver apartado de utilización de la aplicación. Se han seleccionado las opciones más severas.

3) Comprobamos que no ha ocurrido error alguno durante el proceso llevado a cabo. Para ello inspeccionamos los ficheros generados. Ver programas 3.31 y 3.32.

Programa 3.31. Archivo que refleja incidencias del proceso, JoGaLog.txt.

```
output directory: D:\Temp\prueba\JoGaDIR

analysed totally (i.e. plus system classes):

#classes: 10
#methods: 153
#fields: 19
#methodCalls: 12
#fieldCalls: 7

choosen optimizations:
----------------------
   remove debug infos
   compress constant pools
   merge classes
   inline methods
   remove classes
   remove methods
   remove fields
   remove superflouse byte code
   rename packages
   rename classes
   rename methods
   rename fields
   optimize byte code:
     - optimize local variable order
     - optimize local variable compression
     - analyse local variable liveness
     - analyse constant values
     - optimize dead local variables

REMOVED CLASSES:
================
   none

REMOVED METHODS:
================
   none

REMOVED SUPERFLUOUS BYTE CODE:
==============================
   none

REMOVED FIELDS:
===============
   none

MERGED CLASSES:
===============
   none

INLINED METHODS:
================
   none

OPTIMIZED BYTE CODE:
====================
   optimized method code: Test1.main([Ljava/lang/String;)V
   optimized method code: Test1.<init>()V
   optimized method code: Test1.pruebaOfus()Ljava/lang/String;
   optimized method code: Test1.getVector()I
   optimized method code: Hola.<init>()V
   optimized method code: Hola.getHola(Ljava/lang/String;)Ljava/lang/String;

RENAMED PACKAGES:
=================
   none
```

```
RENAMED CLASSES:
================
   class: Hola --> a

RENAMED METHODS:
================
   meth: Hola.getHola(Ljava/lang/String;)Ljava/lang/String; --> a
   meth: Test1.getVector()I --> a
   meth: Test1.pruebaOfus()Ljava/lang/String; --> a

RENAMED FIELDS:
===============
   field: Test1.aux --> b
   field: Test1.hello --> a
   field: Test1.vector --> c

REMOVED DEBUG INFOS:
====================
   from class: Test1
   from class: Hola
```

Programa 3.32. Archivo de resultados del proceso, JoGaResults.txt.

```
#################### JOGA - RESULTFILE ####################

optimised: D:\Temp\prueba
2 files processed

CLASSES  before:2  after:2  => removed:0
METHODS  before:6  after:6  => removed:0
FIELDS   before:3  after:3  => removed:0

SIZE OF ZIP/JAR FILE  before:1425  after:1109  => 77%
SIZE OF ALL CLASS FIELS  before:1386  after:1091  => 78%

SIZE PER CLASS:
---------------

class: Hola  before:284  after:185 => 65%
class: Test1  before:1102  after:906 => 82%
```

Observamos el proceso de reducción de tamaño de las aplicaciones, así como de los *bytecodes* en el archivo de resultados. Se consiguen reducciones entorno al 80%. La eficacia de esta aplicación es sorprendente en este aspecto. No debemos olvidar que JoGa es ante todo un optimizador y compactador de *bytecode*.

4) Comprobamos que la aplicación optimizada y ofuscada sigue realizando la misma función que el *bytecode* original. Como salida del proceso hemos seleccionado que se generase un archivo contenedor "*Test1res.jar*". Este contenedor se crea por defecto dentro de un directorio cuya ruta hemos especificado previamente en el menú *Outputs*. Extraemos su contenido y compilamos mediante las instrucciones:

```
D:\Temp\prueba\JoGaDIR\jar xf Test1res.jar
D:\Temp\prueba\JoGaDIR\java Test1
HOLA MUNDO
PRUEBA DE OFUSCACION
Ejemplo=1
```

Efectivamente sigue realizando la misma función a pesar de haber empleado las opciones más severas de optimización y ofuscación. Notar que hemos preservado el nombre de la clase que contiene el método *main*.

5) Por último evaluamos el proceso mediante la comparación con los *bytecodes* originales. Ver programas 3.23 y 3.24.

D:\Temp\prueba\javap –c –p –l Test1
D:\Temp\prueba\javap –c –p –l a

Programa 3.23. Bytecode Test1.class.

```
public class Test1 extends java.lang.Object{
public a a;

protected java.lang.String b;

private int[] c;

public static void main(java.lang.String[]);
  Code:
   0:   new     #1; //class Test1
   3:   dup
   4:   invokespecial   #2; //Method "<init>":()V
   7:   pop
   8:   return

public Test1();
  Code:
   0:   aload_0
   1:   invokespecial   #3; //Method java/lang/Object."<init>":()V
   4:   aload_0
   5:   new     #49; //class a
   8:   dup
   9:   invokespecial   #50; //Method a."<init>":()V
   12:  putfield        #53; //Field a:La;
   15:  aload_0
   16:  ldc     #4; //String
   18:  putfield        #56; //Field b:Ljava/lang/String;
   21:  aload_0
   22:  bipush  10
   24:  newarray int
   26:  putfield        #59; //Field c:[I
   29:  getstatic       #5; //Field java/lang/System.out:Ljava/io/PrintStream;
   32:  aload_0
   33:  getfield        #53; //Field a:La;
   36:  ldc     #6; //String HOLA MUNDO
   38:  invokevirtual #61; //Method a.a:(Ljava/lang/String;)Ljava/lang/String;
   41:  invokevirtual #7; //Method java/io/PrintStream.println:(Ljava/lang/Str
ing;)V
   44:  getstatic     #5; //Field java/lang/System.out:Ljava/io/PrintStream;
   47:  aload_0
   48:  invokevirtual   #63; //Method a:()Ljava/lang/String;
   51:  invokevirtual #7; //Method java/io/PrintStream.println:(Ljava/lang/Str
ing;)V
   54:  getstatic       #5; //Field java/lang/System.out:Ljava/io/PrintStream;
   57:  new     #8; //class StringBuffer
   60:  dup
   61:  invokespecial   #9; //Method java/lang/StringBuffer."<init>":()V
   64:  ldc     #10; //String Ejemplo=
   66:  invokevirtual #11; //Method java/lang/StringBuffer.append:(Ljava/lang/
String;)Ljava/lang/StringBuffer;
   69:  aload_0
   70:  invokespecial   #65; //Method a:()I
```

```
   73:  invokevirtual #12; //Method java/lang/StringBuffer.append:(I)Ljava/lan
g/StringBuffer;
   76:  invokevirtual #13; //Method java/lang/StringBuffer.toString:()Ljava/la
ng/String;
   79:  invokevirtual #7; //Method java/io/PrintStream.println:(Ljava/lang/Str
ing;)V
   82:  return

protected java.lang.String a();
  Code:
   0:   aload_0
   1:   ldc     #14; //String PRUEBA DE OFUSCACION
   3:   putfield        #56; //Field b:Ljava/lang/String;
   6:   aload_0
   7:   getfield        #56; //Field b:Ljava/lang/String;
   10:  areturn

private int a();
  Code:
   0:   aload_0
   1:   getfield        #59; //Field c:[I
   4:   iconst_0
   5:   iconst_1
   6:   iastore
   7:   aload_0
   8:   getfield        #59; //Field c:[I
   11:  iconst_0
   12:  iaload
   13:  ireturn

}
```

Programa 3.24. Bytecode a.class.

```
class a extends java.lang.Object{
public a();
  Code:
   0:   aload_0
   1:   invokespecial   #1; //Method java/lang/Object."<init>":()V
   4:   return

public java.lang.String a(java.lang.String);
  Code:
   0:   aload_1
   1:   areturn
}
```

En base a la comparación de estos *bytecodes* ofuscados y los originales podemos evaluar la métrica. Ver tabla 3.12.

3.2.8.3. Ofuscación de Ejemplo 2

JoGa no proporciona como opción ofuscación de Control de Flujo. Por tanto puntuamos con un 0 la métrica individual correspondiente. No debemos olvidar que JoGa no es un ofuscador propiamente dicho, es un optimizador de código que incluye opciones básicas de ofuscación.

MÉTRICA		EJEMPLO 1 (sin)		EJEMPLO 1(ofuscado)		PUNTUACIÓN
		Test1class	*Hola.class*	*Test1.class*	*a.class*	
Public	**Clase**					1
	Métodos	main	GetHola, Hola	main	a,a	
	Campos	hello		a		
Protected	**Métodos**	pruebaOfus		a		1
	Campos	aux		b		
Private	**Métodos**	getVector		a		1
	Campos	vector		c		
Borrado Numero Linea		Numero de Linea		Borrado		1
Encriptado de String		PRUEBA DE OFUSCACION		PRUEBA DE OFUSCACION		0
Desestructuracion de Datos		vector[]		c[]		0
Puntuación para Ofuscación de Layout y Datos						4

Tabla 3.12. Evaluación de ofuscación de Estructura y Datos para el ejemplo 1.

3.2.8.4. Conclusiones

JoGa puede realizar ofuscación de Estructura, renombrando identificadores de clases y miembros de clases *Public*, *Protected* y *Private*. Se proporcionan mecanismos de renombrado severo. Además borra toda información redundante, como comentarios, miembros si uso, etc.

- No es capaz de realizar ofuscación de Datos como puede verse en la tabla 3.12.
- Después de ser ofuscado con las opciones más severas, nuestro código seguía realizando la misma función.
- No proporciona mecanismos para ofuscación de Control de Flujo.
- Aún no siendo una aplicación específica de ofuscación, realiza este proceso de manera aceptable y muy similar a ofuscadores analizados en este estudio de arte.
- No se proporciona manual de usuario, si bien es bastante fácil de manejar gracias a una interfaz gráfica sencilla.
- No es ejecutable desde línea de comandos.
- Destacar que es una aplicación *Opensource* y por tanto *freeware*.

3.2.9. JZIPPER

Este proyecto parece haber sido abandonado. La última versión liberada de la aplicación es de 1999. Al ejecutarla nos da el siguiente mensaje:

Jzipper software has expired, please return to its website to retrieve the latest version.

Hemos vuelto a bajar la última versión disponible pero nos encontramos con el mismo problema. Por tanto no hemos podido evaluarla.

3.2.10. JOBFUSCATE

3.2.10.1. Introducción

Jobfuscate permite a desarrolladores de software proteger sus archivos "*.class*" Java. De otra manera, sin protección, cualquiera podría usar un descompilador para efectuar ingeniería inversa y obtener nuestro código Java original.

Destacar que esta aplicación es una solución comercial. Solo disponemos de una versión de prueba para la que no están disponibles todas las opciones de ofuscación. La versión completa cuesta 100 $. Con esta versión de prueba se nos advierte que realmente no estamos realizando ofuscación de Estructura, es decir, el renombrado que lleva a cabo no es tal, se renombran todas las clases y miembros de clases pero sus nombres originales siguen apareciendo como apéndice. Lo veremos más adelante al ofuscar el ejemplo 1.

Desde la página web asociada se advierte que de esta manera el usuario que adquiera la versión de prueba, puede comprobar como trabaja la aplicación y que las clases ofuscadas (aunque no realmente) siguen realizando la misma función.

3.2.10.2. Ofuscación de Ejemplo 1

Hemos realizado los siguientes pasos para la ofuscación de nuestro ejemplo: (no debemos olvidar que ésta es una versión de prueba)

1) En primer lugar hemos generado el *scriptfile* que contendrá las opciones de configuración para el proceso de ofuscación llevado a cabo por Jobfuscate. Ver programa 3.25.

Programa 3.44. Script de configuración, Config.txt.

```
-log:D:\Temp\prueba\Jobfus.log
-out:D:\Temp\prueba\Testres1.jar
-xm:D:\Temp\prueba\Test1.main
-sys:C:\Archivos de programa\j2sdk1.4.2_07\jre\lib\rt.jar
```

Destacar que hemos redireccionado la salida por defecto a un archivo que reflejará posibles incidencias en el proceso de ofuscación llevado a cabo. Además en este archivo se especifica el proceso de renombrado que se ha seguido. Esto nos puede interesar para comprobar que la versión de prueba realmente no realiza ofuscación de Estructura.

2) Ejecutamos nuestra aplicación:

D:\Temp\Jobfuscate 3.0\jobfuscate @Config Test1

3) Comprobamos que no ha ocurrido ningún error durante el proceso de ofuscación, para ello inspeccionamos el *logfile* generado. Ver programa 3.26.

Programa 3.26. Archivo donde se reflejan posibles incidencias, Jobfus.log.

```
ON: D:\Temp\Jobfuscate 3.0\Test1.class
ON: D:\Temp\Jobfuscate 3.0\Hola.class
ADDING[1]: Hola
*** pushed: toString()
*** pushed: toString()
*** pushed: charAt(I)
*** pushed: length()
*** pushed: subSequence(II)
*** pushed: hasMoreElements()
*** pushed: nextElement()
*** pushed: registerNatives()
*** pushed: hashCode()
*** pushed: put(Ljava/lang/Object;Ljava/lang/Object;)
*** pushed: clone()
*** pushed: equals(Ljava/lang/Object;)
*** pushed: get(Ljava/lang/Object;)
*** pushed: size()
*** pushed: elements()
*** pushed: remove(Ljava/lang/Object;)
*** pushed: keys()
*** pushed: isEmpty()
*** pushed: hashCode()
*** pushed: put(Ljava/lang/Object;Ljava/lang/Object;)
*** pushed: equals(Ljava/lang/Object;)
*** pushed: get(Ljava/lang/Object;)
*** pushed: size()
*** pushed: values()
*** pushed: remove(Ljava/lang/Object;)
*** pushed: clear()
*** pushed: containsKey(Ljava/lang/Object;)
*** pushed: containsValue(Ljava/lang/Object;)
*** pushed: entrySet()
*** pushed: isEmpty()
*** pushed: keySet()
*** pushed: putAll(Ljava/util/Map;)
*** pushed: compareTo(Ljava/lang/Object;)
*** pushed: toString()
*** pushed: charAt(I)
*** pushed: length()
*** pushed: subSequence(II)
**push/pull
**push/pull
Test1 <- Test1
  AO_hello      <- hello
  BO_aux        <- aux
  CO_vector     <- vector
  *             <- main()                                    // main
  EO_pruebaOfus <- pruebaOfus()
  FO_getVector  <- getVector()
ZO_Hola <- Hola
```

```
  DO_getHola <- getHola()
Writing files into D:\Temp\prueba\Testres1.jar...
jobfuscate finished in 150ms - 2 files, 0 errors, 0 warnings


  ****************************************************************
  *                       RUNNING IN TRIAL MODE                  *
  ****************************************************************
  * All field, method, and class names are changed, just like they *
  * are in the registered version, but the names are APPENDED with *
  * the original name.  This intentially provides NO OBFUSCATION  *
  * PROTECTION, but it allows you to: (1) see what names would be  *
  * obfuscated to, and (2) verify that your class files continue   *
  * to work after being changed by jobfuscate.  Please purchase    *
  * the registered version to obtain full obfuscation protection.  *
  ****************************************************************
  *        http://www.duckware.com  -  support@duckware.com        *
  ****************************************************************
```

Comprobamos que efectivamente no se realiza un renombrado efectivo, pues los nombres de clases y miembros de clases aparecen con apéndices de las nuevas etiquetas. Esto es debido a que trabajamos con una versión de prueba como hemos comentado anteriormente.

Destacar que no se ha producido error alguno. Como salida hemos obtenido un contenedor "*Test1res.jar*" (1.414 *bytes*).

4) A continuación extraemos los *bytecodes* ofuscados del contenedor de salida y comprobamos que siguen realizando la misma función. Para ello:

```
D:\Temp\prueba\jar xf Test1res.jar
```

Obtenemos dos archivos: "*Test1.class*" (986 *bytes*) y "*ZO_Hola.class*" (206 *bytes*). Observamos una reducción del tamaño de los *bytecodes* en comparación con los originales. Hemos conseguido por tanto un código más compacto.

```
D:\Temp\prueba\java Test1
HOLA MUNDO
PRUEBA DE OFUSCACION
Ejemplo=1
```

Efectivamente sigue realizando la misma función.

5) Por último comparamos los bytecodes ofuscados obtenidos con los originales para poder así evaluar la aplicación de acuerdo a la métrica diseñada. Ver programa 3.27 y 3.28.

```
D:\Temp\prueba\javap –c –p –l Test1
D:\Temp\prueba\javap –c –p –l ZO_Hola
```

Programa 3.27. Bytecode Test1.class.

```
public class Test1 extends java.lang.Object{
public ZO_Hola AO_hello;

protected java.lang.String BO_aux;

private int[] CO_vector;

public static void main(java.lang.String[]);
  Code:
   0:   new     #1; //class Test1
   3:   dup
   4:   invokespecial   #2; //Method "<init>":()V
   7:   pop
   8:   return


public Test1();
  Code:
   0:   aload_0
   1:   invokespecial   #3; //Method java/lang/Object."<init>":()V
   4:   aload_0
   5:   new     #4; //class ZO_Hola
   8:   dup
   9:   invokespecial   #5; //Method ZO_Hola."<init>":()V
   12:  putfield        #6; //Field AO_hello:LZO_Hola;
   15:  aload_0
   16:  ldc     #7; //String
   18:  putfield        #8; //Field BO_aux:Ljava/lang/String;
   21:  aload_0
   22:  bipush  10
   24:  newarray int
   26:  putfield        #9; //Field CO_vector:[I
   29:  getstatic      #10; //Field java/lang/System.out:Ljava/io/PrintStream;
   32:  aload_0
   33:  getfield        #6; //Field AO_hello:LZO_Hola;
   36:  ldc     #11; //String HOLA MUNDO
   38:  invokevirtual #12; //Method ZO_Hola.DO_getHola:(Ljava/lang/String;)Lja
va/lang/String;
   41:  invokevirtual   #13; //Method
java/io/PrintStream.println:(Ljava/lang/St
ring;)V
   44:  getstatic     #10; //Field java/lang/System.out:Ljava/io/PrintStream;
   47:  aload_0
   48:  invokevirtual   #14; //Method EO_pruebaOfus:()Ljava/lang/String;
   51:  invokevirtual #13; //Method java/io/PrintStream.println:(Ljava/lang/St
ring;)V
   54:  getstatic      #10; //Field java/lang/System.out:Ljava/io/PrintStream;
   57:  new     #15; //class StringBuffer
   60:  dup
   61:  invokespecial   #16; //Method java/lang/StringBuffer."<init>":()V
   64:  ldc     #17; //String Ejemplo=
   66:  invokevirtual #18; //Method java/lang/StringBuffer.append:(Ljava/lang/
String;)Ljava/lang/StringBuffer;
   69:  aload_0
   70:  invokespecial   #19; //Method FO_getVector:()I
   73:  invokevirtual #20; //Method java/lang/StringBuffer.append:(I)Ljava/lan
g/StringBuffer;
   76:  invokevirtual #21; //Method java/lang/StringBuffer.toString:()Ljava/la
ng/String;
   79:  invokevirtual #13; //Method java/io/PrintStream.println:(Ljava/lang/St
ring;)V
   82:  return


protected java.lang.String EO_pruebaOfus();
  Code:
   0:   aload_0
```

```
   1:   ldc     #22; //String PRUEBA DE OFUSCACION
   3:   putfield        #8; //Field BO_aux:Ljava/lang/String;
   6:   aload_0
   7:   getfield        #8; //Field BO_aux:Ljava/lang/String;
   10:  areturn


private int FO_getVector();
  Code:
   0:   aload_0
   1:   getfield        #9; //Field CO_vector:[I
   4:   iconst_0
   5:   iconst_1
   6:   iastore
   7:   aload_0
   8:   getfield        #9; //Field CO_vector:[I
   11:  iconst_0
   12:  iaload
   13:  ireturn
}
```

Programa 3.28. Bytecode ZO_Hola.class.

```
class ZO_Hola extends java.lang.Object{
public ZO_Hola();
  Code:
   0:   aload_0
   1:   invokespecial   #1; //Method java/lang/Object."<init>":()V
   4:   iconst_1
   5:   istore_1
   6:   return


public java.lang.String DO_getHola(java.lang.String);
  Code:
   0:   aload_1
   1:   areturn
}
```

En base a la comparación de estos *bytecodes* ofuscados y los originales podemos evaluar la métrica. Ver tabla 3.13.

Hemos puntuado el renombramiento llevado a cabo porque aunque no sea del todo efectivo demuestra que la aplicación es capaz de realizar ofuscación de Estructura.

3.2.10.3. Ofuscación de Ejemplo 2

Jobfuscate no proporciona como opción ofuscación de Control de Flujo. Por tanto puntuamos con un 0 la métrica individual correspondiente. Para la versión de evaluación no se especifica como opción. Sin embargo tampoco parece que la versión registrada ofrezca ofuscación de Control a juzgar por la información que se ofrece al usuario en la página web asociada a la aplicación.

MÉTRICA		EJEMPLO 1 (sin)		EJEMPLO 1(ofuscado)		PUNTUACIÓN
		Test1class	*Hola.class*	*Test1.class*	*ZO_Hola.class*	
Public	**Clase**					1
	Métodos	main	GetHola, Hola	main	DO_GetHola, ZO_Hola	
	Campos	hello		AO_hello		
Protected	**Métodos**	pruebaOfus		EO_pruebaOfus		1
	Campos	aux		BO_aux		
Private	**Métodos**	getVector		FO_getVector		1
	Campos	vector		CO_vector		
Borrado Numero Linea		Numero de Linea		Borrado		1
Encriptado de String		PRUEBA DE OFUSCACION		PRUEBA DE OFUSCACION		0
Desestructuracion de Datos		vector[]		CO_Vector[]		0
Puntuación para Ofuscación de Layout y Datos						4

Tabla 3.13. Evaluación de ofuscación de Estructura y Datos para el ejemplo 1.

3.2.10.4. Conclusiones

- Jobfuscate puede realizar ofuscación de Estructura, renombrando identificadores de clases y miembros de clases Public, Protected y Private.

- En este caso el proceso de renombrado no es efectivo como hemos explicado anteriormente pero comprobamos que la versión de prueba lleva a cabo este cometido. Además borra toda información redundante, como comentarios, miembros si uso, etc.

- No es capaz de realizar ofuscación de Datos como puede verse en la tabla 3.13.
- Después de ser ofuscado con las opciones más severas que permite la versión de prueba, nuestro código seguía realizando la misma función.

- No proporciona mecanismos para ofuscación de Control de Flujo.

Se proporcionan unas breves indicaciones sobre su utilización en la página web asociada.

3.2.11. MARVINOBFUSCATOR

3.2.11.1. Introducción

Esta herramienta rescribe las aplicaciones Java de manera que las hace imposibles de descompilar y entender su funcionamiento interno. Algunas de las características más destacables de MarvinObfuscator se exponen a continuación:

- Prohíbe virtualmente la ingeniería inversa.
- El código ofuscado no es usualmente recompilable.
- Procesa cualquier aplicación, *applet* o *servlet*.
- El código resultante es compatible con cualquier JVM.
- Genera un archivo de salida "*.jar*" muy compacto.
- Borra información de paquetes: renombra identificadores de clases, métodos y campos.
- Encripta *strings*.
- Borra clases innecesarias.
- Borra toda información redundante.

3.2.11.2. Ofuscación de Ejemplo 1

Los pasos que hemos seguido para la ofuscación del ejemplo 1 y la posterior evaluación de la métrica han sido los siguientes:

1) Modificar el *script* de configuración que proporciona la aplicación indicando *path* de librerías para resolver posibles dependencias, así como el nombre de la clase principal de nuestra aplicación que tomará MavirObfuscator como punto de entrada para el proceso de ofuscación. Ver apartado de utilización.

2) Incluir este script en el directorio que contiene las clases de nuestro ejemplo 1: "*Test1.class*" y "*Hola.class*".

3) Ejecutar la aplicación MarvinObfuscator desde línea de comandos como se ha especificado en el apartado de utilización:

```
C:\MarvinObfuscator\obfuscate prueba Test1res.jar
```

4) Comprobar que se ha generado el archivo de salida y que no ha ocurrido error alguno durante el proceso de ofuscación. En este caso no se proporciona opción

para generar un *logfile* pero por la salida estándar se muestran las incidencias del proceso. Ver programa 3.29.

Programa 3.29. Incidencias del proceso llevado a cabo por MarvinObfuscator.

```
The Marvin Obfuscator 1.2b, (c) 2000-2001 by Dr. Java (www.drjava.de)
Pass 1
Pass 2
2 entries written to jar file, total size=1294, processing time: 150 ms
Saved Test1res.jar (1294 bytes)
```

5) Extraer el contenido del archivo de salida y comprobar que efectivamente los *bytecodes* ofuscados siguen realizando la misma función:

C:\MarvinObfuscator\jar xf Test1res.jar

Obtenemos los bytecodes de las dos clases ofuscadas: "*Test1.class*" (1.450 *bytes*) y "*super.class*" (219 *bytes*). Comprobamos que el tamaño de la clase principal se ha incrementado. Esto es debido a que MarvinObfuscator ofrece la opción de encriptado y esto supone una carga de procesado mayor.

C:\MarvinObfuscator\java Test1
HOLA MUNDO
PRUEBA DE OFUSCACION
Ejemplo=1

Sigue realizando la misma función. Por tanto el proceso de ofuscación llevado a cabo por MarvinObfuscator ha sido exitoso.

6) Por último y con el objetivo de evaluar la métrica diseñada hemos comparado los bytecodes ofuscados con los originales. El resultado se muestra en la tabla 3.14.

D:\Temp\prueba\javap –c –p –l Test1
D:\Temp\prueba\javap –c –p –l super

Programa 3.30. Bytecode Test1.class.

```
Compiled from ""
public class Test1 extends java.lang.Object{
public super Ka;

protected java.lang.String La;

private int[] Ma;

private static java.lang.String Na;
private static java.lang.String Oa;

private static java.lang.String Pa;

private static java.lang.String Qa;

public static void main(java.lang.String[]);
  Code:
   0:   new     #2; //class Test1
   3:   dup
   4:   invokespecial   #17; //Method "<init>":()V
   7:   pop
   8:   return
public Test1();
```

```
  Code:
   0:   aload_0
   1:   invokespecial   #20; //Method java/lang/Object."<init>":()V
   4:   aload_0
   5:   new     #22; //class super
   8:   dup
   9:   invokespecial   #24; //Method super."<init>":()V
   12:  putfield        #26; //Field Ka:Lsuper;
   15:  aload_0
   16:  getstatic       #31; //Field Na:Ljava/lang/String;
   19:  putfield        #33; //Field La:Ljava/lang/String;
   22:  aload_0
   23:  bipush  10
   25:  newarray int
   27:  putfield        #35; //Field Ma:[I
   30:  getstatic     #41; //Field java/lang/System.out:Ljava/io/PrintStream;
   33:  aload_0
   34:  getfield        #26; //Field Ka:Lsuper;
   37:  getstatic       #46; //Field Oa:Ljava/lang/String;
   40:  invokevirtual #50; //Method super._:(Ljava/lang/String;)Ljava/lang/Str
ing;
   43:  invokevirtual #56; //Method java/io/PrintStream.println:(Ljava/lang/St
ring;)V
   46:  getstatic      #41; //Field java/lang/System.out:Ljava/io/PrintStream;
   49:  aload_0
   50:  invokevirtual   #60; //Method a:()Ljava/lang/String;
   53:  invokevirtual #56; //Method java/io/PrintStream.println:(Ljava/lang/St
ring;)V
   56:  getstatic     #41; //Field java/lang/System.out:Ljava/io/PrintStream;
   59:  new     #62; //class StringBuffer
   62:  dup
   63:  invokespecial   #64; //Method java/lang/StringBuffer."<init>":()V
   66:  getstatic       #69; //Field Pa:Ljava/lang/String;
   69:  invokevirtual #73; //Method java/lang/StringBuffer.append:(Ljava/lang/
String;)Ljava/lang/StringBuffer;
   72:  aload_0
   73:  invokespecial   #77; //Method b:()I
   76:  invokevirtual #80; //Method java/lang/StringBuffer.append:(I)Ljava/lan
g/StringBuffer;
   79:  invokevirtual #83; //Method java/lang/StringBuffer.toString:()Ljava/la
ng/String;
   82:  invokevirtual #56; //Method java/io/PrintStream.println:(Ljava/lang/St
ring;)V
   85:  return


protected java.lang.String a();
  Code:
   0:   aload_0
   1:   getstatic       #88; //Field Qa:Ljava/lang/String;
   4:   putfield        #33; //Field La:Ljava/lang/String;
   7:   aload_0
   8:   getfield        #33; //Field La:Ljava/lang/String;
   11:  areturn


private int b();
  Code:
   0:   aload_0
   1:   getfield        #35; //Field Ma:[I
   4:   iconst_0
   5:   iconst_1
   6:   iastore
   7:   aload_0
   8:   getfield        #35; //Field Ma:[I
   11:  iconst_0
   12:  iaload
   13:  ireturn
```

```
private static java.lang.String a(java.lang.String);
  Code:
   0:   aload   0
   2:   invokevirtual   #93; //Method java/lang/String.length:()I
   5:   istore  1
   7:   iload   1
   9:   newarray char
   11:  astore  2
   13:  ldc     #94; //int 0
   15:  istore  3
   17:  iload   3
   19:  iload   1
   21:  if_icmpge       46
   24:  aload   2
   26:  iload   3
   28:  aload   0
   30:  iload   3
   32:  invokevirtual   #98; //Method java/lang/String.charAt:(I)C
   35:  ldc     #99; //int 59725
   37:  ixor
   38:  i2c
   39:  castore
   40:  iinc    3, 1
   43:  goto    17
   46:  new     #92; //class String
   49:  dup
   50:  aload   2
   52:  invokespecial   #102; //Method java/lang/String."<init>":([C)V
   55:  areturn

public static {};
  Code:
   0:   getstatic       #31; //Field Na:Ljava/lang/String;
   3:   invokestatic    #105; //Method
a:(Ljava/lang/String;)Ljava/lang/String;
   6:   putstatic       #31; //Field Na:Ljava/lang/String;
   9:   getstatic       #46; //Field Oa:Ljava/lang/String;
   12:  invokestatic    #105; //Method
a:(Ljava/lang/String;)Ljava/lang/String;
   15:  putstatic       #46; //Field Oa:Ljava/lang/String;
   18:  getstatic       #69; //Field Pa:Ljava/lang/String;
   21:  invokestatic    #105; //Method
a:(Ljava/lang/String;)Ljava/lang/String;
   24:  putstatic       #69; //Field Pa:Ljava/lang/String;
   27:  getstatic       #88; //Field Qa:Ljava/lang/String;
   30:  invokestatic    #105; //Method
a:(Ljava/lang/String;)Ljava/lang/String;
   33:  putstatic       #88; //Field Qa:Ljava/lang/String;
   36:  return
}
```

Programa 3.31. Bytecode super.class.

```
Compiled from ""
class super extends java.lang.Object{
public super();
  Code:
   0:   aload_0
   1:   invokespecial   #9; //Method java/lang/Object."<init>":()V
   4:   iconst_1
   5:   istore_1
   6:   return


public java.lang.String _(java.lang.String);
```

```
  Code:
   0:   aload_1
   1:   areturn
}
```

MÉTRICA		EJEMPLO 1 (sin)		EJEMPLO 1(ofuscado)		PUNTUACIÓN
		Test1class	*Hola.class*	*Test1.class*	*super.class*	
Public	**Clase**					1
	Métodos	main	GetHola, Hola	main, Test1	_, super	
	Campos	hello		Ka		
Protected	**Métodos**	pruebaOfus		a		1
	Campos	aux		La		
Private	**Métodos**	getVector		a, b		1
	Campos	vector		Ma		
Borrado Numero Linea		Numero de Linea		Borrado		1
Encriptado de String		PRUEBA DE OFUSCACION		Encriptado		1
Desestructuracion de Datos		vector[]		Ma[]		0
Puntuación para Ofuscación de Layout y Datos						5

Tabla 3.14. Evaluación de ofuscación de Estructura y Datos para el ejemplo 1.

3.2.11.3. Ofuscación de Ejemplo 2

MarvinObfuscator no proporciona como opción ofuscación de Control de Flujo. Por tanto puntuamos con un 0 la métrica individual correspondiente.

3.2.11.4. Conclusión

Smokescreen puede realizar ofuscación de Estructura, renombrando identificadores de clases y miembros de clases *Public*, *Protected* y *Private*. Además borra toda información redundante, como comentarios, miembros si uso, etc.

- Es capaz de realizar ofuscación de Datos como puede verse en la tabla 3.14, proporciona opciones de encriptado. Sin embargo no realiza desestructuración de datos.

- Después de ser ofuscado con las opciones más severas, nuestro código seguía realizando la misma función.

- No proporciona mecanismos para ofuscación de Control de Flujo.

- No se proporciona un manual de usuario detallado en la página web asociada a la aplicación. Quizás sea una de las principales desventajas con respecto a otras aplicaciones estudiadas en esta comparativa.

- Destacar que la encriptación produce *bytecodes* de mayor peso que los originales.

3.2.12. YGUARD

3.2.12.1. Introducción

yGuard es un ofuscador de *bytecode*. Ofrece una solución frente a la ingeniería inversa mediante el renombramiento de identificadores de paquetes, clases, métodos y campos; de tal manera que hace el código descompilado resultante completamente ilegible.

Además, como efecto añadido del proceso de ofuscación, yGuard proporciona un fichero de salida "*.jar*" de menor peso, es decir, consigue reducir su tamaño. Esto repercute en un tiempo de carga y en un tiempo de ejecución menores para las aplicaciones desarrolladas.

yGuard está basado en una versión del ofuscador Retroguard, que es distribuido bajo licencia LGPL , para la que se ha mejorado el código existente y se ha construido una librería reutilizable. Por encima de esto cuenta con integración bajo Ant, esto permite a desarrolladores integrar el proceso de ofuscación dentro del proceso de construcción de una aplicación.

Las ventajas de la utilización de yGuard frente a otras aplicaciones son:

- Es una aplicación gratuita.

- Integra Ant. Las herramientas existentes usan mecanismos propietarios para invocar y configurar la tarea de ofuscación. yGuard puede ser configurado completamente utilizando sintaxis XML en cualquier tarea Ant.

- Solventa muchos de los problemas que los usuarios experimentan al utilizar la librería original de Retroguard.

- Ofusca correctamente programas que con dependencias de librerías externas.

- Puede automáticamente renombrar y optimizar el código fuente de acuerdo a las opciones de ofuscación especificadas.

3.2.12.2. Ofuscación de Ejemplo 1

Los pasos que hemos seguido para la ofuscación de nuestro primer ejemplo y la posterior evaluación de la métrica has sido los siguientes:

1) Generar el *script* necesario para correr la tarea de ofuscación en Ant. Ver apartado de utilización.

2) A continuación hemos compilado el archivo "*build.xml*" mediante Ant.

3) Comprobamos que no han ocurrido ninguna incidencia en el archivo *logfile* que hemos especificado que se genere como opción. Ver programa 3.32.

Programa 3.54. Archivo de incidencias, obfuscationlog.xml.

```
<?xml version="1.0" encoding="UTF-8"?>
<yguard version="1.1">
<!--
  yGuard Bytecode Obfuscator, v1.3.2, a Product of yWorks GmbH -
http://www.yworks.com

  Logfile created on Thu Apr 14 13:33:42 CEST 2005

  Jar file to be obfuscated:           Test1.jar
  Target Jar file for obfuscated code: Test1res.jar
-->
<!--
  Memory in use after class data structure built: 1228216 bytes
  Total memory available                        : 2637824 bytes
-->
<expose>
  <method class="Test1" name="void main(java.lang.String[])"/>
</expose>
<map>
  <class name="Test1" map="A"/>
  <field class="Test1" name="aux" map="A"/>
  <field class="Test1" name="vector" map="B"/>
  <field class="Test1" name="hello" map="C"/>
  <method class="Test1" name="java.lang.String pruebaOfus()" map="A"/>
  <method class="Test1" name="int getVector()" map="B"/>
  <class name="Hola" map="B"/>
  <method class="Hola" name="java.lang.String getHola(java.lang.String)"
map="A"/>
</map>
</yguard>
```

Por salida estandar se muestra un mensaje de que el proceso de construcción se ha llevado a cabo con éxito. Además en este *logfile* se muestra el proceso de renombrado llevado a cabo.

4) Seguidamente comprobamos que el código ofuscado sigue realizando la misma función, para ello en primer lugar tenemos que extraer los bytecodes del contendor de salida "*Testres1.jar*".

```
D:\Temp\yguard-1.3.2\lib\jar xf Test1res.jar
```

Obtenemos los archivos "*.class*" ofuscados: "*A.class*" (925 *bytes*) y "*B.class*" (200 *bytes*). Comprobamos que se han reducido con respecto a los originales.

```
D:\Temp\yguard-1.3.2\lib>java A
HOLA MUNDO
PRUEBA DE OFUSCACION
```

Ejemplo=1

Comprobamos que efectivamente sigue realizándose la misma función.

5) Por último evaluamos la métrica mediante la comparación de los *bytecodes* que hemos obtenido como resultado del proceso de ofuscación con yGuard y los originales. Así podemos crear la tabla 3.15.

MÉTRICA		EJEMPLO 1 (sin)		EJEMPLO 1(ofuscado)		PUNTUACIÓN
		Test1class	*Hola.class*	*A.class*	*B.class*	
Public	**Clase**					1
	Métodos	main	GetHola, Hola	main, A	A, B	
	Campos	hello		C		
Protected	**Métodos**	pruebaOfus		A		1
	Campos	aux		A		
Private	**Métodos**	getVector		B		1
	Campos	vector		B		
Borrado Numero Linea		Numero de Linea		Borrado		1
Encriptado de String		PRUEBA DE OFUSCACION		PRUEBA DE OFUSCACION		0
Desestructuracion de Datos		vector[]		B[]		0
Puntuación para Ofuscación de Layout y Datos						4

Tabla 3.15. Evaluación de ofuscación de Estructura y Datos para el ejemplo 1.

3.2.12.3. Ofuscación de Ejemplo 2

yGuard no proporciona opción de ofuscación de Control, por tanto la ofuscación del ejemplo 2 no es necesaria. Puntuamos con un 0 la métrica individual correspondiente.

3.2.12.4. Conclusión

yGuard puede realizar ofuscación de Estructura, renombrando identificadores de clases y miembros de clases *Public*, *Protected* y *Private*. Además borra toda información redundante, como comentarios, miembros si uso, etc.

- No es capaz de realizar ofuscación de Datos como puede verse en la tabla 3.15.

- Después de ser ofuscado con las opciones más severas, nuestro código seguía realizando la misma función.

- No proporciona mecanismos para ofuscación de Control de Flujo.

- Se proporciona un manual de usuario muy detallado en la página web asociada a la aplicación.

- Dificil de utilizar si no se conoce XML y la herramienta de compilación Ant. Aunque el manual explica como proceder desde cero.

3.3. PROGRAMA LÓGICO

En este apartado tratamos de comprobar la eficacia de los programas analizados con anterioridad, pero en este caso al ofuscar una aplicación de mayor peso, es decir, de un mayor tamaño. Hasta ahora, nuestro análisis se basaba en la ofuscación de pequeños ejemplos diseñados específicamente para comprobar la eficacia de los ofuscadores en cuanto a las principales características que una aplicación de este tipo debe ofrecer. Con esta nueva prueba añadimos mayor realismo a los resultados obtenidos hasta el momento y por tanto a la comparativa final.

Para los 11 ofuscadores, que hemos comprobado que funcionan correctamente de la colección inicial, vamos a proceder como refleja el siguiente esquema:

Esquema 3.2. Proceso de evaluación de programa lógico.

```
Para cada ofuscador o {
      Intentar ofuscar programa lógico con las opciones de seguridad mas
      restricctivas a.
      Verificar que el programa realiza la misma función.
      Si fallo, intentar con opciones menos restricctivas
            Repetir (o, a)
      Si fallo persiste, devolver error(o, a)
      Devolver (o, a)
```

Como programa lógico hemos seleccionado una aplicación que se ajuste a nuestras necesidades, es decir, una aplicación de cierto peso sin ser excesivamente compleja pues debemos ser capaces de determinar puntos de entrada para la ofuscación y resolver todas las dependencias para evitar conflictos debido al proceso de renombrado.

No es objeto de este proyecto el escribir un ejemplo de tal envergadura, así pues hemos recurrido a Internet para encontrar tal aplicación. En concreto se trata de un programa diseñado para la exploración del conjunto de Mandelbrot. El denominado "Mandelbrot Set" es un fractal. Los fractales son objetos que presentan ciertas similitudes a diferentes escalas. Tampoco es objeto de este proyecto el análisis de la aplicación en concreto, por tanto, no se entrará en detalle de las características de la misma así como de su funcionalidad.

Para nuestro análisis solo necesitamos comprobar si después de ofuscar el código de la aplicación, ésta sigue funcionando.

A la hora de ofuscar una aplicación más compleja los ofuscadores han de tener en cuenta ciertos aspectos para evitar posibles conflictos. Algunas de éstas características son :

1) La aplicación puede contar con varios métodos "*main*" que habrá que preservar del proceso de ofuscación.

2) Preservar métodos nativos.

3) Preservar métodos y campos de clases que implementan la interfaz "*Serializable*". Clases que requieran un especial manejo a la hora del proceso de serialización deben implementar métodos especiales con las siguientes asignaturas:

```
private void writeObject(java.io.ObjectOutputStream out)
private void readObject(java.io.ObjectInputStream in)
```

4) Durante la ejecución del proceso de serialización se asocia a cada clase serializada un número de versión, el cual es usado durante la deserialización para verificar que el transmisor y el receptor de un objeto serializado ha cargado clases para ese objeto que son compatibles respecto a la serialización. Este número de versión ha de ser preservado.

5) Manejo de *RMI*, invocación de métodos remotos.

6) Manejo de llamadas de la clase "*java.lang.Class*", como por ejemplo "*Class.ForName()*". Este tipo de llamadas reciben el nombre de "*Reflection API calls*"

Algunas características sobre nuestro programa lógico son:

- jmanex.jar tiene un tamaño de 149 *Kbytes*.
- El código fuente cuenta con 45 archivos ".java".
- Programa con más de 3400 líneas de código fuente.
- Ejecutable desde cualquier plataforma que utilice Java 1.2.2.
- Es una aplicación que cuenta con varios métodos "*main*".
- Es necesario tener en cuenta serialización.
- Cuenta con métodos nativos, aunque no parece influir en su ejecución.
- Maneja llamadas de la clase "*java.lang.Class*".

La forma de proceder ha sido la siguiente. Una vez descargada la aplicación, hemos comprobado su correcto funcionamiento. Para ello, hemos consultado el manual de usuario que se proporciona en la web asociada.

```
D:\Temp\Aplicaciones Java>java -jar jmanex.jar
Unable to load native library jmanexNative
Using Java computation instead of native.
```

Comprobamos que se ejecuta sin problema alguno. Al parecer no cuenta con ciertas librerías nativas pero se emplean por defecto las de Java. A continuación seguimos el esquema de análisis mostrado anteriormente (esquema 3.2).

3.3.1. PROGUARD

Para la ofuscación de nuestra aplicación hemos seguido los mismos pasos que los indicados a la hora de ofuscar los ejemplos del apartado anterior. En este caso ha sido necesaria la inclusión en el *script* de configuración de ciertas líneas para tener en cuenta la serialización. Solo se ha requerido el especificar como punto de entrada para el proceso de ofuscación el método *main* de mayor orden jerárquico, el resto se resuelven de forma transparente. Por otro lado Proguard también soluciona de forma automática las denominadas "*Reflection API calls*".

Programa 3.55. Script que contiene opciones de ofuscación para Proguard, Config.pro.

```
-injars             jmanex.jar
-outjars            jmanexRes.jar

-libraryjars    "C:\j2sdk1.4.2_07\jre\lib\rt.jar"

-printmapping   program.map

-overloadaggressively

-keep  public class JManEx{
       public static void main(java.lang.String[]);
       }

-keepnames class * implements java.io.Serializable

-keepclassmembers class * implements java.io.Serializable {
    static final long serialVersionUID;
    !static !transient <fields>;
    private void writeObject(java.io.ObjectOutputStream);
    private void readObject(java.io.ObjectInputStream);
    java.lang.Object writeReplace();
    java.lang.Object readResolve();
}
```

El resultado del proceso de ofuscación es un contenedor "*jmanexRes.jar*" (94 *KB*) cuyo tamaño inicial se ha reducido considerablemente.

Destacar que el proceso de renombrado llevado a cabo se recoge en un fichero de mapeo especificado en el *script* como opción, "*program.map*".

Por pantalla hemos obtenido el resultado del proceso llevado a cabo, ya que Proguard no incorpora opción de generar un *logfile*. Ver programa 3.56.

Programa 3.56. Resultado del proceso de ofuscación.

```
ProGuard, version 3.2
Reading jars...
Reading program jar [jmanex.jar]
Reading library jar [C:\Archivos de programa\Java\j2re1.4.2_07\lib\rt.jar]
Removing unused library classes...
    Original number of library classes: 5672
    Final number of library classes:    150
Shrinking...
Removing unused program classes and class elements...
    Original number of program classes: 54
    Final number of program classes:    47
Optimizing...
```

```
Shrinking...
Removing unused program classes and class elements...
    Original number of program classes: 47
    Final number of program classes:    47
Obfuscating...
Renaming program classes and class elements...
Printing mapping to [program.map]
Writing jars...
Preparing output jar [jmanexRes.jar]
Copying resources from program jar [jmanex.jar]
```

Una vez comprobado que no ha ocurrido error alguno, ejecutamos la aplicación ofuscada. El resultado es satisfactorio. Proguard consigue ofuscar nuestro programa lógico sin problema, puntuaremos en consecuencia la métrica correspondiente.

3.3.2. RETROGUARD

En este caso mediante el *Wizard* hemos generado un *script* de configuración para la ofuscación, en el que: se han preservado todos los métodos *main* presentes en la aplicación. Además ha sido necesaria una opción para tener en cuenta la serialización. Por ultimo también se ha tenido en cuenta la presencia de "*Reflection API calls*", en concreto ha sido necesario el preservar "*toString()*", presente en una de las clases de la aplicación. Todo esto se ve reflejado en el siguiente *script*, ver programa 3.57.

Programa 3.57. Script de configuración para Retroguard.

```
# Automatically generated script for RetroGuard bytecode obfuscator.
# To be used with Java JAR-file: D:\Temp\prueba\jmanex.jar
# 05-may-2005 20:16:20
#
.class util/ProgressiveMemoryImageProducer
.class JuliaExplorerPanel
.class GradPresetsEditor
.class JuliaPreset
.class MandelPreset
.class JManEx
.class MandelComputer
.method util/ProgressiveMemoryImageProducer/main ([Ljava/lang/String;)V
.method grad/BasicGradientPainter/toString ()Ljava/lang/String;
.method MandelPreset/main ([Ljava/lang/String;)V
.method MandelComputer/main ([Ljava/lang/String;)V
.method GradPresetsEditor/main ([Ljava/lang/String;)V
.method JManEx/main ([Ljava/lang/String;)V
.method JuliaPreset/main ([Ljava/lang/String;)V
.method JuliaExplorerPanel/main ([Ljava/lang/String;)V
.field grad/BasicGradientPainter/serialVersionUID J
.option Serializable
```

En este caso hemos especificado, como opción desde línea de comandos, la generación de un *logfile* en el que se refleje el proceso de renombrado llevado a cabo, así como posibles incidencias en la ofuscación.

Tras comprobar que no ha ocurrido error alguno, ejecutamos la aplicación ofuscada para comprobar su correcto funcionamiento. Después de algunas simulaciones conseguimos sin problemas que funcione bien. Por lo tanto puntuamos con un "1" la métrica correspondiente.

3.3.3. JAVAGUARD

Como estudiamos en el apartado 3.2.3, Javaguard era una de las aplicaciones analizadas con menos opciones a la hora de configurar el proceso de ofuscación. Solo ofrece como opción el preservar los nombres de clases y miembros de las mismas. Tanto en la página web asociada como en el breve manual de usuario proporcionado por la aplicación, no se especifican opciones para tener en cuenta serialización. Tampoco como tener en cuenta las "*Reflection API calls*". La aplicación en cuestión parece no ser compatible con estos aspectos.

Para la ofuscación de nuestro programa lógico comenzamos preservando del proceso de ofuscación todos los métodos *main* presentes en la aplicación. De esta forma conseguimos que la aplicación ofuscada se ejecutara pero no del todo bien, pues no se cargaban ciertas partes del programa (*presets*). Por pantalla se mostraba el siguiente resultado de ejecución:

Programa 3.58. Resultado de ejecución jmanexRes.jar.

```
Unable to load gradient presets data.
Exception: java.lang.ClassNotFoundException: grad.PresetGradientPainter
java.lang.ClassNotFoundException: grad.PresetGradientPainter
        at java.net.URLClassLoader$1.run(Unknown Source)
        at java.security.AccessController.doPrivileged(Native Method)
        at java.net.URLClassLoader.findClass(Unknown Source)
        at java.lang.ClassLoader.loadClass(Unknown Source)
        at sun.misc.Launcher$AppClassLoader.loadClass(Unknown Source)
        at java.lang.ClassLoader.loadClass(Unknown Source)
        at java.lang.ClassLoader.loadClassInternal(Unknown Source)
        at java.lang.Class.forName0(Native Method)
        at java.lang.Class.forName(Unknown Source)
        at java.io.ObjectInputStream.resolveClass(Unknown Source)
        at java.io.ObjectInputStream.readNonProxyDesc(Unknown Source)
        at java.io.ObjectInputStream.readClassDesc(Unknown Source)
        at java.io.ObjectInputStream.readOrdinaryObject(Unknown Source)
        at java.io.ObjectInputStream.readObject0(Unknown Source)
        at java.io.ObjectInputStream.readArray(Unknown Source)
        at java.io.ObjectInputStream.readObject0(Unknown Source)
        at java.io.ObjectInputStream.defaultReadFields(Unknown Source)
        at java.io.ObjectInputStream.readSerialData(Unknown Source)
        at java.io.ObjectInputStream.readOrdinaryObject(Unknown Source)
        at java.io.ObjectInputStream.readObject0(Unknown Source)
        at java.io.ObjectInputStream.readObject(Unknown Source)
        at g.<init>(Unknown Source)
        at g.<init>(Unknown Source)
        at c.<init>(Unknown Source)
        at v.<init>(Unknown Source)
        at u.a(Unknown Source)
        at u.<init>(Unknown Source)
        at k.main(Unknown Source)
Unable to load native library jmanexNative
Using Java computation instead of native.
```

Comprobamos que se produce un error al no encontrar una clase denominada: "*grad.PresetGradientPainter*". Al intentar preservar esta clase, así como sus miembros del proceso de ofuscación, la aplicación dejaba de funcionar. Además, aparecen también errores debido a la no exclusión del proceso de ofuscado de métodos y miembros de clases que implementen la interfaz *Serializable*. Javaguard, como se ha comentado anteriormente, no proporciona opciones para solventar este problema.

Por tanto, concluimos que Javaguard no es capaz de ofuscar el programa lógico y que éste siga funcionando como originalmente lo hacía la aplicación sin ofuscar. Se asignará un "0", en la métrica correspondiente.

3.3.4. JSHRINK

Jshrink es una de las aplicaciones más completas en cuanto a opciones que pueden tenerse en cuenta para el proceso de ofuscación. Consultando su completo manual de usuario comprobamos que es compatible con: serialización, *Reflection API calls*, métodos nativos, etc. Por tanto a priori no debemos tener problemas a la hora de ofuscar el programa lógico.

Jshrink proporciona la posibilidad de crear el *script* de configuración de forma manual o mediante una sencilla interfaz gráfica. Hemos optado por la primera opción. De acuerdo al manual de usuario, el *script* de configuración para tener en cuenta todos los aspectos mencionados anteriormente será de la forma:

Programa 3.59. Script de configuración para proceso de ofuscación por Jshrink.

```
-classpath C:\j2re1.4.2_07\jre\lib\rt.jar
-classStringMatch
-noFinalize
-serialInherit
-stringEncrypt
D:\Temp\prueba\jmanex.jar
-keep grad.BasicGradientPainter
-keep JManEx
-keep grad.PresetGradientPainter
-keep grad.GradientSegment
```

Destacar que tras varias simulaciones erróneas en las que no se encontraban ciertas clases y después de preservarlas del proceso de ofuscación, la posterior ejecución de la aplicación ofuscada fue un éxito.

El resultado de la ofuscación es un contenedor de salida "*jmanexRes.jar*" (103 *KB*) de menor tamaño que el archivo original, se ha conseguido una reducción del 31'9%. Por tanto concluimos que Jshrink es capaz de ofuscar el programa lógico.

3.3.5. CAFEBABE

CafeBabe es un completo desensamblador de código Java. Ofrece opciones de compactación y ofuscación del código pero no es una aplicación diseñada con tal finalidad. En cuanto al proceso de ofuscación, las opciones ofrecidas se limitan a ofuscación de Estructura (renombrado), pudiendo preservar identificadores de clases y miembros de clases.

En primer lugar hemos llevado a cabo el proceso de compactación del programa lógico. El resultado ha sido un contendor de salida de menor peso que el original "*jmanexStrip.jar*" (124 *KB*). La aplicación compactada aún seguía funcionado correctamente.

Como hemos mencionado anteriormente, CafeBabe no es una aplicación diseñada para la ofuscación. No es compatible con serialización, métodos nativos o *Reflection API calls*. Después de ofuscar preservando únicamente la clase principal de mayor jerarquía (es necesario seleccionar punto de entrada para el proceso de ofuscación) obtuvimos que al ejecutar nuestro programa se producían los siguientes errores, ver programa 3.60:

Programa 3.60. Resultado de ejecución programa lógico ofuscado.

```
Unable to load gradient presets data.
Exception: java.lang.ClassNotFoundException: grad.PresetGradientPainter
java.lang.ClassNotFoundException: grad.PresetGradientPainter
        at java.net.URLClassLoader$1.run(Unknown Source)
        at java.security.AccessController.doPrivileged(Native Method)
        at java.net.URLClassLoader.findClass(Unknown Source)
        at java.lang.ClassLoader.loadClass(Unknown Source)
        at sun.misc.Launcher$AppClassLoader.loadClass(Unknown Source)
        at java.lang.ClassLoader.loadClass(Unknown Source)
        at java.lang.ClassLoader.loadClassInternal(Unknown Source)
        at java.lang.Class.forName0(Native Method)
        at java.lang.Class.forName(Unknown Source)
        at java.io.ObjectInputStream.resolveClass(Unknown Source)
        at java.io.ObjectInputStream.readNonProxyDesc(Unknown Source)
        at java.io.ObjectInputStream.readOrdinaryObject(Unknown Source)
        at java.io.ObjectInputStream.readObject0(Unknown Source)
        at java.io.ObjectInputStream.readArray(Unknown Source)
        at java.io.ObjectInputStream.readObject0(Unknown Source)
        at java.io.ObjectInputStream.defaultReadFields(Unknown Source)
        at java.io.ObjectInputStream.readSerialData(Unknown Source)
        at java.io.ObjectInputStream.readObject0(Unknown Source)
        at java.io.ObjectInputStream.readObject(Unknown Source)
        at G.<init>(Unknown Source)
        at G.<init>(Unknown Source)
        at C.<init>(Unknown Source)
        at Y.<init>(Unknown Source)
        at W.init(Unknown Source)
        at W.<init>(Unknown Source)
        at JManEx.main(Unknown Source)
Exception in thread "main" java.lang.NoSuchMethodError:
As.apk(Ljava/util/Observ
er;)V
        at Ak.apk(Unknown Source)
        at C.<init>(Unknown Source)
        at Y.<init>(Unknown Source)
        at W.init(Unknown Source)
        at W.<init>(Unknown Source)
        at JManEx.main(Unknown Source)
```

Se produce un error porque no se encuentra la clase: "*grad.PresetGradientPainter*". Tras preservar esta clase obtuvimos lo siguiente como resultado de la ejecución del programa lógico ofuscado:

Programa 3.61. Resultado de ejecución programa lógico ofuscado.

```
Unable to load gradient presets data.
Exception: java.io.InvalidClassException: grad.PresetGradientPainter; local
class incompatible: stream classdesc serialVersionUID = 5786935299744842060,
local class serialVersionUID = -9117130052050238138
java.io.InvalidClassException: grad.PresetGradientPainter; local class
incompatible: stream classdesc serialVersionUID = 5786935299744842060, local
class serialVersionUID = -9117130052050238138
        at java.io.ObjectStreamClass.initNonProxy(Unknown Source)
        at java.io.ObjectInputStream.readNonProxyDesc(Unknown Source)
        at java.io.ObjectInputStream.readClassDesc(Unknown Source)
        at java.io.ObjectInputStream.readOrdinaryObject(Unknown Source)
        at java.io.ObjectInputStream.readObject0(Unknown Source)
        at java.io.ObjectInputStream.readArray(Unknown Source)
        at java.io.ObjectInputStream.readObject0(Unknown Source)
        at java.io.ObjectInputStream.defaultReadFields(Unknown Source)
        at java.io.ObjectInputStream.readSerialData(Unknown Source)
        at java.io.ObjectInputStream.readOrdinaryObject(Unknown Source)
        at java.io.ObjectInputStream.readObject0(Unknown Source)
        at java.io.ObjectInputStream.readObject(Unknown Source)
        at H.<init>(Unknown Source)
        at H.<init>(Unknown Source)
        at C.<init>(Unknown Source)
        at Ab.<init>(Unknown Source)
        at Z.init(Unknown Source)
        at Z.<init>(Unknown Source)
        at JManEx.main(Unknown Source)
Exception in thread "main"
java.lang.NoSuchMethodError:util.Av.apk(Ljava/util/O
bserver;)V
        at util.An.apk(Unknown Source)
        at C.<init>(Unknown Source)
        at Ab.<init>(Unknown Source)
        at Z.init(Unknown Source)
        at Z.<init>(Unknown Source)
        at JManEx.main(Unknown Source)
```

Errores asociados a la serialización de métodos y campos. No se ha conseguido solventar estos problemas porque CafeBabe no ofrece opción alguna para tener en cuenta aspectos relacionados con la serialización.

3.3.6. ZELIX KLASSMASTER

Zelix KlassMaster también es un ofuscador bastante completo según lo analizado en el apartado 3.2.6. Por defecto ofrece manejo automático de: *RMI*, *JavaBeans* y *EJB's*. Además se preservan por defecto campos asociados a serialización como por ejemplo "*serialVersionUID*". En principio no deberíamos encontrar problemas para la ofuscación del programa lógico, pues se tienen en cuenta todas las posibles fuentes de conflicto a la hora de la ofuscación para una aplicación de envergadura como la que analizamos.

Hemos comenzado por compactar el código original. No debemos olvidar que Zelix KlassMaster ofrece esta opción (*Trim*) por separado del proceso de ofuscación. Al compactar el programa lógico original obtenemos un contenedor de salida "*jmanexTrim.jar*" (118 *KB*) de menor tamaño que el contenedor original. Comprobamos que la el código compactado sigue realizando la misma función, es decir, la aplicación sigue funcionando.

A continuación hemos procedido a la ofuscación del anterior contenedor (código compacto). Al cargar la aplicación mediante la interfaz gráfica se nos avisa de que existen métodos que pueden que no se manejen de forma automática. En concreto métodos relacionados con las *Reflection API calls*, y que debemos preservar del proceso de ofuscación por tanto. Tras preservar los métodos especificados procedemos a la ofuscación de la aplicación. Al ejecutar el contenedor resultante nos encontramos con el siguiente error por pantalla:

Programa 3.62. Resultado ejecución aplicación ofuscada con Zelix KlassMaster.

```
Unable to load gradient presets data.
Exception: java.lang.ClassNotFoundException: grad.PresetGradientPainter
java.lang.ClassNotFoundException: grad.PresetGradientPainter
        at java.net.URLClassLoader$1.run(Unknown Source)
        at java.security.AccessController.doPrivileged(Native Method)
        at java.net.URLClassLoader.findClass(Unknown Source)
        at java.lang.ClassLoader.loadClass(Unknown Source)
        at sun.misc.Launcher$AppClassLoader.loadClass(Unknown Source)
        at java.lang.ClassLoader.loadClass(Unknown Source)
        at java.lang.ClassLoader.loadClassInternal(Unknown Source)
        at java.lang.Class.forName0(Native Method)
        at java.lang.Class.forName(Unknown Source)
        at java.io.ObjectInputStream.resolveClass(Unknown Source)
        at java.io.ObjectInputStream.readNonProxyDesc(Unknown Source)
        at java.io.ObjectInputStream.readClassDesc(Unknown Source)
        at java.io.ObjectInputStream.readOrdinaryObject(Unknown Source)
        at java.io.ObjectInputStream.readObject0(Unknown Source)
        at java.io.ObjectInputStream.readArray(Unknown Source)
        at java.io.ObjectInputStream.readObject0(Unknown Source)
        at java.io.ObjectInputStream.defaultReadFields(Unknown Source)
        at java.io.ObjectInputStream.readSerialData(Unknown Source)
        at java.io.ObjectInputStream.readOrdinaryObject(Unknown Source)
        at java.io.ObjectInputStream.readObject0(Unknown Source)
        at java.io.ObjectInputStream.readObject(Unknown Source)
        at m.<init>(Unknown Source)
        at m.<init>(Unknown Source)
        at l.<init>(Unknown Source)
        at t.<init>(Unknown Source)
        at k.a(Unknown Source)
        at k.<init>(Unknown Source)
        at JManEx.main(Unknown Source)
Unable to load native library jmanexNative
Using Java computation instead of native.
```

Comprobamos que los errores están relacionados con: la búsqueda de una clase que parece no estar presente "*grad.PresetGradientPainter*"; y con la serialización de ciertos métodos.

Después de varias simulaciones sin éxito intentando preservar diferentes métodos y campos de clases que implementan la interfaz "*Serializable*", optamos por preservar de manera genérica cualquier método de cualquier clase de cualquier paquete que implemente un objeto perteneciente al paquete "*java.io.Serializable*".

Al ofuscar teniendo en cuenta lo anterior y despues de comprobar que no ocurrió ningún error durante el proceso, ejecutamos la aplicación resultante. Comprobamos que funciona perfectamente. El contenedor resultante "*jmanexObf.jar*" (108 KB) se ha reducido aún más como consecuencia de la ofuscación.

Concluimos por tanto que Zelix KlassMaster es capaz de oscurecer con éxito el código Java de una aplicación. Puntuamos con un "1" la métrica correspondiente.

3.3.7. SMOKESCREEN

Smokescreen es otra de las aplicaciones más completas de las estudiadas en la sección anterior (3.2), proporciona mecanismos para ofuscación de Estructura, Datos y Control. Sin embrago, en el manual de usuario no se especifican opciones para manejar serialización o *Reflection API calls* . Las únicas opciones están destinadas a preservar identificadores de clases y miembros de clases, como en la mayoría de los ofuscadores.

Después de varias simulaciones intentando preservar los identificadores de clases y miembros de clases conflicitivos no hemos conseguido la ofuscación de nuestro programa lógico. En todas las ocasiones obtuvimos por pantalla la siguiente salida:

Programa 3.63. Resultado ejecución de aplicación ofuscada.

```
Unable to load gradient presets data.
Exception: java.lang.ClassNotFoundException: grad.PresetGradientPainter
java.lang.ClassNotFoundException: grad.PresetGradientPainter
        at java.net.URLClassLoader$1.run(Unknown Source)
        at java.security.AccessController.doPrivileged(Native Method)
        at java.net.URLClassLoader.findClass(Unknown Source)
        at java.lang.ClassLoader.loadClass(Unknown Source)
        at sun.misc.Launcher$AppClassLoader.loadClass(Unknown Source)
        at java.lang.ClassLoader.loadClass(Unknown Source)
        at java.lang.ClassLoader.loadClassInternal(Unknown Source)
        at java.lang.Class.forName0(Native Method)
        at java.lang.Class.forName(Unknown Source)
        at java.io.ObjectInputStream.resolveClass(Unknown Source)
        at java.io.ObjectInputStream.readNonProxyDesc(Unknown Source)
        at java.io.ObjectInputStream.readClassDesc(Unknown Source)
        at java.io.ObjectInputStream.readOrdinaryObject(Unknown Source)
        at java.io.ObjectInputStream.readObject0(Unknown Source)
        at java.io.ObjectInputStream.readArray(Unknown Source)
        at java.io.ObjectInputStream.readObject0(Unknown Source)
        at java.io.ObjectInputStream.defaultReadFields(Unknown Source)
        at java.io.ObjectInputStream.readSerialData(Unknown Source)
        at java.io.ObjectInputStream.readOrdinaryObject(Unknown Source)
        at java.io.ObjectInputStream.readObject0(Unknown Source)
        at java.io.ObjectInputStream.readObject(Unknown Source)
        at F.<init>(Unknown Source)
        at F.<init>(Unknown Source)
        at C.<init>(Unknown Source)
        at U.<init>(Unknown Source)
        at T.A(Unknown Source)
```

```
        at T.<init>(Unknown Source)
        at JManEx.main(Unknown Source)
Exception in thread "main" java.lang.VerifyError: (class: grad/A, method: A
sign
ature: (D)I) Register 3 contains wrong type
        at D.B(Unknown Source)
        at D.<init>(Unknown Source)
        at C.<init>(Unknown Source)
        at U.<init>(Unknown Source)
        at T.A(Unknown Source)
        at T.<init>(Unknown Source)
        at JManEx.main(Unknown Source)
```

Concluimos que Smokescreen no es capaz de ofuscar programa lógico. Asignaremos por tanto un "0" a la métrica individual correspondiente.

3.3.8. JOGA

Como hemos estudiado anteriormente JoGa más que un ofuscador es un optimizador, aunque también proporciona opciones para la ofuscación. Estas opciones no contemplan serialización o *Reflection API calls*, que son las más conflictivas. Sin embargo JoGa es capaz de preservar del proceso de ofuscación las clases indicadas en su totalidad, es decir, preserva el nombre de la clase así como todos sus métodos y campos.

Después de haber ofuscado con anteriores aplicaciones hemos comprobado que los consabidos errores se producen por clases y miembros de clases de los paquetes "grad" y "util" presentes en el código de nuestro programa. Así, preservando estos paquetes en su totalidad hemos sido capaces de ofuscar la aplicación consiguiendo que el contenedor resultante al ser ejecutado siga realizando la misma función. Hemos perdido efectividad a la hora oscurecer el código porque hemos dejado parte sin ofuscar pero hemos conseguido que la aplicación siga funcionando.

El proceso llevado a cabo es análogo al seguido para la ofuscación de ejemplos estudiados en el apartado anterior 3.2.8. Hemos cargado la aplicación, indicando a JoGa que se trataba de una *Application*. Por su parte JoGa detecta de forma automática clases que contienen métodos *main* y las preserva de la ofuscación. Especificando las opciones deseadas para el proceso de optimización y ofuscación obtenemos un contenedor de salida: "*jmanex_JOGA.jar*" (102 *KB*). Observamos que se ha reducido su tamaño comparándolo con el archivo original.

Destacar que hemos especificado como opción la generación de dos archivos: uno en el que se indican los resultados del proceso de ofuscación y otro en el que se puede observar el proceso de renombrado llevado a cabo.

3.3.9. JOBFUSCATE

Jobfuscate es un ofuscador de archivos "*.class*" como estudiamos en el apartado 3.2.10. Desde el manual de usuario proporcionado en la página web asociada se asegura

que es capaz de manejar de forma automática serialización, pero que presenta problemas si nuestra aplicación cuenta con *Reflection API calls*. El programa lógico cuenta con este tipo de llamadas por tanto se aconseja que toda clase que las contenga sea excluida del proceso de ofuscación, con objeto de que no se produzcan errores.

Después de varias simulaciones hemos sido incapaces de conseguir que se generara el contenedor de salida. Siempre se producen errores por este tipo de llamadas. En el *logfile* que se genera se advierte de estos errores pero no donde se producen concretamente, para poder preservar todas las clases donde hay llamadas de este tipo. Al no haber escrito el código fuente de la aplicación pues resulta bastante difícil determinar en cual de las 45 clases existentes hay llamadas a "*Class.ForName()*".

Debido a la no inclusión de una opción para tener en cuenta las "*Reflection API calls*" asignamos a la métrica correspondiente un "0". Jobfuscate no es capaz de ofuscar nuestro programa lógico.

3.3.10. MARVINOBFUSCATOR

MarvinObfuscator es una aplicación de ofuscación muy elemental, permite ofuscación de Estructura y de Datos, pero no proporciona opciones de configuración para el proceso de ofuscación concernientes a serialización o *Reflection API calls*. Por tanto cabe esperar que la ofuscación de nuestro programa lógico no se lleve a cabo correctamente.

Efectivamente, después de varias simulaciones intentando preservar clases conflictivas no hemos conseguido que la aplicación ofuscada funcionara. El proceso de ofuscación se llevaba a cabo sin errores pero al intentar ejecutar el resultado obteníamos una y otra vez la siguiente salida por pantalla. Ver programas 3.63 y 3.64.

Programa 3.64. Resultado del proceso de ofuscación.

```
The Marvin Obfuscator 1.2b, (c) 2000-2001 by Dr. Java (www.drjava.de)
Pass 1
Pass 2
56 entries written to jar file, total size=128187, processing time: 1182 ms
Saved jmanexRes.jar (128187 bytes)
```

Programa 3.65. Resultado ejecución aplicación ofuscada.

```
Exception in thread "main" java.lang.IncompatibleClassChangeError
        at volatile.<clinit>()
        at transient.<init>()
        at throw.b()
        at throw.<init>()
        at JManEx.main()
```

Por tanto MarvinObfuscator no es capaz de ofuscar con éxito el programa lógico. Puntuamos con un "0" la métrica correspondiente.

3.3.11. YGUARD

Como estudiamos en el apartado 3.2.12 yGuard es un ofuscador integrado para su utilización con Ant. Analizando el manual de usuario comprobamos que no se proporcionan opciones explicitas para el manejo de serialización y *Reflection API calls* que es lo que nos interesa resolver a nosotros. Todas las opciones van destinadas a preservar los identificadores de clases, métodos y campos.

Para el análisis de la aplicación hemos comenzado por preservar solamente los métodos *main* de la misma. Al ejecutar encontrábamos los siguientes errores, que se reflejan en el programa 3.65.

Programa 3.66. Resultado ejecución aplicación ofuscada.

```
Unable to load gradient presets data.
Exception: java.lang.ClassNotFoundException: grad.PresetGradientPainter
java.lang.ClassNotFoundException: grad.PresetGradientPainter
        at java.net.URLClassLoader$1.run(Unknown Source)
        at java.security.AccessController.doPrivileged(Native Method)
        at java.net.URLClassLoader.findClass(Unknown Source)
        at java.lang.ClassLoader.loadClass(Unknown Source)
        at sun.misc.Launcher$AppClassLoader.loadClass(Unknown Source)
        at java.lang.ClassLoader.loadClass(Unknown Source)
        at java.lang.ClassLoader.loadClassInternal(Unknown Source)
        at java.lang.Class.forName0(Native Method)
        at java.lang.Class.forName(Unknown Source)
        at java.io.ObjectInputStream.resolveClass(Unknown Source)
        at java.io.ObjectInputStream.readNonProxyDesc(Unknown Source)
        at java.io.ObjectInputStream.readClassDesc(Unknown Source)
        at java.io.ObjectInputStream.readOrdinaryObject(Unknown Source)
        at java.io.ObjectInputStream.readObject0(Unknown Source)
        at java.io.ObjectInputStream.readArray(Unknown Source)
        at java.io.ObjectInputStream.readObject0(Unknown Source)
        at java.io.ObjectInputStream.defaultReadFields(Unknown Source)
        at java.io.ObjectInputStream.readSerialData(Unknown Source)
        at java.io.ObjectInputStream.readOrdinaryObject(Unknown Source)
        at java.io.ObjectInputStream.readObject0(Unknown Source)
        at java.io.ObjectInputStream.readObject(Unknown Source)
        at L.<init>(Unknown Source)
        at L.<init>(Unknown Source)
        at Q.<init>(Unknown Source)
        at V.<init>(Unknown Source)
        at W.A(Unknown Source)
        at W.<init>(Unknown Source)
        at JManEx.main(Unknown Source)
Unable to load native library jmanexNative
Using Java computation instead of native.
```

Comprobamos que los errores son debidos por un lado a la no presencia de una clase denominada "*grad.PresetGradientPainter*" y por otro a objetos que implementan la interfaz *Serializable*.

A continuación hemos procedido a preservar la clase que falta, encontrándonos con que seguían produciéndose los siguientes errores:

Programa 3.67. Resultado ejecución aplicación ofuscada.

```
Exception: java.io.InvalidClassException: grad.PresetGradientPainter; local
class incompatible: stream classdesc serialVersionUID = 5786935299744842060,
local class serialVersionUID = 7715789927339721043
java.io.InvalidClassException: grad.PresetGradientPainter; local class
incompatible: stream classdesc serialVersionUID = 5786935299744842060, local
class serialVersionUID = 7715789927339721043
        at java.io.ObjectStreamClass.initNonProxy(Unknown Source)
        at java.io.ObjectInputStream.readNonProxyDesc(Unknown Source)
        at java.io.ObjectInputStream.readClassDesc(Unknown Source)
        at java.io.ObjectInputStream.readOrdinaryObject(Unknown Source)
        at java.io.ObjectInputStream.readObject0(Unknown Source)
        at java.io.ObjectInputStream.readArray(Unknown Source)
        at java.io.ObjectInputStream.readObject0(Unknown Source)
        at java.io.ObjectInputStream.defaultReadFields(Unknown Source)
        at java.io.ObjectInputStream.readSerialData(Unknown Source)
        at java.io.ObjectInputStream.readOrdinaryObject(Unknown Source)
        at java.io.ObjectInputStream.readObject0(Unknown Source)
        at java.io.ObjectInputStream.readObject(Unknown Source)
        at L.<init>(Unknown Source)
        at L.<init>(Unknown Source)
        at Q.<init>(Unknown Source)
        at V.<init>(Unknown Source)
        at W.A(Unknown Source)
        at W.<init>(Unknown Source)
        at JManEx.main(Unknown Source)
Unable to load native library jmanexNative
Using Java computation instead of native.
```

Encontramos que todos los errores son debidos a serialización. Después de realizar varias simulaciones, incluyendo comandos en el *script* para preservar objetos que implementen la interfaz *Serializable* no conseguimos que la aplicación ofuscada funcione correctamente. Esto es debido a que no existe ninguna opción que nos permita preservar cualquier objeto que implemente dicha interfaz, siendo imposible determinar todos los objetos del código que la implementan. Se consigue que la aplicación funcione pero no se cargan ciertos *presets* como ocurría al ofuscar con otras herramientas.

Por tanto concluimos que yGuard no es capaz de ofuscar el programa lógico. Puntuaremos con un “0” la métrica correspondiente.

3.4. DESCOMPILACIÓN

En este apartado comprobaremos si el código ofuscado puede ser descompilado nuevamente. Para este propósito nos apoyaremos en la aplicación JODE (Entorno de Descompilación y Optimización Java). Nos limitaremos a analizar si JODE es capaz de descompilar tantos los ejemplos diseñados para analizar la métrica como el programa lógico, en los casos para los que el proceso de ofuscación haya resultado exitoso.

3.4.1. DESCOMPILACIÓN DE EJEMPLOS

En el aparatado 3.2 (Procesado), se ha conseguido ofuscar con éxito los ejemplos mediante 11 de las 12 herramientas de ofuscación de las que partíamos inicialmente. Para cada una de estas aplicaciones vamos a analizar si el resultado del proceso de ofuscación, es decir, los ejemplos ofuscados pueden descompilarse nuevamente. Al descompilar probablemente no podamos descifrar el código oscurecido pero en esta sección tratamos de ir más allá. Asignaremos un punto en la métrica definitiva si el descompilador no es capaz de regenerar el código oscurecido. Esto será señal de que el proceso llevado a cabo es tan potente que no permite ni la visualización del código ofuscado.

En la siguiente tabla se resumen los resultados:

HERRAMIENTA	RESULTADO DE DESCOMPILACIÓN		
	EJEMPLO1		EJEMPLO 2
	Test1.class	*Hola.class*	*Test2.class*
Proguard	Si	Si	
Retroguard	Si	Si	
Javaguard	Si	Si	
Jshrink	Si	Si	
CafeBabe	Si	Si	
Zelix KlassMaster	No	Si	No
Smokescreen	Si	Si	Si
JoGa	Si	Si	
Jobfuscate	Si	Si	
Marvin Obfuscator	Si	Si	
yGuard	Si	Si	

Tabla 3.16. Resultados del proceso de descompilación mediante JODE.

Destacar que para cada ejemplo hemos especificado el nombre las clases originales para poder clasificar si la descompilación era posible. En realidad como consecuencia del proceso de ofuscación llevado a cabo por cada herramienta los nombres de las clases se han modificado en cada caso.

Aparecen huecos en blanco debido a que el ejemplo 2 no ha sido evaluado para la mayoría de las aplicaciones ya que no ofrecían ofuscación de Control.

Del análisis realizado se desprende que la única aplicación para la cual se consigue que la descompilación del código ofuscado no sea posible es Zelix KlassMaster. Por tanto asignaremos un "1" para la métrica correspondiente.

3.4.2. DESCOMPILACIÓN DE PROGRAMA LÓGICO

Procedemos de forma análoga pero en este caso para el programa lógico, en aquellos casos en los que se haya conseguido ofuscar con éxito. Ver tabla 3.17.

HERRAMIENTA	RESULTADO DESCOMPILACIÓN
	PROGRAMA LÓGICO
Proguard	Si
Retroguard	Si
Javaguard	
Jshrink	Si
CafeBabe	
Zelix KlassMaster	No
Smokescreen	
JoGa	Si
Jobfuscate	
Marvin Obfuscator	
yGuard	

Tabla 3.17. Resultados del proceso de descompilación mediante JODE.

En el apartado 3.3 (Programa Lógico) realizamos un análisis de las aplicaciones de ofuscación desde el punto de vista del oscurecimiento de un programa de cierto peso. Solamente 5 de las aplicaciones pudieron ofuscar con éxito el programa, de ahí que la mayoría de las herramientas no puedan ser evaluadas en este apartado.

Concluimos que la única aplicación capaz de oscurecer el código para no permitir ni la descompilación del *bytecode* ofuscado es Zelix KlassMaster. Por tanto puntuaremos con un “1” la métrica correspondiente.

3.5. EVALUACIÓN DE MÉTRICA

En este último apartado de la sección se van a presentar en forma de tablas, las evaluaciones de la métrica diseñada para la comprobación de la efectividad de cada una de las herramientas de ofuscación analizadas en el presente proyecto.

Una vez realizado todo el análisis experimental mediante simulaciones, estamos en disposición de establecer una comparativa final de los ofuscadores. Esta comparativa se basa en el trabajo desarrollado en apartados anteriores de esta sección. Para cada herramienta de trabajo se adjunta el resultado en forma de tabla.

3.5.1. PROGUARD

OBJETOS	CONTENIDO		PUNTUACIÓN
OFUSCACIÓN DE LAYOUT	Renombrar Identificadores	Public	1
		Protected	1
		Private	1
	Borrado de Número de Líneas		1
OFUSCACIÓN DE DATOS	Encriptado de String		0
	Desestructuracón de Datos		0
OFUSCACIÓN DE CONTROL	Transformaciones en Sentencias y Bucles		0
OFUSCACIÓN DE PROGRAMA			1
DECOMPILACIÓN DE CÓDIGO OFUSCADO			0
DECOMPILACIÓN DE PROGRAMA OFUSCADO			0
PUNTUACIÓN TOTAL			5

Tabla 3.18. Evaluación de métrica de ofuscación mediante Proguard.

3.5.2. RETROGUARD

OBJETOS	CONTENIDO		PUNTUACIÓN
OFUSCACIÓN DE LAYOUT	Renombrar Identificadores	Public	1
		Protected	1
		Private	1
	Borrado de Número de Líneas		1
OFUSCACIÓN DE DATOS	Encriptado de String		0
	Desestructuracón de Datos		0
OFUSCACIÓN DE CONTROL	Transformaciones en Sentencias y Bucles		0
OFUSCACIÓN DE PROGRAMA			1
DECOMPILACIÓN DE CÓDIGO OFUSCADO			0
DECOMPILACIÓN DE PROGRAMA OFUSCADO			0
PUNTUACIÓN TOTAL			5

Tabla 3.19. Evaluación de métrica de ofuscación mediante Retroguard.

3.5.3. JAVAGUARD

OBJETOS	CONTENIDO		PUNTUACIÓN
OFUSCACIÓN DE LAYOUT	Renombrar Identificadores	Public	1
		Protected	1
		Private	1
	Borrado de Número de Líneas		1
OFUSCACIÓN DE DATOS	Encriptado de String		0
	Desestructuracón de Datos		0
OFUSCACIÓN DE CONTROL	Transformaciones en Sentencias y Bucles		0
OFUSCACIÓN DE PROGRAMA			0
DECOMPILACIÓN DE CÓDIGO OFUSCADO			0
DECOMPILACIÓN DE PROGRAMA OFUSCADO			0
PUNTUACIÓN TOTAL			4

Tabla 3.20. Evaluación de métrica de ofuscación mediante Javaguard.

3.5.4. JSHRINK

OBJETOS	CONTENIDO		PUNTUACIÓN
OFUSCACIÓN DE LAYOUT	Renombrar Identificadores	Public	1
		Protected	1
		Private	1
	Borrado de Número de Líneas		1
OFUSCACIÓN DE DATOS	Encriptado de String		1
	Desestructuracón de Datos		0
OFUSCACIÓN DE CONTROL	Transformaciones en Sentencias y Bucles		0
OFUSCACIÓN DE PROGRAMA			1
DECOMPILACIÓN DE CÓDIGO OFUSCADO			0
DECOMPILACIÓN DE PROGRAMA OFUSCADO			0
PUNTUACIÓN TOTAL			6

Tabla 3.21. Evaluación de métrica de ofuscación mediante Jshrink.

3.5.5. CAFEBABE

OBJETOS	CONTENIDO		PUNTUACIÓN
OFUSCACIÓN DE LAYOUT	Renombrar Identificadores	Public	1
		Protected	1
		Private	1
	Borrado de Número de Líneas		1
OFUSCACIÓN DE DATOS	Encriptado de String		0
	Desestructuracón de Datos		0
OFUSCACIÓN DE CONTROL	Transformaciones en Sentencias y Bucles		0
OFUSCACIÓN DE PROGRAMA			0
DECOMPILACIÓN DE CÓDIGO OFUSCADO			0
DECOMPILACIÓN DE PROGRAMA OFUSCADO			0
PUNTUACIÓN TOTAL			4

Tabla 3.22. Evaluación de métrica de ofuscación mediante CafeBabe.

3.5.6. ZELIX KLASSMASTER

OBJETOS	CONTENIDO		PUNTUACIÓN
OFUSCACIÓN DE LAYOUT	Renombrar Identificadores	Public	1
		Protected	1
		Private	1
	Borrado de Número de Líneas		1
OFUSCACIÓN DE DATOS	Encriptado de String		1
	Desestructuracón de Datos		0
OFUSCACIÓN DE CONTROL	Transformaciones en Sentencias y Bucles		1
OFUSCACIÓN DE PROGRAMA			1
DECOMPILACIÓN DE CÓDIGO OFUSCADO			1
DECOMPILACIÓN DE PROGRAMA OFUSCADO			1
PUNTUACIÓN TOTAL			9

Tabla 3.23. Evaluación de métrica de ofuscación mediante Zelix KlassMaster.

3.5.7. SMOKESCREEN

OBJETOS	CONTENIDO		PUNTUACIÓN
OFUSCACIÓN DE LAYOUT	Renombrar Identificadores	Public	1
		Protected	1
		Private	1
	Borrado de Número de Líneas		1
OFUSCACIÓN DE DATOS	Encriptado de String		0
	Desestructuracón de Datos		0
OFUSCACIÓN DE CONTROL	Transformaciones en Sentencias y Bucles		1
OFUSCACIÓN DE PROGRAMA			0
DECOMPILACIÓN DE CÓDIGO OFUSCADO			0
DECOMPILACIÓN DE PROGRAMA OFUSCADO			0
PUNTUACIÓN TOTAL			5

Tabla 3.24. Evaluación de métrica de ofuscación mediante Smokescreen.

3.5.8. JOGA

OBJETOS	CONTENIDO		PUNTUACIÓN
OFUSCACIÓN DE LAYOUT	Renombrar Identificadores	Public	1
		Protected	1
		Private	1
	Borrado de Número de Líneas		1
OFUSCACIÓN DE DATOS	Encriptado de String		0
	Desestructuracón de Datos		0
OFUSCACIÓN DE CONTROL	Transformaciones en Sentencias y Bucles		0
OFUSCACIÓN DE PROGRAMA			1
DECOMPILACIÓN DE CÓDIGO OFUSCADO			0
DECOMPILACIÓN DE PROGRAMA OFUSCADO			0
PUNTUACIÓN TOTAL			5

Tabla 3.25. Evaluación de métrica de ofuscación mediante JoGa.

3.5.9. JOBFUSCATE

OBJETOS	CONTENIDO		PUNTUACIÓN
OFUSCACIÓN DE LAYOUT	Renombrar Identificadores	Public	1
		Protected	1
		Private	1
	Borrado de Número de Líneas		1
OFUSCACIÓN DE DATOS	Encriptado de String		0
	Desestructuracón de Datos		0
OFUSCACIÓN DE CONTROL	Transformaciones en Sentencias y Bucles		0
OFUSCACIÓN DE PROGRAMA			0
DECOMPILACIÓN DE CÓDIGO OFUSCADO			0
DECOMPILACIÓN DE PROGRAMA OFUSCADO			0
PUNTUACIÓN TOTAL			4

Tabla 3.26. Evaluación de métrica de ofuscación mediante Jobfuscate.

3.5.10. MARVINOBFUSCATOR

OBJETOS	CONTENIDO		PUNTUACIÓN
OFUSCACIÓN DE LAYOUT	Renombrar Identificadores	Public	1
		Protected	1
		Private	1
	Borrado de Número de Líneas		1
OFUSCACIÓN DE DATOS	Encriptado de String		1
	Desestructuracón de Datos		0
OFUSCACIÓN DE CONTROL	Transformaciones en Sentencias y Bucles		0
OFUSCACIÓN DE PROGRAMA			0
DECOMPILACIÓN DE CÓDIGO OFUSCADO			0
DECOMPILACIÓN DE PROGRAMA OFUSCADO			0
PUNTUACIÓN TOTAL			5

Tabla 3.27. Evaluación de métrica de ofuscación mediante Marvin Obfuscator.

3.5.11. YGUARD

OBJETOS	CONTENIDO		PUNTUACIÓN
OFUSCACIÓN DE LAYOUT	Renombrar Identificadores	Public	1
		Protected	1
		Private	1
	Borrado de Número de Líneas		1
OFUSCACIÓN DE DATOS	Encriptado de String		0
	Desestructuracón de Datos		0
OFUSCACIÓN DE CONTROL	Transformaciones en Sentencias y Bucles		0
OFUSCACIÓN DE PROGRAMA			0
DECOMPILACIÓN DE CÓDIGO OFUSCADO			0
DECOMPILACIÓN DE PROGRAMA OFUSCADO			0
PUNTUACIÓN TOTAL			4

Tabla 3.28. Evaluación de métrica de ofuscación mediante yGuard.

3.5.12. CONCLUSIONES

De todas las herramientas de ofuscación analizadas concluimos que Zelix KlassMaster es la más completa. En el apartado de ofuscación no solamente proporciona ofuscación de Estructura, como la mayoría de las aplicaciones, sino que aporta opciones para la ofuscación de Datos y Control. De todo ello, según lo estudiado en el apartado referente a técnicas de ofuscación, se desprende que es la aplicación que mayor oscurecimiento produce en el código. Esto lo hemos podido comprobar al intentar descompilar el *bytecode* ofuscado. Ha sido la única herramienta capaz de oscurecer el código hasta el punto de resultar imposible su descompilación de forma automática.

Además ha sido capaz de ofuscar una aplicación de cierto peso sin problema alguno, teniendo en cuenta la mayoría de casos conflictivos que pueden aparecer al llevarse a fin tal tarea, como por ejemplo: Serialización, *Reflection API calls*, *RMI*, manejo de métodos nativos, manejo de *JavaBeans*.

Por otro lado es una herramienta completa tanto en su manejo, permite ejecución desde línea de comandos o mediante interfaz gráfica (*GUI*), como por las muchas opciones disponibles para la configuración del proceso de ofuscado. El manual de usuario proporcionado es bastante completo a su vez. Además incluye, como se ha comentado en el apartado de procesado correspondiente a esta aplicación, una opción (*Stack Trace Translate*) para recuperar código ofuscado mediante el archivo de renombrado creado en el proceso de ofuscación. Esto resulta útil para *RMI* (invocación de métodos remotos). Esta última opción no ha sido objeto de estudio en este proyecto.

Si nuestro objetivo a la hora de ofuscar es el adelgazamiento del tamaño de nuestra aplicación comprobamos que Zelix KlassMaster también consigue buenos resultados en este aspecto.

El único “pero” a esta herramienta es que es una solución comercial. Su coste es bastante elevado, alrededor de 400$ pero su efectividad está fuera de toda duda. En el caso de querer preservar el *bytecode* de nuestras aplicaciones de ataques de ingeniería inversa sería recomendable la adquisición de esta herramienta.

Frente a Zelix KlassMaster encontramos aplicaciones de ofuscación muy similares: unas ofrecen encriptado (MarvinObfuscator, Jshrink), otras (Proguard, Retroguard) son mas completas a la hora de ofuscar programas de cierto peso; pero sus características son muy similares y no proporcionan ventajas sustanciales unas respecto a otras.

En cualquier caso, si queremos ofuscar nuestras aplicaciones para evitar ataques de ingeniería inversa tenemos dos opciones, una es adquirir Zelix KlassMaster asegurándonos la mejor protección, y otra, si no deseamos gastar dinero, acudir a herramientas *freeware* disminuyendo sustancialmente la protección que conseguimos.

4. TEMPORIZACIÓN

El proyecto se ha dividido en las siguientes tareas:

1) Documentación. En esta primera etapa se han adquirido los conocimientos necesarios para el posterior desarrollo del proyecto. Extraída en su totalidad de Internet (formato digital) al no estar documentado el estudio de la ofuscación hasta el momento. Esta tarea ha constado de:

- Documentación del proceso de ofuscación.
- Documentación de utilización de las aplicaciones (manuales de usuario).

2) Recopilación y testeo de herramientas bajo estudio. Se han recopilado todas las soluciones disponibles y accesibles (freeware y versiones de evaluación comerciales) en la Internet.

3) Procesado. En esta tercera etapa se ha llevado a cabo un análisis experimental según los objetivos marcados en el proyecto. La tarea de procesado ha constado de dos pasos:

- Procesado de ejemplos.
- Procesado de programa lógico.
- Evaluación.

4) Redacción del proyecto fin de carrera. Esta tarea se ha llevado a cabo simultáneamente a la realización de las etapas anteriores.

5) Seguimiento del proyecto fin de carrera. Se ha informado regularmente al profesor tutor, Don Antonio Jesús Sierra Collado, del estado de ejecución en cada momento. Así, ha podido seguir la evolución del proyecto y establecer las medidas correctivas que ha estimado oportunas. La información necesaria para el seguimiento del proyecto, se ha hecho llegar mediante correos electrónicos y reuniones en su despacho.

6) Corrección y entrega de versión definitiva del proyecto fin de carrera.

En el siguiente diagrama se muestra la evolución temporal de las diferentes tareas:

Id	Nombre de tarea	Comienzo
1	Documentación.	mar 01/03/05
2	Estudio y Análisis de la Información.	mié 16/03/05
3	Recopilación de Ofuscadores.	jue 31/03/05
4	Análisis y Testeo.	mié 06/04/05
5	Redacción.	mar 15/03/05
6	Entrega y Corrección.	vie 27/05/05
7	Seguimiento del proyecto.	mar 01/03/05

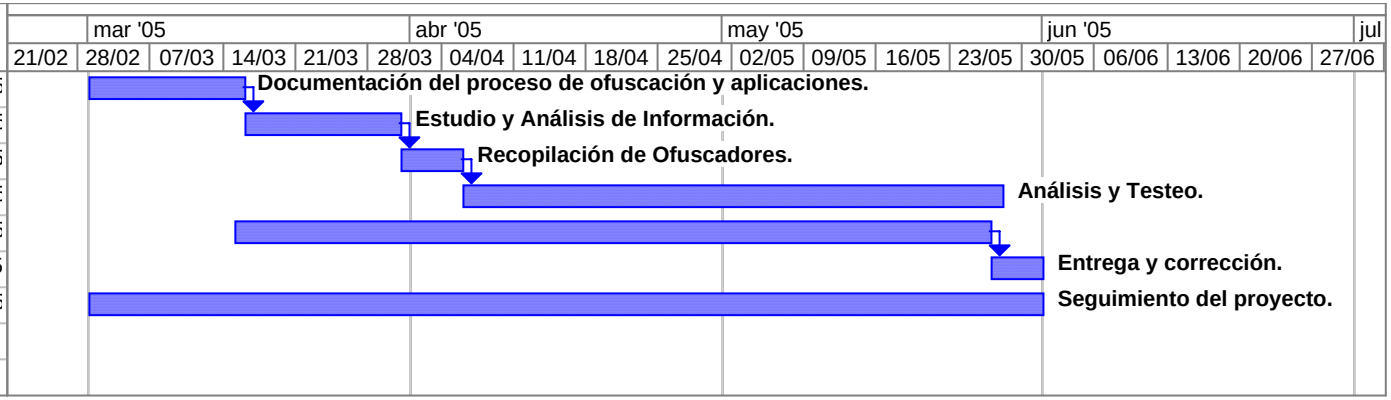


Diagrama 5.1. Temporización del proyecto mediante diagrama de Grant.

5. COSTES

En la realización del presupuesto se han tenido en cuenta los siguientes aspectos:

- El proyecto fin de carrera ha sido desarrollado por una sola persona cuyo nivel profesional podría corresponder al de un ingeniero *junior*.

- La ejecución del proyecto fin de carrera ha comprendido el periodo de tiempo entre Marzo y Mayo (ambos incluidos), del año 2005.

A continuación, se realiza un desglose del presupuesto y un resumen de dicho desglose:

5.1. DESGLOSE DE PRESUPUESTO

El presupuesto del proyecto se puede desglosar en: coste de recursos humanos y coste de los recursos materiales empleados.

5.1.1. COSTE DE RECURSOS HUMANOS

El coste presupuestario de personal se ha calculado de la siguiente forma. Si se supone que se ha desempeñado una jornada de trabajo básica de 40 horas semanales y que se puede establecer un sueldo base de unos 1.800 € mensuales para un ingeniero *junior*, el coste de recursos humanos es:

Periodo de Marzo a Mayo (1.800 € x 3 meses)……...5.400 €

Subtotal: 5.400 €

5.1.2. COSTE DE RECURSOS MATERIALES

Este apartado incluye los costes del *hardware* y consumibles utilizados. Destacar que no hay costes referentes a software debido a que las herramientas bajo estudio son aplicaciones *freeware* o versiones de evaluación.

5.1.2.1. Coste Hardware

El hardware empleado para desarrollar todo el proyecto ha sido un ordenador personal y una impresora. Por lo tanto, el coste de *hardware* es el siguiente:

PC……...900 €
Impresora……..60 €

Subtotal: 960 €

5.1.2.2. Coste de Consumibles

Los costes asociados a los consumibles son los siguientes:

Electricidad consumida…….20 €
Conexión a Internet (ADSL)……135€
Material de oficina (papel, CD's...)……...…20 €

Subtotal: 175 €

5.2. RESUMEN PRESUPUESTO

Todo el coste anteriormente desglosado puede resumirse en la siguiente tabla:

Coste de recursos humanos

Periodo de Marzo a Mayo (1.800 € x 3 meses)……...5.400 €

Subtotal: 5.400 €

Coste de recursos materiales

Coste Hardware
- PC……...900 €
- Impresora……..60 €

Coste de Consumibles
- Electricidad consumida…….20 €
- Conexión a Internet (ADSL)……135€
- Material de oficina (papel, CD's...)……...…20 €

Subtotal: 1135 €

Total: 6535 €

Tabla 5.1. Resumen costes proyecto.

6. REFERENCIAS

6.1. REFERENCIAS DOCUMENTACIÓN

[1] Christian Collberg, Clark Thomborson, Douglas Low, “A Taxonomy of ObfuscatingTransformations”, http://www.cs.arizona.edu/~collberg/Research/Publications/CollbergThomborsonLow97a/index.html

[2] C.S. Collberg, y C. Thomborson, “Watermarking, Tamper-Proofing, and Obfuscation-Tools for Software Protection” IEEE Transactions on Software Engineering, Vol. 28, no. 6, June 2002, http://www.cs.auckland.ac.nz/~cthombor/Pubs/112393-2a.pdf

[3] C.S. Collberg,, “Security Through Obscurity”, http://www.cs.arizona.edu/~collberg/Teaching/620/2002/Handouts/Handout-13.pdf

[4] D. Low: "Protecting Java Code Via Code Obfuscation", http://www.cs.arizona.edu/~collberg/Research/Students/DouglasLow/obfuscation.html

[5] G. Álvarez, “Ofuscación del código”, http://www.cs.arizona.edu/~collberg/Teaching/620/2002/Handouts/Handout-13.pdf

[6] Hongying Lai, “A comparative survey of Java obfuscator”, http://www.cs.auckland.ac.nz/~cthombor/Students/hlai/hongying.pdf

[7] J. Knudsen, “Obfuscating MIDlet Suites with Proguard”, http://developers.sun.com/techtopics/mobility/midp/ttips/proguard

[8] W. A. Marroquín, “Ofuscadores (De la protección relativa del código intermedio)”, http://www.microsoft.com/spanish/msdn/comunidad/mtj.net/voices/art146.asp.
Publicado originalmente por Universal Thread Magazine (www.UTMag.com/Spanish).

6.2. REFERENCIAS MANUALES OFUSCADORES

[1] Proguard, http://proguard.sourceforge.net/

[2] Retroguard, http://www.retrologic.com/retroguard-main.html

[3] Javaguard, http://sourceforge.net/projects/javaguard/

[4] Jshrink, http://www.e-t.com/jshrink.html

[5] CafeBabe, http://www.geocities.com/CapeCanaveral/Hall/2334/Programs/cafebabe.html

[6] Zelix KlassMaster, http://www.zelix.com/klassmaster/

[7] Smokescreen, http://www.leesw.com/

[8] JoGa, http://www.nq4.de/

[9] Jobfuscate, http://www.jobfuscator.com/

[10] MarvinObfuscator, http://www.drjava.de/obfuscator/

[11] yGuard, http://www.yworks.com/en/products_yguard_about.htm

ANEXO

UTILIZACIÓN APLICACIONES

PROGUARD

Para la ejecución de Proguard solo es necesario teclear:

```
java –jar proguard.jar Opciones
```

Las opciones pueden especificarse en uno o más archivos de configuración. Generalmente esta suele ser la forma más típica de proceder, la utilización de un archivo de configuración "*.pro*":

```
java –jar proguard.jar @config.pro
```

También podemos combinar opciones desde la línea de comandos y opciones especificadas desde un archivo:

```
java –jar proguard.jar @config.pro –verbose
```

Las diversas opciones vienen enumeradas en la guía de usuario proporcionada en la web asociada a la aplicación. Las opciones más destacables las detallamos a continuación:

- **-injars** , especifica el archivo de entrada "*.jar*" (o "*.wars*", "*.ears*", "*.zips*" o directorios) de la aplicación a ser procesada. Los archivos "*.class*" serán procesados y transformados en un archivo de salida.

- **-outjars** , especifica el nombre del archivo de salida "*.jar*".

- **-libraryjars** , especifica las librerías de la aplicación que se va a procesar. Estas librerías externas se requieren para resolver posibles dependencias y herencia. No serán incluidas en el archivo de salida.

- **-keep** , para especificar clases y miembros de clases que queramos preservar del proceso de compactación, optimización y ofuscación. Generalmente se preserva la clase que contiene el método principal (punto de entrada). Esto hará que se conserve su nombre original.

- **-printmapping** , especifica que se imprima en un archivo "*.map*" el proceso de renombrado de identificadores, es decir, clases y miembros que han sido renombrados y las transformaciones llevadas a cabo.

- **-overloadaggressively** , especifica que se lleve a cabo un agresivo proceso de *overload* durante la ofuscación. De esta forma puede ocurrir que métodos y miembros sean renombrados con los mismos identificadores, que se cambien los tipos de los parámetros de los métodos, etc.

RETROGUARD

Retroguard está diseñado para ser integrado en el proceso de construcción de aplicaciones Java, de manera que se convierta en un paso más del proceso de diseño de una aplicación.

Para ejecutar Retroguard desde línea de comandos tecleamos:

–jar retroguard.jar [input.jar [output.jar [script.rgs [logfile]]]]

donde:

- **input.jar** , es el archivo original sin ofuscar.
- **output.jar** , fichero resultante tras el proceso de ofuscación.
- **script.rgs** , fichero que contiene las opciones según las cuales se realiza la ofuscación.
- **logfile** , archivo donde se reflejan los posibles errores o avisos en el proceso de ofuscación.

Para la generación del *script* de configuración, Retroguard proporciona una interfaz gráfica que nos permite crearlo de manera sencilla. También pueden escribirse estos *scripts* directamente, en el manual de usuario se especifica como hacerlo. Este manual de usuario se proporciona en la página de Internet asociada (ver tabla 3.2). En nuestro caso concreto hemos acudido a la ejecución de la interfaz.

Debemos tener en cuenta además que si nuestro código necesita de otras librerías externas para resolver posibles dependencias o herencias, es necesario que estas librerías estén disponibles en el *classpath* antes de ejecutar Retroguard. En nuestro caso, no hemos tenido que realizar nada extraordinario porque tenemos J2SDK1.4.2_07 integrado en el *classpath.*

JAVAGUARD

Ofusca archivos de entrada "*.jar*". La salida también se da en el mismo formato. Para ello emplea un *scritpfile* en el que se proporcionan opciones de ofuscación diferentes de las opciones por defecto. Las opciones por defecto son las siguientes:

Borrado de atributos excepto "código", "valores de constantes", "excepciones" e "*Innerclasses*".

Ofusca todos los paquetes, interfaces, clases, métodos y miembros del archivo de entrada exceptuando aquellos en los que se pueda romper el polimorfismo de clases e interfaces fuera del "*.jar*".

Para ejecutarlo desde línea de comandos tecleamos:

java JavaGuard –i <Input.jar> -o <Output.jar> [options]

donde las opciones se resumen en el programa 1.

Programa 1. Listado de opciones de JavaGuard.

```
2.0.6.jar JavaGuard
Usage: java JavaGuard -i <input-file> -o <output-file> [options]

-i <input-file>  The name of an input JAR file to be obfuscated
--input

-d <dir> <regex> Obfuscate all files below the directory that match the
regular
--dir            expression

-o <output-file> The name of the output file that will contain the obfuscated
--output         contents of all input files

-s <script>      The name of a valid JavaGuard script file
--script

-l <log-file>    The name for the log file
--log

--dump           Dump the parsed class tree before obfuscation.

-v               Increment the logging level. To be more verbose specify the
--verbose        parameter several times.

-h               Show this info page.
--help

--version        Show the program version.
```

El *scriptfile* puede contener comentarios y directivas que especifican las opciones para el ofuscador. No se especifica muy bien como crear este *scriptfile*. En él las opciones son referentes a preservar clases y miembros de clases. En nuestro caso estaremos interesados en preservar únicamente el método principal.

Las directivas que pueden aparecer en este archivo de exclusión podrían ser las siguientes:

- **.attribute <atributo>** , el atributo podría ser un determinado código fuente dentro de una aplicación, la tabla de variables locales, etc.

- **.class <option>** , donde el campo opción puede tomar los valores: *public*, *protected*, *method*, *field*; en orden a preservar las clases publicas, las clases definidas como protected o los miembros de una determinada clase respectivamente.

- **.method <método>** , para excluir un método. Esta es la opción que hemos empleado para la ofuscación de nuestro ejemplo.

- **.field <campo>** , para preservar un determinado campo.

JSHRINK

Jshrink es una herramienta que puede ejecutarse desde línea de comandos aunque proporciona una interfaz gráfica (*GUI*) atractiva y de fácil manejo.

En nuestro caso hemos optado por trabajar en línea de comandos en orden a preservar la mayor cantidad de opciones disponibles para aplicar. Para la ejecución de la aplicación Jshrink tecleamos:

java –jar jshrink.exe [opciones] [input file] [opciones] [-o output file]

El archivo de entrada puede ser "*.jar*" o "*.zip*", un directorio o un archivo individual. Si es un directorio entonces todos los archivos que contiene son cargados incluyendo subdirectorios. Sin embargo, los archivos "*.java*" no serán cargados a menos que se especifique la opción adecuada. Si la opción "*-o*" está presente la aplicación se ejecuta en modo comando sin interfaz gráfica. Las opciones desde línea de comandos son órdenes independientes. Las opciones más comunes se detallan a continuación:

- **-classpath path** , especifica la ruta de los archivos que pueden requerirse para resolver dependencias (superclases) a la hora de compactar y ofuscar nuestro código. Los archivos incluidos en el path son leídos exclusivamente y no incluidos en la salida de la aplicación.

- **-l** , muestra por salida estándar información referente al proceso de renombramiento de símbolos, así como estadísticas de tamaño. Puede ser redireccionado hacia un archivo "*.log*".

- **-o outfile** , salva la salida en el archivo outfile. Este archivo de salida puede tener extensión "*.jar*" o "*.zip*". Si no se especifica archivo de salida la aplicación escribirá la salida en el directorio actual.

- **-keep package.classname** , fuerza la salida de esta clase y a que no sea renombrada. El *classname* tiene que estar referido un archivo "*.class*" que se haya cargado previamente, por tanto la especificación del archivo de entrada debe ser anterior. Si no hay ninguna especificación de reservar ningún método y no se encuentra el *main()*, no se procederá al renombramiento de ninguna clase. Destacar que en el classname no es necesario especificar el "*.class*", se supone implícito. Si el argumento termina en '?', entonces todas las clases encontradas serán incluidas para mantener su nombre.

- **-keepMember package.classname.member** , mediante esta opción impedimos que un miembro de una determinada clase sea renombrado. Cuidado porque no se fuerza a incluir la clase que contiene el miembro. Si es un método debemos añadir paréntesis "()".

- **-script filename** , abre el fichero y lee las opciones especificadas como si se leyeran de la línea de comandos. Cada línea del *script* debe contener una sola orden u opción y sus atributos si fueran necesarios. Estos *scripts* pueden generarse mediante la interfaz gráfica que proporciona la aplicación.

- **-stringEncrypt** , sustituye *strings* por llamadas a métodos static de clases que devuelven strings internos extraídos de un código encriptado. Este método supone una cierta sobrecarga.

Estas son las opciones más destacadas. Mediante ellas conseguimos nuestro objetivo de ofuscar el código de los ejemplos de la forma más agresiva posible. Existen más opciones disponibles que pueden consultarse en el manual de usuario proporcionado en la página web de la aplicación, ver tabla 3.2.

CAFEBABE

CafeBabe es una herramienta que proporciona una interfaz gráfica para su utilización. No es posible ejecutarlo desde la línea de comandos. Al cargar un archivo "*.class*", en la ventana principal aparece directamente la estructura del *bytecode*, siguiendo el formato del archivo especificado.

Mediante la opción "*Class-Hound Service*" se nos presenta por pantalla una ventana que contiene toda la información de las clases cargadas, es decir: clases, miembros de clases (campos y métodos), así como el árbol de herencia para la clase seleccionada en cada caso.

Esta herramienta nos permite optimizar el código como paso previo a la ofuscación con el objetivo de borrar información inservible: atributos del código fuente, atributos desconocidos o información redundante en el cuerpo de los métodos. Además algunas aplicaciones asignan etiquetas al *bytecode* al procesarlo que también son borradas en este paso.

Otra tarea llevada a cabo en la etapa de optimización (*strip*) es encontrar todos los métodos que pueden ser declarados como final y establecer el modificador de métodos como final a nivel de *bytecode*.

En la etapa de ofuscación se permite la inclusión de paquetes, clases y miembros de clases. Además se incluye la opción de mover clases de distintos paquetes hacia un paquete anónimo.

ZELIX KLASSMASTER

En primer lugar debemos destacar que Zelix KlassMaster no se ejecuta desde línea de comandos. Proporciona una interfaz gráfica (*GUI*) desde donde podemos controlar todas las opciones.

Al abrir la aplicación Zelix KlassMaster obtenemos una ventana principal que contiene 5 paneles: el panel de herencia, en el que se muestra las clases e interfaces abiertas en orden jerárquico; el panel de propiedades de la clase, que nos permite

modificar el nombre y el especificador de acceso de la clase en cuestión seleccionada; el panel de propiedades de campos, que nos permite las mismas modificaciones; el panel de propiedades de métodos, que nos permite modificar el modificador de acceso del método seleccionado; el panel de propiedades de constantes, que nos permite cambiar el valor de las mismas; y el panel de visión, en el que se muestra el *bytecode* de la clase o método seleccionado.

Lo primero es cargar nuestra aplicación. Para ello solo tenemos que ir a la pestaña *File* y seleccionar *Open.* Se abrirá un navegador que nos permitirá abrir clases especificando un directorio o un archivo "*.zip*" o "*.jar*", en los cuales estarán contenidas. Una vez cargada, podemos ver el *bytecode* completo de nuestra aplicación estructurado por ventanas como hemos explicado anteriormente.

A nosotros, de todas las herramientas proporcionadas por Zelix KlassMaster solo nos interesa el proceso de ofuscación. Para ofuscar solo tenemos que ir a la pestaña de herramientas y seleccionar *Obfuscate*. Esta herramienta renombra paquetes, clases y miembros de clases. En primer lugar nos aparece una ventana: *Obfuscate Name Exclusions*, desde la cual podemos especificar que paquetes, clases y miembros de clases no queremos que sean renombrados. En el manual de usuario se explica detalladamente como realizar esta selección. En nuestro caso nos limitaremos a no renombrar la clase que contiene el método *main*, así como éste. Para ello, dispone de una pestaña que directamente lo selecciona.

De esta forma vamos a conseguir que el renombrado sea lo mas agresivo posible. A continuación pulsamos *Next* para especificar las opciones adecuadas en el proceso de ofuscación. Las opciones que proporciona Zelix KlassMaster son:

- **use input change log file** , de esta forma se especifica un archivo desde el que se realiza el renombrado. Esto tiene cierta utilidad en aplicaciones *RMI* y envío por entregas (*serialization*).

- **produce a change log file** , se genera entonces un archivo *log* en el que se especifica el renombrado llevado a cabo. Es importante generarlo por si queremos emplear la herramienta *Stack Trace Translate* para la traducción de código ofuscado.

- **obfuscate control flow** , Zelix ofuscará todas las sentencias condicionales (*if…else…*) y bucles (*for..while..*) de manera que no puedan ser descompilados directamente a código Java de nuevo. Se dan tres opciones de ofuscación de control: *light*, *normal* y *aggressive*. En nuestro caso seleccionamos la más agresiva. Hay que destacar que esto produce un incremento del tamaño de nuestra aplicación de entorno a un 3%.

- **encrypt string literals** , al activarlo Zelix sustituirá *strings* por *strings* encriptados y añadirá instrucciones al *bytecode* que desencriptarán los *strings* durante la ejecución (*runtime*). Tenemos dos posibilidades a la hora de encriptar: *normal* y *aggressive*. Hemos seleccionado la opción más agresiva. La diferencia estriba en que con esta opción se va más a la hora de borrar alguna constante *static final String* que pueda dejarse sin encriptar en modo normal. La compatibilidad del proceso de encriptado dependerá de la

máquina virtual de Java (*MJV*) que genere el *bytecode*. En general no debemos tener problemas con el compilador para Windows.

- **collapse package** , con esta opción se intenta reducir el tamaño del *bytecode*. Los paquetes que han sido excluidos del proceso de ofuscación no son comprimidos.

- **aggressive method renaming** , el renombrado será más agresivo. No debemos utilizar esta opción cuando nuestra clase pertenece a un entramado o a una librería.

- **keep inner class information** , a nivel de *bytecode* el interior de una clase se distingue por una estructura de nombres y una serie de atributos. Esta información no es crítica para la ejecución de nuestra aplicación. Se proporciona para compiladores, *debuggers* y utilidades similares. Para nuestro caso, no seleccionamos esta opción.

- **keep generics information** , a nivel de *bytecode* J2SDK almacena información genérica en atributos especiales. Como en el caso *de inner class information*, esta información no es del todo necesaria para la ejecución del *bytecode*. Hemos descartado esta opción también.

- **line number tables** , los compiladores Java pueden incluir *line number tables* en el *bytecode* que mapean las instrucciones de *bytecode* en número de líneas del código fuente. Estos números de línea pueden dar pistas a descompiladores. Zelix KlassMaster proporciona tres formas de enfrentarnos a estos números de línea: *delete*, que borra todo rastro de información referente; *scramble*, que mezclará la entrada *line number table* y escribirá el nuevo mapeo de número de línea en el archivo *log*, esto es útil si posteriormente queremos emplear la herramienta *Stack Trace Translate*; y *keep* que dejará los números de línea existentes.

SMOKESCREEN

Smokescreen puede ser utilizado con o sin interfaz gráfica. La *GUI* proporciona una interfaz de fácil manejo que nos permite la selección de las diferentes opciones de ofuscación. Cuando la aplicación es abierta con *GUI* se presenta al usuario una ventana, que tiene una serie de menús, botones y campos. El usuario puede seleccionar los parámetros de ofuscación, el directorio fuente y el destino. Cuando la aplicación se ejecuta desde línea de comandos los parámetros de ofuscación se especifican en un archivo de directivas.

En nuestro caso hemos optado por ejecutar desde línea de comandos. Para ello es necesario especificar un archivo de directivas que contendrá las opciones para el proceso de ofuscación. Cada línea de este archivo debe contener una directiva. Estas directivas comienzan con un asterisco ‘* ‘.

Todas las posibles directivas a aplicar vienen especificadas en el completo manual de usuario de la aplicación. Las más destacadas y las que hemos utilizado en nuestros ejemplos son:

- **source_directory** , para especificar la fuente. Puede ser un directorio, un archivo "*.zip*" o uno "*.jar*".
- **destination_directory** , para especificar la salida. Es conveniente que sea del mismo tipo que la entrada.
- **superclass_path** , indica el *path* para resolver posibles dependencias.
- **use_class_loader_for_superclasses** , si se especifica esta directiva estamos indicando que la localización de las superclases se tomarán del *classpath* de nuestro sistema. Esta directiva anula a la anterior.
- **log_changes** , hace que el proceso de renombrado se especifique en un archivo *log* de salida.
- **overwrite_classfiles** , para indicar que los archivos "*.class*" de salida pueden ser sobreescritos.
- **classes all_classes** , indica que todas las clases deben ser consideradas en el proceso de ofuscación.
- **methods all_methods** , indica que todos los métodos han de ser incluidos en el proceso de ofuscación. Debemos tener cuidado porque el método *main* no debe ser modoficado. En caso contrario nuestra aplicación dejaría de funcionar. Existe otra opción para excluirlo.
- **fields all_fields** , análogo pero para campos dentro de una clase.
- **bytecode shuffle_stack_operations** , opción de ofuscación de *bytecode* que realiza cambios en el orden de ejecución de instrucciones para que la decompilación sea mucho más complicada.
- **bytecode add_fake_exceptions** , introduce en el *bytecode* bloques de excepciones que se superponen a los bloques de control existentes.
- **bytecode change_switch_statements** , cambia el flujo de control en sentencias de tipo *switch.*
- **bytecode encrypt_strings** , modifica *strings* de manera que no aparacen en su forma original dentro del *bytecode* ofuscado.
- **-name_exclusion_directives** , para especificar clase, método o campo que quiera excluirse del proceso de ofuscación. En nuestro caso solamente hemos excluido el método *main*.

JOGA

JoGa ofrece una interfaz gráfica muy intuitiva que nos permite llevar a cabo fácilmente la optimización y ofuscación de nuestra aplicación, mediante la navegación por sus menús.

Se ofrecen las siguientes opciones de configuración:

- **remove debug infos** , para borrar informaciçon que introduce el debbuger durante la compilación.
- **compress constant pools** ,
- **merge classes** , asocia clases.
- **inlined methods** , introduce el cuerpo del método directamente en algunas llamadas que se producen a éstos.
- **Remove** , partes que se pueden borrar si no son usadas.
- **Rename** , opciones de renombrado.
- **byte code optimizations** , optimizaciones que pueden llevarse a cabo sobre el *bytecode*, como por ejemplo, optimizarlo para que ocupe el menor tamaño posible.

La optimización comienza por el análisis de los archivos "*.class*". JoGa nos pedirá que confirmemos alguna información durante este paso del proceso. En primer lugar JoGa resolverá posibles dependencias con superclases en orden a preservar éstas. Después de esto deberemos identificar el tipo de programa, esto quiere decir si hemos construido una *applet*, una aplicación o una *API*. Dependiendo de la opción seleccionada tendremos que especificar el punto de entrada para el proceso de optimización. El resto del proceso correrá de forma automática.

Debemos destacar que JoGa proporciona información muy detallada del proceso llevado a cabo mediante la selección de un archivo de salida "*logfile.txt*" y uno de resultados "*results.txt*".

JOBFUSCATE

Jobfuscate es una herramienta ejecutada desde línea de comandos que permite ser fácilmente integrada entornos de desarrollo Java. Para ejecutar sólo tenemos que teclear:

```
jobfuscate [-options] class
```

Mediante *class* especificamos las clases que queremos ofuscar. Generalmente solo necesitamos hacer referencia a la clase que contiene en método principal y Jobfuscate busca las posibles dependencias.

Las opciones pueden especificarse en línea de comandos o generar un *scriptfile* que las contenga todas. Nosotros hemos optado por lo segundo. Las opciones de las que disponemos en esta versión de prueba se detallan a continuación:

- **-log:<file>** , redirecciona stderr/stdout del programa al archivo especificado.
- **-out:<file>** , especifica el archivo "*.jar*" de salida. Todos los archivos ofuscados se introducen por defecto en un contenedor de este tipo.
- **-xm:<class.method>** , excluye el método indicado del proceso de renombrado.
- **-xf:<class.field>** , excluye el campo indicado del proceso de renombrado.
- **-xc:<class>** , análogo para la clase especificada.
- **-sys:<class>** , identifica que clases son del sistema y por tanto no serán ofuscadas. Mediante esta opción especificamos la ruta de superclases para que puedan resolverse posibles dependencias.
- **@<file>** , mediante esta opción indicamos a la aplicación que el archivo es un *sriptfile* donde se recogen las opciones de configuración.

Existen más opciones disponibles que pueden consultarse en línea de comandos si ejecutamos la aplicación sin ningún tipo de argumento. Hemos recogido las más interesantes y las que emplearemos en la ofuscación de nuestro ejemplo.

MARVINOBFUSCATOR

Destacar que es la aplicación para la cual hemos tenido mayores problemas a la hora de ejecutarla. No se proporciona un manual de usuario detallado, solamente un *script* de configuración necesario su ejecución, ver el siguiente programa.

Programa 2. Script de configuración MarvinObfuscator, config.txt.

```
// Template for a Marvin Obfuscator config file
////////////////////////////////////////////////////////////////////////////
// Directories and jar/zip archives where your application's classes reside
// (relative to project directory)

classpath=("C:\j2sdk1.4.2_07\jre\lib\rt.jar","C:\MarvinObfuscator\prueba")

////////////////////////////////////////////////////////////////////////////
// The name of your applet or applications's main class
```

```
// (e.g. executable class, servlet or applet)
// (as it would appear in <applet code=...> or in "java ...")
// Note the double parentheses!

mainClasses=(("Test1"))

// If you want your main class to be called differently after the obfuscation,
// you can enter the new name (including package) like this:
//
// mainClasses=(("myapp.Main" newName="main"))
//
// You can also have more than one main class:
//
// mainClasses=(("myapp.ServletA") ("myapp.ServletB"))

////////////////////////////////////////////////////////////////////////////
// Names of methods (without class name) that are accessed via reflection.
// The obfuscator will not change the names of these methods.

//preserveMethodNames=("methodName1","methodName2","reflected*")

////////////////////////////////////////////////////////////////////////////
// Classes that are accessed via reflection.
// The obfuscator will not change the names of these classes.
// Note: Method names within these classes will still be obfuscated.
//       (If you need to preserve method names too, what you probably want
//       is "externalClasses".)
// Note: You can use the wildcard character (*).

//preserveClassNames=("mypackage.SpecialClass")

////////////////////////////////////////////////////////////////////////////
// Classes that are used by your application,
// but should not be included in the obfuscated jar file.
// The obfuscator also ensures that references to these classes continue to
// work ("referencing" includes calling, subclassing and implementing).
// Note: Any part of these libraries that is actually used must be in the
//       classpath (either system classpath or classpath= line in this file).

//externalClasses=("com.company.externallibary.*","org.apache.*")

////////////////////////////////////////////////////////////////////////////
// Locations of resource files.
// If you want resource files (images, properties files, ...) to be included
// in the jar file, specify the resource directories or archives here.
// Do NOT specify individual resources files (these will be interpreted as
// archives).
// Note: Directories will be scanned recursively.
// Note: .class files are excluded automatically.
// Note: You will usually want to reuse entries from the classpath= line
//       (classes and resources are often bundled in the same jar files).

//resources=("resourcedir","resources.jar")

////////////////////////////////////////////////////////////////////////////
// Advanced obfuscation features.

encryptStrings=true  // You can try to set this to false for troubleshooting
```

Para su utilización solo es necesario cambiar el *path* al de nuestro sistema en concreto e indicar la clase principal de nuestra aplicación como punto de entrada para el proceso de ofuscación. A pesar de la facilidad de configuración del *script*, fueron necesarias varias simulaciones hasta conseguir el resultado deseado.

Para la ejecución de MarvinObfuscator es necesario indicar el directorio en el que se encuentra la aplicación a ofuscar, en el que previamente hemos introducido el *script* de configuración, así como el nombre del archivo de salida:

C:\MarvinObfuscator\obfuscate prueba Test1res.jar

Donde contamos con un ejecutable "*obfuscate.bat*" en el que se especifica la ruta de la máquina virtual de Java (JVM) y la instrucción de llamada a la aplicación Marvin. Ver programa 3.

Programa 3. Ejecutable MarvinObfuscator.

```
@echo off
set JAVALIB=C:\j2sdk1.4.2_07\jre\bin
java -cp marvinobfuscator.jar;%JAVALIB% drjava.marvin.Obfuscate
```

YGUARD

Para hacer uso de la tarea de ofuscación de yGuard debemos tener instalado y configurado Ant apropiadamente. Para ejecutar yGuard deberemos crear un *script* "*build.xml*" que contendrá las opciones del proceso de ofuscación. La forma de generar este *script* se detalla en el manual de usuario de la aplicación. Cabe destacar que es uno de los manuales mas detallados que hemos encontrado en el análisis de los ofuscadores, cosa de agradecer porque no todos los usuarios se hallan familiarizados con XML y el compilador Ant.

Las opciones se detallan en lo que se denomina un documento de definición de tipos (*DTD*). Las más destacadas las reseñamos a continuación:

- **Elemento obfuscate** , para el que se pueden especificar los siguientes atributos: "*logfile*", que determina el nombre del archivo de incidencias que se genera durante la ofuscación; "*mainclass*" , indica el nombre de la clase principal de la aplicación,si se especifica ni esta clase ni el método main sufren ofuscación; "*replaceClassNameStrings*", es un atributo boleano para renombrar strings con nombres de clases.

- **Elemento inoutpair** , al menos un elemento de este tipo ha de especificarse para que se lleve a cabo la ofuscación. Este elemento especifica el archivo "*.jar*" que contiene los bytecodes a ofuscar y el "*.jar*" de salida. Tiene dos atributos: "*in*", contenedor de entrada y "*out*", contenedor de salida.

- **Elemento ExternalClasses** , si el archivo a ofuscar tiene dependencias con librerias externas este elemento puede ser usado para indicar el *path* de las entidades requeridas.

- **Elemento property** , para indicar opciones de ofuscación. Estas opciones vienen especificadas mediante nombre y valor, dos atributos. Así tenemos:

  - *Error-checking* , tomando el valor de "pedantic" especifica que se pare el proceso de ofuscación si se detecta cualquier error.
  - *Name-Scheme* , para indicar que utilice diferentes mecanismos de renombrado durante la ofuscación, este atributo puede fijarse con los siguientes valores: "*small*" , producirá nombres cortos; "*best*", nombres que complican la descompilación; "*mix*" , una mezcla de los dos anteriores.

- **Elemento expose** , para especificar clases, métodos o campos. En este elemento puede especificarse la eleminación de mucha información innecesaria que no debería descompilarse. Para controlar esta información tenemos varios atributos boleanos:

  - *linenumbertable* , determina si la tabla de número de línea debe ser borrada. Por defecto se borra.
  - *localvariabletable* , análogo para la tabla de variables local. Por defecto se borra.
  - *sourcefile* , determina si el nombre del archivo de código fuente original debe mantenerse en el *bytecode* ofuscado. Por defecto se borra.

- **Elemento method** , mediante este elemento especificamos métodos que deben preservarse del proceso de ofuscado. En nuestro caso lo utilizaremos para excluir el método main.

Teniendo en cuenta todas estas opciones y siguiendo las indicaciones descritas en el manual de usuario hemos generado el siguiente *script* (ver programa 4):

Programa 4. Script de configuración para yGuard, build.xml.

```
<?xml version="1.0" encoding="UTF-8"?>
    <!-- file build.xml in your project root directory -->
    <!-- ANT build script for yfiles                          -->
        <project name="proyecto" default="obfuscate" basedir=".">
    <!-- obfuscate -->
    <target name="obfuscate">
    <taskdef name="obfuscate" classname="com.yworks.yguard.ObfuscatorTask"
            classpath="yguard.jar"/>
        <!-- the following can be adjusted to your needs -->
        <obfuscate logfile="obfuscationlog.xml"
                replaceclassnamestrings="false">
          <inoutpair in="Test1.jar" out="Test1res.jar"/>
         <externalclasses>
                <pathelement location="C:\j2sdk1.4.2_07\jre\lib\rt.jar"/>
          </externalclasses>

         <property name="error-checking" value="pedantic"/>
          <property name="naming-scheme" value="small"/>
          <expose linenumbertable="false" localvariabletable="false">
                <method class="Test1" name="void main(java.lang.String[])" />
          </expose>
        </obfuscate>
    </target>
    </project>
<!-- end file build.xml -->
```