UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

# Esteganografía usando la redundancia en el juego de instrucciones de la arquitectura Intel x86-64

**TRABAJO FIN DE GRADO**

Grado en Ingeniería Informática

*Autor:* Melero Bargues, Carlos
*Tutor:* Ripoll Ripoll, José Ismael
*Cotutor:* Marco Gisbert, Héctor

Curso 2018-2019

# Resumen

Mediante el uso de técnicas esteganográficas, escondiendo mensajes dentro de otros objetos, el trabajo realizado ha consistido en el diseño y desarrollo de una herramienta capaz de ocultar información aplicando estas técnicas sobre ejecutables de Windows de 32 bits. Esta información oculta puede utilizarse, entre muchas otras cosas, para identificar al causante de una filtración de software (marcas de agua) o usarse como mecanismo encubierto de comunicación entre disidentes (canal encubierto).

Para llevar acabo la implementación de la herramienta primero se analizará en qué partes del juego de instrucciones de Intel X86 existe redundancia para ocultar información. La herramienta analizará las instrucciones de los ejecutables usando Capstone para detectar conjuntos de instrucciones susceptibles de ocultar información y aplicará traducciones basadas en el análisis de redundancia.

**Palabras clave:** CISC, X86, Intel, ensamblador, esteganografía, marcas de agua, canal encubierto

# Abstract

By using steganographic techniques, i.e. hiding messages inside other objects, this work consists in the design and development of a tool able to hide information by applying these steganographic techniques to Windows 32 bits executables. That hidden information can be used, among other things, to identify the culprit of a software leak (watermarking) or it can be used to communicate covertly between disidents (covert channel).

To implement the tool an analysis of the Intel X86 instruction set needs to take place first. The places where redundancy exists and are susceptible to hide information must be uncovered. The tool will analyze the executables using Capstone to detect zones susceptible of hidding information and will apply the translations based upon the redundancy analysis.

**Key words:** CISC, X86, Intel, assembler, steganography, watermarking, covert channel

# Índice general

# Índice de figuras



# Índice de tablas

# CAPÍTULO 1
# Introducción

La esteganografía es el estudio y la aplicación de técnicas para ocultar mensajes dentro de otros mensajes u objetos. De esta forma se establece un canal encubierto de comunicación, de modo que el propio acto de la comunicación pase inadvertido para observadores que tienen acceso a ese canal.

Las aplicaciones de la esteganografía son amplias y se trata de un campo del conocimiento que ha estado ganando importancia recientemente, especialmente los canales encubiertos [14] [3] [12]. La capacidad de ocultar un mensaje en una trasmisión de aspecto inocente para evitar la censura o el análisis de las comunicaciones es algo que se está volviendo necesario en algunos regímenes.

Los ordenadores son máquinas que siguen instrucciones sencillas para crear comportamientos complejos. El engranaje más importante de esta máquina es el procesador. Este se encarga de procesar las instrucciones más simples. Existen diferentes niveles de abstracción para generar esas instrucciones de máquina según su expresividad. Las instrucciones se pueden expresar como un mnemónico del opcode que sería la menor expresividad posible ya que se está diciendo directamente operación por operación qué debe hacer el procesador. Por otro lado existirían abstracciones como la *Programación Orientada a Objetos* (OOP) cuyos conceptos abstractos deben compilarse a instrucciones sencillas que el procesador comprenda.

Dentro de los distintos tipos de procesador hay dos categorías diferenciadas; *Computador con Conjunto de Instrucciones Complejas* (CISC) y *Computador con Conjunto de Instrucciones Reducidas* (RISC). Este trabajo se centrará en CISC ya que el abanico de instrucciones que procesan es mayor. Como caso de estudio se seleccionará la arquitectura *Intel Architecture, 32-bit* (IA32), conocida también como x86.

Dentro de la complejidad de ese conjunto de instrucciones que el procesador comprende, existen múltiples formas de conseguir el mismo resultado. Los compiladores determinan qué conjunto de instrucciones necesitarán para obtener cierto resultado. Puesto que hay múltiples formas de obtener ese resultado es posible codificar información en esa decisión. Así pues sería posible usar las instrucciones de un ejecutable como canal encubierto.

## 1.1 Motivación

La esteganografía es una rama del conocimiento fascinante que obliga a entender la información de forma más amplia. El mismo mensaje puede transportar diferente información según el intérprete.

La arquitectura de los procesadores es rebuscada y llena de detalles que hacen que sean un objeto de estudio apasionante. En especial los *conjuntos de instrucciones* (ISAs), cada recoveco del diseño de los ISAs alberga el resultado de decisiones complejas de aquellos que lo diseñaron.

El uso de ejecutables como medio de transporte es poco común tal y como se verá en el capítulo analizando el Estado del arte. Colocar el foco de atención en éstos es otra motivación para este trabajo.

## 1.2 Objetivos

El objetivo principal es diseñar una herramienta capaz de almacenar información en un fichero *Portable Executable* (PE) de Microsoft Windows para IA32. Para cumplirlo es necesario definir qué se entiende por conjunto de instrucciones equivalentes. También hay que detectar clases de equivalencia susceptibles de almacenar información, para ello hay que determinar cómo codificar la información en ese espacio de símbolos que son las instrucciones de IA32.

Como objetivo secundario hay que diseñar la herramienta de forma que sea tan extensible y adaptable como sea posible.

Además se plantea como objetivo analizar la eficacia de la herramienta sobre un corpus determinado, detectar deficiencias y plantear posibles soluciones.

## 1.3 Impacto esperado

Uno de los casos de uso más interesantes es el de las marcas de agua digitales [6]. Es decir, dejar una marca en un ejecutable que de forma unívoca determine su procedencia.

Las empresas desarrolladoras de software intentan proteger su propiedad intelectual mediante mecanismos anti-copia que son constantemente esquivados. Como mecanismo para desincentivar la piratería las empresas más cautas hacen firmar documentos legales a sus clientes, como los de no divulgación. En caso de filtración del software no es posible determinar qué cliente puede ser legalmente responsable ya que no hay vinculación entre el software y el cliente.

Aquí es donde la capacidad de utilizar la esteganografía sobre los binarios de ese software puede resultar de utilidad. Si los binarios de cada cliente llevan una firma digital e invisible, es trivial por parte de quien quiere proteger su propiedad intelectual saber de dónde ha venido la filtración.

## 1.4 Estructura de la memoria

**Estado del arte**
Se analiza si existe alguna herramienta o trabajo similar, se plantean sus posibles deficiencias y se exponen en contraste las propuestas que se hacen en este trabajo.

**Análisis del problema**
Se identifican las tareas y partes necesarias para abordar el problema.

**Diseño de la solución**
Se diseña una solución que permita solucionar los problemas del capítulo previo. Aquí también se explica con detalle qué métodos y librerías externas componen la implementación final.

**Desarrollo de la solución propuesta**
Se explica con mayor detalle el funcionamiento de cada una de las partes del diseño. Cada sección incluye detalles de implementación.

**Pruebas**
En este capítulo se describe el corpus sobre el que se medirá la eficacia de la herramienta. Se mostrarán múltiples estadísticas y se analizará su resultado.

**Conclusiones**
Se reflexiona sobre el resultado del trabajo y sobre si los objetivos se han cumplido.

**Trabajos futuros**
Se tratan las limitaciones y deficiencias descubiertas y se habla de cómo con futuros trabajos podrían ser solucionadas.

**Bibliografía**
Incluye referencias a los trabajos consultados para el desarrollo del trabajo.

**Glosario**
Contiene las definiciones de los términos técnicos que pueden estar fuera del vocabulario del lector.

**Siglas**
Contiene el significado de todas las siglas usadas durante la redacción del trabajo.

**Código fuente**
Instrucciones para acceder al código fuente de la implementación descrita en el trabajo.

## 1.5 Convenciones

Durante el trabajo se presentarán valores numéricos codificados de varias formas según convenga para el análisis. Todos los valores numéricos irán en una fuente monoespaciada y según su base irán indicados con un prefijo o sin prefijo si es decimal. Como separador para simplificar la lectura se usaría una barra baja. Ejemplos:

- `0b0011_1001` binario
- `0xF00DBABE` hexadecimal
- `1337` decimal

También para el análisis de las instrucciones se utilizará un tabla como la 1.1.

| Extracto de ejemplo | | |
|---|---|---|
| Dirección | Opcode | Instrucción |
| 0x10011770 | 8B FF | mov edi, edi |
| 0x10011772 | 55 | push ebp |
| 0x10011773 | 8B EC | mov ebp, esp |

Tabla 1.1: Tabla de instrucciones de ejemplo

# CAPÍTULO 2
# Estado del arte

El uso de ejecutables como medio de transporte esteganográfico es poco común.

Hydan[1] es la única herramienta que se define como tal y además tiene un trabajo académico asociado[7].

Fuera de la esteganografía se utiliza técnicas similares de recodificación y reensamblado de instrucciones dentro del campo del malware. En este campo las técnicas se utilizan en el metamorfismo [11] o polimorfismo [3] con objetivo de alterar binarios para evadir herramientas antivirus. Con el paso del tiempo estas técnicas han perdido efectividad ya que las herramientas de defensa dan menos peso al análisis estático de binarios [13] [1] [15] [9] [5] [1].

## 2.1 Crítica al estado del arte

Hydan[7] carece de un mecanismo de amplicación simple. La implementación en C es rígida y requiere de un amplio conocimiento sobre la arquitectura de toda la solución para poder aumentar su funcionalidad. La herramienta está abandonada y no ha tenido mucha recepción posiblemente porque no define un caso de uso claro.

## 2.2 Propuesta

La propuesta que se abordará en este trabajo tiene como objetivo facilitar la ampliación de los métodos de codificación de información. La creación de un framework o motor que permita extender fácilmente las posibilidades de la herramienta.

[1] http://www.crazyboy.com/hydan/

# CAPÍTULO 3
# Análisis del problema

Para cubrir los objetivos planteados hay que identificar las diferentes partes necesarias que tendrán que cooperar.

## 3.1 La estructura PE

Los ficheros Portable Executable almacenan la estructura necesaria para que el cargador de binarios de Windows transforme una imagen ejecutable en un proceso listo para ser ejecutado.

VirusShare_000d2c52523cc3c207

| Name | Virtual Size | Virtual Address | Raw Size | Raw Address | Reloc Address | Linenumbers | Relocations N... | Linenumbers ... | Characteristics |
|---|---|---|---|---|---|---|---|---|---|
| | | | | | | | | | |
| Byte[8] | Dword | Dword | Dword | Dword | Dword | Dword | Word | Word | Dword |
| .text | 000D1F44 | 00001000 | 000D2000 | 00001000 | 00000000 | 00000000 | 0000 | 0000 | 60000020 |
| .rdata | 00035CF6 | 000D3000 | 00036000 | 000D3000 | 00000000 | 00000000 | 0000 | 0000 | 40000040 |
| .data | 00009098 | 00109000 | 00005000 | 00109000 | 00000000 | 00000000 | 0000 | 0000 | C0000040 |
| .rsrc | 000CA314 | 00113000 | 000CB000 | 0010E000 | 00000000 | 00000000 | 0000 | 0000 | 40000040 |

**Figura 3.1:** Ejemplo de secciones de un ejecutable.

Una sección 3.1 define las características de un bloque de bytes. Estas características como su dirección virtual, su tamaño virtual, o permisos de memoria permiten que el cargador de binarios de Windows cree páginas de memoria que se correspondan con las que el compilador determinó para el ejecutable.

Para cubrir los objetivos definidos hay que analizar las secciones ejecutables. Una sección ejecutable es aquella que está marcada como susceptible de contener instrucciones que el procesador interpretará.

Las secciones que suelen generar la mayoría de compiladores son: [16]

- **.text** : Donde van las instrucciones del programa. Esta sería la sección más relevante para este trabajo. La página de memoria donde se aloje su contenido tendrá que ser ejecutable.
- **.reloc** : Se almacena la tabla de recolocación que permite que Windows mueva la dirección base del ejecutable y arregle las referencias de las instrucciones.
- **.data** : Se almacena el valor de las variables inicializadas. La página que contenga la sección deberá estar marcada como escribible.

- **.rdata** : Aquí van los datos de solo lectura. La página que contenga a esta sección no será escribible.

- **.idata** : Esta sección almacena la tabla de importaciones. Esta tabla indica a Windows qué librerías tendrá que cargar y de qué funciones tendrá que calcular la dirección.

Los nombres pueden cambiar pero lo importante son las características que definen las secciones.

Aunque técnicamente es posible que un binario tenga más de una sección con la *flag* de **IMAGE_SCN_MEM_EXECUTE** [16], que indica que la sección sería copiada a una página de memoria ejecutable, no es algo común. Esta técnica es gastada por algunas muestras de *malware*[4] [10].

Nada evita que una sección marcada como ejecutable no tenga instrucciones reales. Para evitar esa posible confusión, y acabar alterando datos en lugar de código, se debe escoger la sección ejecutable que contenga la **AddressOfEntryPoint** [16]. La **AddressOfEntryPoint** señala a la primera instrucción que debe ejecutarse una vez el ejecutable ha sido cargado en memoria.

## 3.2 Desensamblar instrucciones IA32

Una vez encontrada la sección que contiene las instrucciones ejecutables será necesario interpretarlas y determinar si son susceptibles de ocultar información. Este proceso de interpretación de los opcodes se conoce como desensamblado.

Durante el desensamblado es importante que se obtenga cuanta más información posible de cada instrucción ya que algunas de las formas de ocultar información pueden depender de sutilezas en la codificación de la instrucción.

La IA32 es una arquitectura compleja, con instrucciones de tamaño variable y varios mecanismos de direccionado. De la posibilidad de tener instrucciones de tamaño variable y poder colocar el *contador de programa* (PC) (en el caso de IA32, **EIP**) a mitad de una instrucción se puede dar el siguiente caso. En la tabla 3.1 se puede ver las instrucciones que un decompilador encuentra en el bloque de código y en la tabla 3.2 se observa las instrucciones que ejecutaría el procesador si empezamos en la dirección `0x40000001`.

Como se observa en las tablas el resultado de la ejecución varía profundamente. Un decompilador que decidiese empezar su análisis en la dirección `0x40000000` interpretaría unas instrucciones y otro que lo hiciese solo un byte después desensamblaría otras.

La funcionalidad real del bloque de opcodes no se puede saber sin antes detectar cómo es ejecutado [2].

| Dirección | Opcode | Instrucción | Comentario |
|---|---|---|---|
| 0x40000000 | 68 90 90 6A 01 | push 0x16a9090 | Guardamos un valor en la pila |
| 0x40000005 | 58 | pop eax | EAX := 0x16a9090 |
| 0x40000006 | C3 | ret | Devolvemos la ejecución |

Tabla 3.1: Bloque de código problemático.

| Dirección | Opcode | Instrucción | Comentario |
|---|---|---|---|
| 0x40000001 | 90 | nop | Nada |
| 0x40000002 | 90 | nop | Nada |
| 0x40000003 | 6A 01 | push 0x1 | Guardamos valor en la pila |
| 0x40000005 | 58 | pop eax | EAX := 0x1 |
| 0x40000006 | C3 | ret | Devolvemos la ejecución |

Tabla 3.2: Bloque de código problemático. Desensamblado descartando el primer byte.

## 3.3 Estego-algoritmos

Se deberán analizar los diferentes tipos de de redundancia e implementar los estego-algoritmos que incorporen la información sobre las instrucciones.

A continuación se describen las diferentes formas de redundancia a explorar.

### 3.3.1. Redundancia de codificación

Se trata de la redundancia implícita a la forma en la que las instrucciones están construidas.

Un ejemplo de este mecanismo es el caso de la existencia de codificaciones especiales para algunas instrucciones con el registro **EAX** e inmediatos de tamaño byte [8]. Puede observarse en la siguiente tabla 3.3 como diferentes opcodes se traducen a la misma instrucción.

| Dirección | Opcode | Instrucción |
|---|---|---|
| 0x00000000 | 83 C0 01 | add eax,0x1 |
| 0x00000000 | 81 C0 01 00 00 00 | add eax,0x1 |
| 0x00000000 | 05 01 00 00 00 | add eax,0x1 |

Tabla 3.3: Instrucciones equivalentes codificadas en diferentes opcodes

### 3.3.2. Redundancia semántica

Se entiende como redundancia semántica aquellas instrucciones, o conjuntos de instrucciones, diferentes que obtienen el mismo resultado.

Un ejemplo de este mecanismo es la carga copia de un registro a otro. En la tabla 3.4 se observan bloques de instrucciones que obtienen el mismo resultado; **EAX** contendrá el valor de **EBX**.

| Dirección | Opcode | Instrucción |
|---|---|---|
| 0x00000000 | 89 D8 | mov eax,ebx |
| 0x00000000 | 8D 03 | lea eax,[ebx] |
| 0x00000000 | 53 | push ebx |
| 0x00000001 | 58 | pop eax |

Tabla 3.4: EAX := EBX

## 3.4 Ensamblado de instrucciones IA32

Una vez determinadas las instrucciones o los conjuntos de instrucciones susceptibles de albergar información hay que aplicar el estego-algoritmo. Éste puede necesitar ensamblar nuevas instrucciones o alterar las existentes dependiendo del tipo de redundancia de la que se haga valer.

## 3.5 Seguridad de la información

Ya habiendo almacenado la información hay que plantearse si la técnica esteganográfica es segura o únicamente usa *seguridad por oscuridad*[1].

Si una tercera parte fuese conocedora de los estego-algoritmos utilizados, el *mensaje encubierto* sería fácilmente extraíble.

Para evitar este problema se plantea la posibilidad de cifrar de antemano el *mensaje encubierto*. También es importante evitar que un adversario pudiese alterar el mensaje sin ser detectado.

[1] https://es.wikipedia.org/wiki/Seguridad_por_oscuridad

# CAPÍTULO 4
# Diseño de la solución

## 4.1 Arquitectura del sistema

En la figura 4.1 puede observarse un diagrama de clases de todo el sistema. A grandes rasgos el diagrama de flujo se desglosa en los bloques señalados en la figura 4.2.

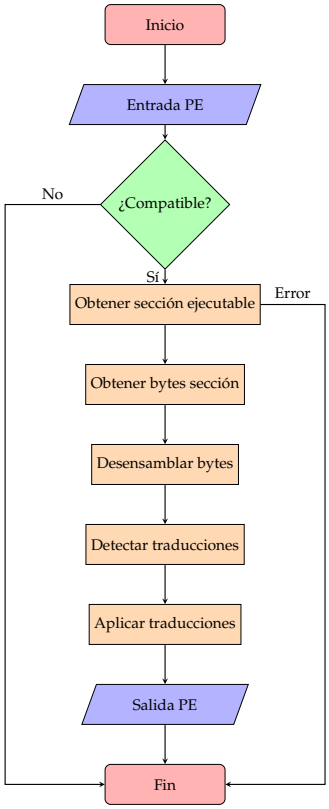


**Figura 4.2:** Diagrama de flujo de la ejecución

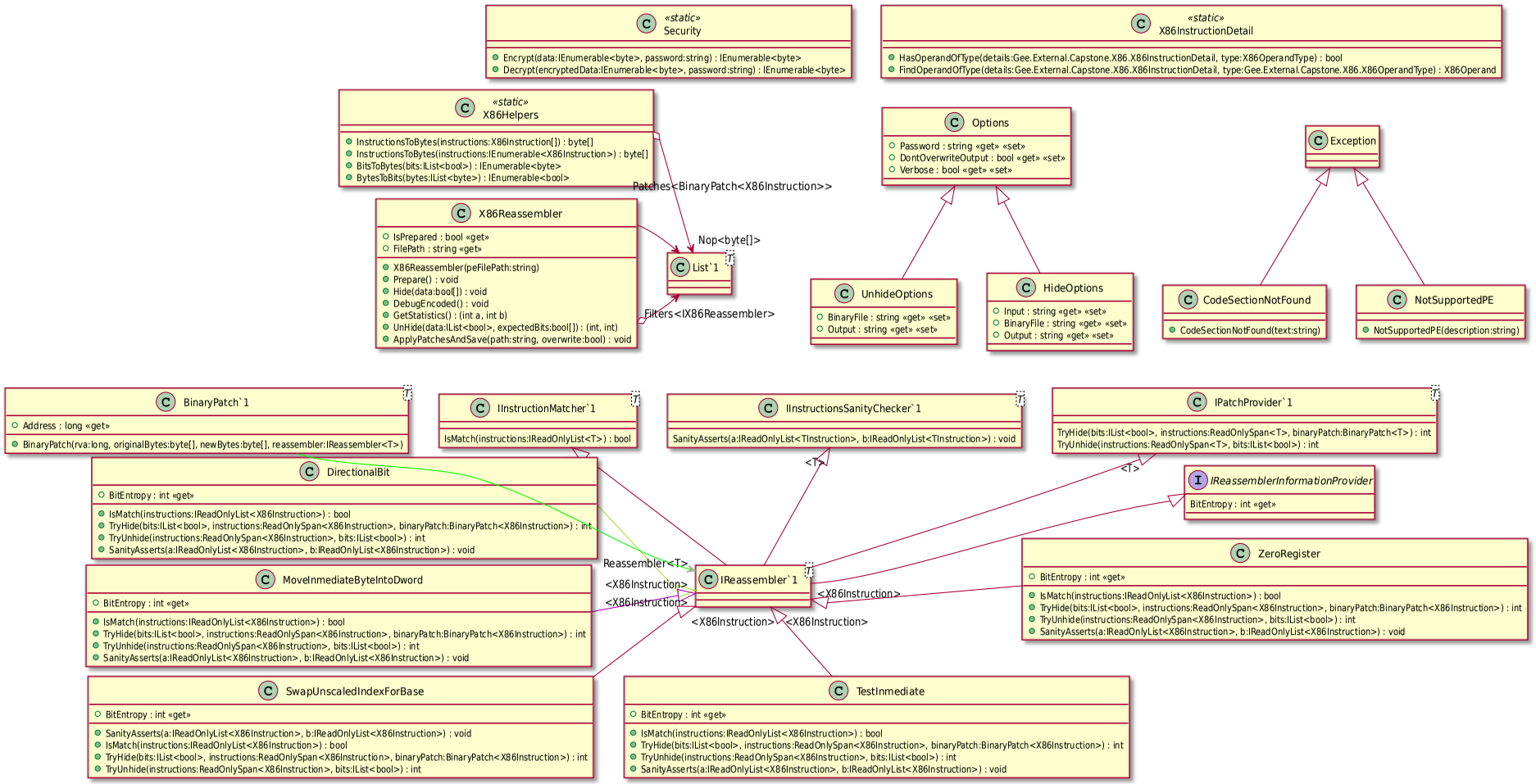


**Figura 4.1:** Diagrama completo. Clases auxiliares eliminadas por espacio.

Descripción de los bloques principales:

**Reensamblador**
Este bloque es el encargado de a partir de una ruta de un fichero PE:

- Comprobar que se trata de un ejecutable válido.
- Obtener la sección de código de interés.
- Leer el bloque de código.
- Esconder un array de bits en ese bloque de código.
- Ejecutar el conjunto de algoritmos de traducción sobre las instrucciones del bloque.
- Recuperar un array de bits de ese bloque de código.

**Traductores**
Los algoritmos de traducción o traductores son los responsables de:

- Determinar si una lista de instrucciones es compatible con ellos.
- Recibiendo esa lista y un conjunto de bits generar una lista de instrucciones con esos bits codificados.
- Desde una lista de instrucciones extraer los bits ocultos.

## 4.2 Diseño detallado

En esta sección se observa con detalle el diseño de los bloques más significativos.

### 4.2.1. Reensamblador

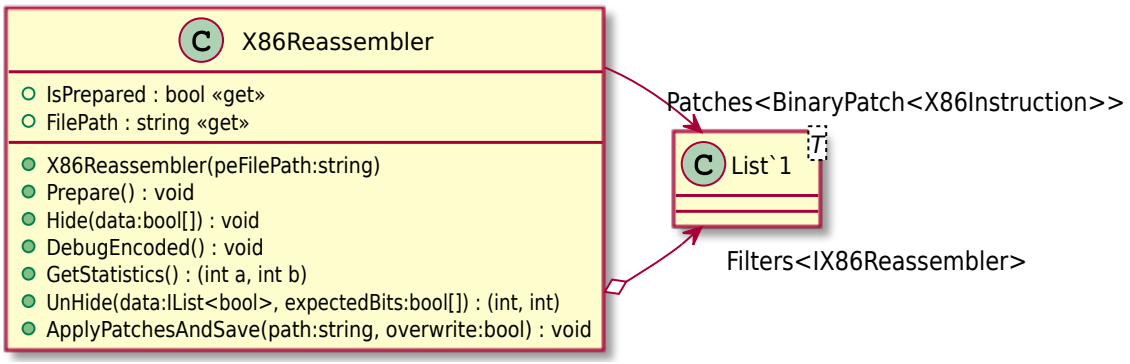


**Figura 4.3:** Diagrama del bloque reensamblador

| | |
|---|---|
| X86Reassembler | El constructor recibe la ruta del potencial fichero PE, comprueba que sea un fichero válido y obtiene la dirección del fichero donde está la sección ejecutable. |
| Prepare | Vuelca la sección de código a un array de bytes. |
| Hide | Recibe un listado de bits a ocultar en las instrucciones de la sección de código. Para ello decompila las instrucciones de la misma y va ejecutando cada uno de los traductores. Va manteniendo un listado de parches que debería aplicar sobre el fichero para almacenar los bits. |

DebugEncoded — Comprueba que los parches generados cumplen los requisitos de los algoritmos de traducción y muestra información sobre los mismos.

GetStatistics — Devuelve estadísticas sobre la ejecución del proceso de reensamblado.

UnHide — Devuelve un listado de bits obtenido de analizar la sección ejecutable del fichero.

ApplyPatchesAndSave — Aplica los parches generados durante *Hide* y los almacena en la ruta recibida.

### 4.2.2. Algoritmos de traducción

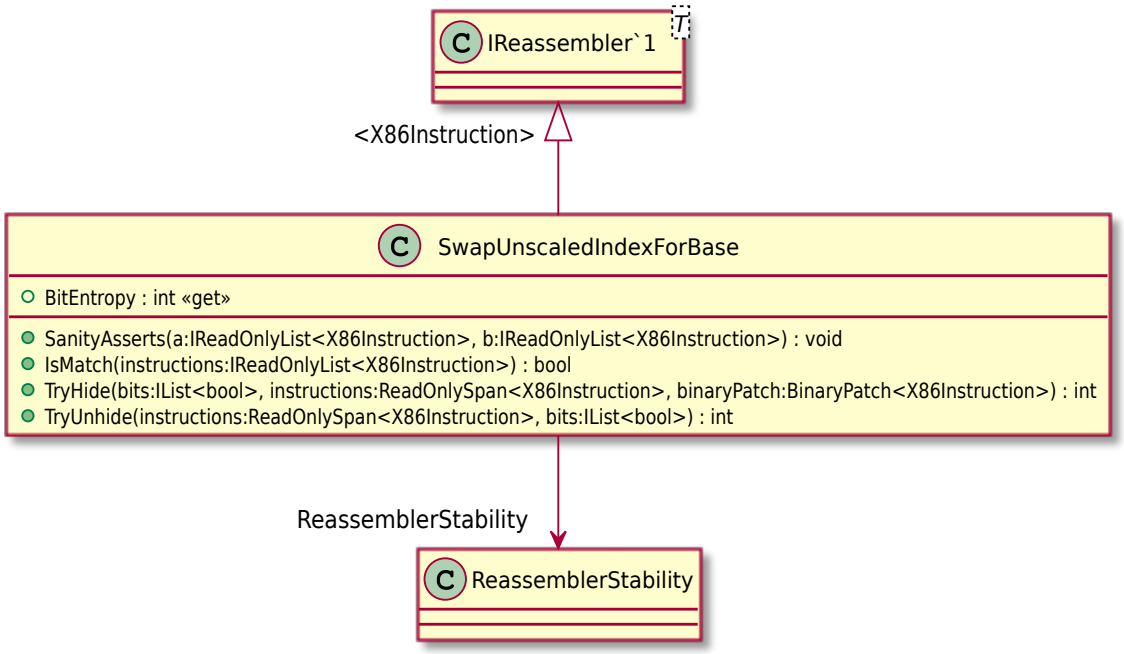


**Figura 4.4:** Diagrama del algoritmo de traducción *SwapUnscaledIndexForBase*

Los Algoritmos de traducción tienen dos propiedades que los caracterizan:

ReassemblerStability — La estabilidad de la técnica que utiliza. En algunos casos varios traductores pueden coincidir en *BitEntropy* y serían ordenados según su robustez.

BitEntropy — Cantidad de bits que pueden generar.

Descripción de las funciones públicas:

SanityAsserts — Comprueba que dos sets de instrucciones son válidos como entrada y salida del traductor.

IsMatch — Recibe un conjunto de instrucciones y determina si el algoritmo de traducción es capaz de procesarlas.

TryHide — Intenta ocultar tantos bits como *BitEntropy* en el listado de instrucciones. Devuelve la cantidad de instrucciones que ha consumido del listado y el parche que propone para almacenar los bits si ha sido capaz de procesar las instrucciones.

TryUnhide — Recibe un listado de instrucciones. Devuelve la cantidad que ha consumido y los bits que ha extraído de las mismas.

## 4.3 Tecnología utilizada

En esta sección se listarán los *frameworks* más relevantes utilizados en la implementación de la solución.

### 4.3.1. PENet

PENet [1] es librería multiplataforma que permite obtener la información necesaria de los ficheros PE. Es utilizado para comprobar que el binario es IA32 y obtener la sección de código.


**Figura 4.5:** Logotipo de PENet

Además de interpretar las estructuras de los ficheros PE y permitir accederlas desde objetos de C# es capaz de validar las firmas digitales de estos ficheros.

### 4.3.2. Capstone

Capstone [2] es un framework ligero, multiplataforma y multiarquitectura para el desensamblado de instrucciones. Es utilizado en el análisis de las instrucciones del binario. Su análisis detallado incluye información de bajo nivel sobre la codificación de las instrucciones.

**Figura 4.6:** Logotipo de Capstone

Se trata de un framework muy popular utilizado al menos en 480 proyectos de código abierto. Especialmente en proyectos para análisis de malware.

[1] http://secana.github.io/PeNet/
[2] https://www.capstone-engine.org/

### 4.3.3. Keystone

Keystone [3] es el framework análogo a Capstone pero en lugar de desensamblado se encarga de ensamblado. Algunos de los traductores se hacen servir de este framework para ensamblar nuevas instrucciones desde sus mnemónicos.

**Figura 4.7:** Logotipo de Keystone

Este framework fue fundado a través de una campaña de micromecenazgo [4]. Junto con Capstone y Unicorn [5] compone la trifecta ideal para trabajar con ensamblador en todos los niveles.

### 4.3.4. NaCl.Core

NaCl.Core [6] es una librería con primitivas criptográficas. Se utiliza para el cifrado del mensaje antes de ocultarlo en el binario.

[3] https://www.keystone-engine.org/
[4] https://igg.me/at/keystone/
[5] https://www.unicorn-engine.org/
[6] https://github.com/idaviddesmet/NaCl.Core

# CAPÍTULO 5
# Desarrollo de la solución propuesta

En las siguientes secciones se describirán las diferente soluciones implementadas en detalle y se analizarán sus limitaciones.

## 5.1 La estructura PE

Como se puede observar las estructuras del fichero PE de la figura 5.1 se trata de un formato de fichero con mucha información.

Usando PENet se obtienen fácilmente los campos del PE necesarias para la toma de decisiones. En el caso actual se comprueban los siguientes:

- `IMAGE_NT_HEADERS.FileHeader.Machine`: Filtramos por ejecutables de i386
- `IMAGE_NT_HEADERS.OptionalHeader.Magic`: Filtramos por ejecutables PE32, es decir, de 32 bits.
- `IMAGE_NT_HEADERS.OptionalHeader.AddressOfEntryPoint`: Dirección de la primera instrucción que se ejecutará una vez el ejecutable inicie su ejecución.
- `IMAGE_NT_HEADERS.FileHeader.NumberOfSections`: Cantidad de direcciones que contiene el ejecutable.
- `IMAGE_SECTION_HEADER.Characteristics`: Mapa de bits con las características de la sección. Especialmente relevante es el flag `IMAGE_SCN_MEM_EXECUTE`.

Además de estos campos se recorren las secciones del fichero en busca de la sección que contenga el `AddressOfEntryPoint`.

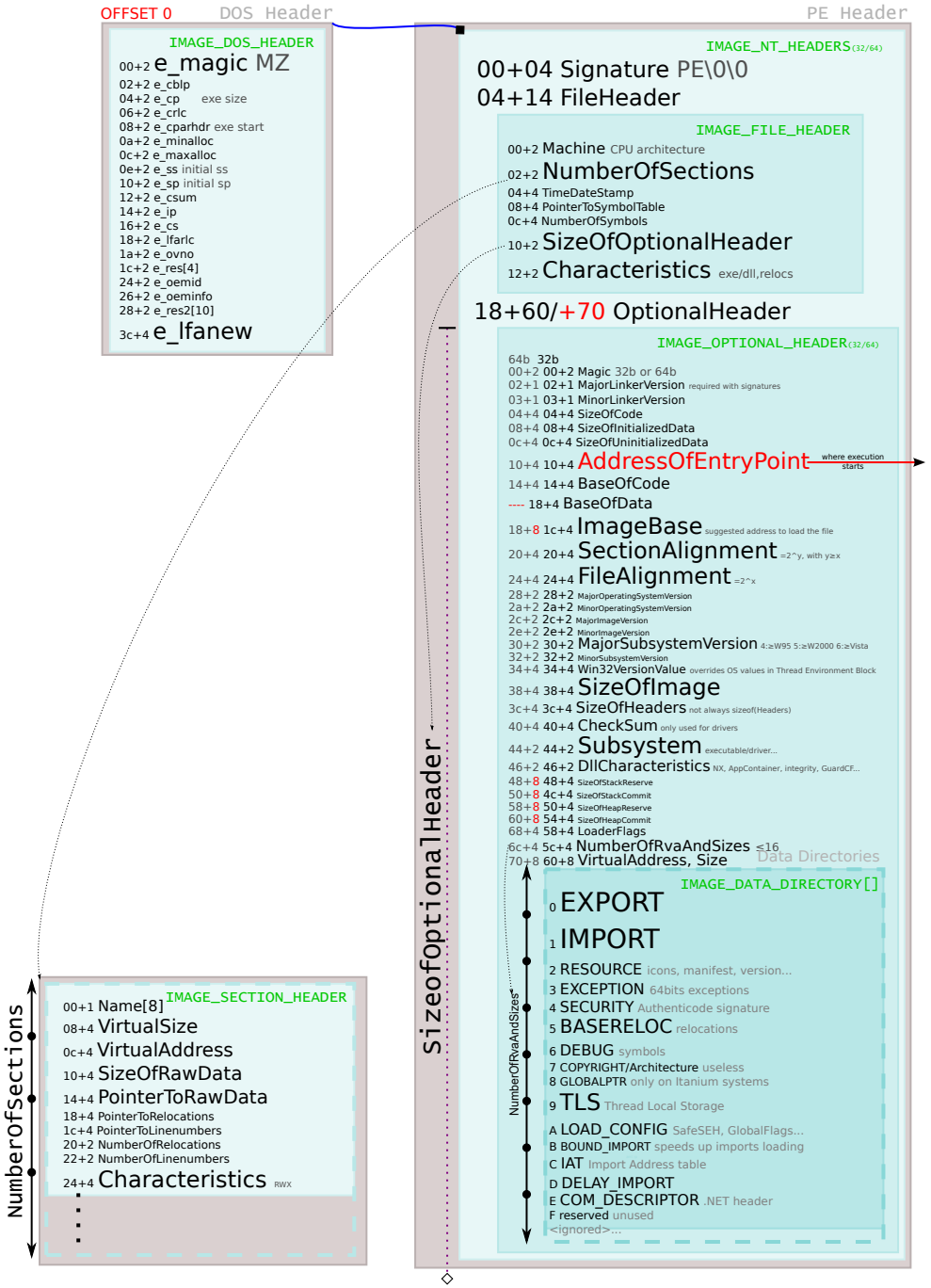


**Figura 5.1:** Diagrama de las estructuras PE relevantes.[1]

## 5.2 Desensamblar instrucciones IA32

En el proceso de compilación muchísima información sobre la estructura original del código se pierde. Sin esa información separar bloques funcionales y distinguirlos de datos se vuelve una tarea ardua. Algunos compiladores usan siempre los mismos preámbulos para las funciones o puede que el ejecutable lleve información para depuración. A pesar

[1] Adaptación del trabajo de Corkami https://github.com/corkami/pics/blob/master/binary/README.md

de todo esto el algoritmo que se utilice debe trabajar sin asumir nada sobre el ejecutable que analiza.

El algoritmo que se utiliza para el desensamblado de instrucciones es conocido como *linear sweep* [17]. Consiste en recorrer cada uno de los opcodes del bloque de código desde su principio hasta encontrar uno que no pueda ser desensamblado. Su implementación es sencilla pero tiene fuertes limitaciones que se analizarán en el siguiente punto.

### 5.2.1. Limitaciones

*Linear sweep* es poco eficaz con binarios que almacenan algún tipo de estructura de datos en su sección ejecutable. Debido a la naturaleza lineal del algoritmo éste debe parar al encontrarse con bytes que no corresponden con ningún opcode. Esto hace que posibles instrucciones válidas jamás llegan a alcanzarse para el análisis. Por otro lado también es posible que acabe interpretando como instrucciones válidas un conjunto de datos.

Este último caso es una de las principales problemáticas. El algoritmo puede identificar como instrucciones válidas un conjunto de bytes que son datos. Al aplicar los algoritmos de traducción se obtendrán nuevos bytes que a priori serían instrucciones equivalentes pero es muy probable que no sean datos equivalentes para la ejecución del ejecutable original. Esto acaba alterando el ejecutable de forma impredecible.

En la sección de Trabajos futuros se tratará la solución a estas limitaciones.

## 5.3 Estego-algoritmos

En la implementación asociada a este trabajo se han diseñado cinco este-algoritmos diferentes basados en las redundancias tratadas en el capítulo Análisis del problema.

Los estego-algoritmos de redundancia de codificación están basados en el trabajo de Rakan El-Khalil para Hydan [**?**].

En la tabla 5.1 puede verse un resumen de los algoritmos de traducción que se describen en detalle a continuación.

| Estego-algoritmo | Tipo | Bits |
|---|---|---|
| TestInmediate | Redundancia de codificación | 1 |
| SwapUnscaledIndexForBase | Redundancia de codificación | 1 |
| DirectionalBit | Redundancia de codificación | 1 |
| ZeroRegister | Redundancia semántica | 1 |
| MoveInmediateByteIntoDword | Redundancia semántica | 1 |

Tabla 5.1: Bits por estego-algoritmos

### 5.3.1. Redundancia de codificación

#### TestInmediate

Utiliza un opcode equivalente para la instrucción `test` no documentado en el manual de Intel sobre IA32. Sí que está documentado en el manual de AMD.

Este traductor es capaz de almacenar un bit de información en una instrucción `test` que use un inmediato sobre cualquier registro que no sea *EAX*/*AX*/*AH*/*AL*.

En la tabla 5.2 se puede observar como el segundo byte de las instrucciones es diferente. Se está codificando `test ebx, 0x5` que encaja en `test r/m32,imm32`[2] que es el opcode `0xF7 /0` ya que `ebx` es un *r* (registro) y `0x5` es un *imm32* (inmediato de 32 bits).

| Dirección | Opcode | Instrucción |
|---|---|---|
| 0x00000000 | F7 C3 05 00 00 00 | test ebx, 0x5 |
| 0x00000000 | F7 CB 05 00 00 00 | test ebx, 0x5 |

Tabla 5.2: Ejemplo de *TestInmediate*

Las instrucciones con operandos tienen un byte específico llamado *Mod-REG-R/M* que almacena qué registros o métodos de dirección utiliza la instrucción. En el ejemplo que está siendo codificado el byte es `0xC3` o `0xCB`. Si se analizan los bits correspondientes:

| Mod-REG-R/M | Mod | REG | R/M |
|---|---|---|---|
| 0xC3 | 0b11 | 0b000 | 0b011 |
| 0xCB | 0b11 | 0b001 | 0b011 |

Tabla 5.3: Codificación Mod-REG-R/M

Cuando el valor de *Mod* es `0b11` significa que *R/M* es un registro. En este caso es `0b011` (*EBX*). La diferencia de *REG* es la clave; ahí es donde se está almacenando el bit de información.

### SwapUnscaledIndexForBase

Dentro de las formas de codificación de la instrucción cuando el *Mod* de *Mod-REG-R/M* es diferente de `0b11` significa que la instrucción tiene un byte extra llamado *Scale Index Byte* (SIB) [3]. Este byte codifica direccionamientos del tipo `[base + index*scale + disp]` donde *base* e *index* son registros.

Cuando *scale* es `1` la *base* y el *index* son intercambiables. Este algoritmo de traducción guarda un bit de información en el orden de los registros cuando una instrucción usar el byte SIB. Si quiere guardar el bit a `1` hará que `base >index` y lo contrario para guardar un bit a `0`.

| Dirección | Opcode | Instrucción |
|---|---|---|
| 0x00000000 | 8B 04 0B | mov eax,DWORD PTR [ebx+ecx*1] |
| 0x00000000 | 8B 04 19 | mov eax,DWORD PTR [ecx+ebx*1] |

Tabla 5.4: Ejemplo de *SwapUnscaledIndexForBase*

Como se puede observar en la tabla 5.4 el tercer byte de la instrucción ha cambiado. Este byte es el *SIB*. Se observa en la siguiente tabla qué información se está codificando:

`0b001` es *ECX* y `0b011` es *EBX*. Puede observarse como en el proceso de reensamblado se ha intercambiado la *base* con el *index*.

Un detalle importante de este traductor es que no puede guardar información en toda las instrucciones que tengan un byte *SIB*. Hay ciertos requisitos que deben cumplir:

- **El índice y la base deben ser diferentes:** Evidentemente si ambos son idénticos no es posible distinguir un bit establecido a `1` o `0`.

[2] https://c9x.me/x86/html/file_module_x86_id_315.html
[3] http://www.c-jump.com/CIS77/CPU/x86/lecture.html#X77_0100_sib_byte_layout

| SIB | Scale | Index | Base |
|---|---|---|---|
| 0x0B | 0b00 | 0b001 | 0b011 |
| 0x19 | 0b00 | 0b011 | 0b001 |

Tabla 5.5: Codificación SIB

- **La base no puede ser ESP:** Dado que *ESP* no es un registro válido para hacer de índice no sería posible intercambiarlos y generar una instrucción válida.
- **EBP no puede ser índice si Mod = 0:** Si `Mod = 0` el valor de `0b101` (*EBP*) está reservado para indicar que se trata de direccionado exclusivamente con desplazamiento.

#### DirectionalBit

Debido a la forma en la que se codifica el direccionamiento de las instrucciones se puede observar que para el caso en el que los dos operandos de una instrucción son registros hay dos formas de codificar la misma instrucción.

Este algoritmo de traducción afecta a instrucciones *add, adc, and, cmp, mov, or, sbb, sub, xor* cuyos operandos son registros. Es capaz de almacenar un bit directamente en el bit de dirección del opcode.

Se observa en el ejemplo de la tabla 5.6 como cambia tanto el opcode de la instrucción como el byte *Mod-REG-R/M*. A continuación se analiza cómo la misma instrucción resulta de codificaciones tan diferentes.

| Dirección | Opcode | Instrucción |
|---|---|---|
| 0x00000000 | 39 D1 | cmp ecx,edx |
| 0x00000000 | 3B CA | cmp ecx,edx |

Tabla 5.6: Ejemplo de *DirectionalBit*

| Opcode | Bit | Mod | Reg | R/M |
|---|---|---|---|---|
| 39 D1 | 0 | 0b11 | 0b010 | 0b001 |
| 3B CA | 1 | 0b11 | 0b001 | 0b010 |

Tabla 5.7: Codificación DirectionalBit

El bit de dirección determina si *Reg* es la fuente o el destino de la operación. Si la intrucción solo usa registros ese bit es redundante ya que pueden intercambiarse *Reg* y *R/M*.

### 5.3.2. Redundancia semántica

#### ZeroRegister

Este algoritmo de traducción de redundancia semántica almacena un bit de información en instrucciones *xor* o *sub* que se aplican sobre el mismo registro, es decir, cuyo objetivo es dejar un registro a `0`.

Como se puede observar en la tabla 5.8 este traductor cambia la instrucción por completo. Se basa en la equivalencia semántica de poner a cero un registro. Otra posibilidad sería añadir al conjunto de equivalencia `mov r, 0` o `push 0; pop r`.

| Dirección | Opcode | Instrucción |
|---|---|---|
| 0x00000000 | 31 C0 | xor eax,eax |
| 0x00000000 | 29 C0 | sub eax,eax |

Tabla 5.8: Ejemplo de *ZeroRegister*

**MoveInmediateByteIntoDword**

Este traductor se aplica sobre instrucciones cuyo objetivo es cargar un valor inmediato de cuatro bytes que cabría en un byte. Particularmente se aplica sobre `mov r, imm32` y `push imm32; pop r`.

| Dirección | Opcode | Instrucción |
|---|---|---|
| 0x00000000 | BB 13 00 00 00 | mov ebx,0x13 |
| 0x00000000 | 6A 13 | push 0x13 |
| 0x00000002 | 5B | pop ebx |
| 0x00000003 | 66 90 | xchg ax,ax |

Tabla 5.9: Ejemplo de *MoveInmediateByteIntoDword*

Como se observa en el ejemplo de la tabla 5.9 se ha añadido un `nop` [4] de dos bytes. Esto es necesario para cumplir el requisito de mantener la cantidad de bytes después del reensamblado.

## 5.4 Ensamblado de instrucciones IA32

Algunos de los estego-algoritmos, o algoritmos de traducción, necesitan generar nuevas instrucciones para facilitar la comprensión en lugar de construir los opcodes a mano se utiliza Keystone para ensamblar los mnemónicos.

Cada uno de los algoritmos de traducción que ha funcionado exitosamente devuelve una estructura definiendo el parche que hay que aplicar sobre el ejecutable para almacenar los bits deseados. El mecanismo de reensamblado recorre estas estructuras y reemplaza los bytes originales por los bytes nuevos que codifican los bits necesarios.

### 5.4.1. Limitaciones

Existe una limitación considerable que los estego-algoritmos deben tener en cuenta. Dado que la IA32 tiene tamaños de instrucción variable podría darse el caso de que el resultado del estego-algoritmo no coincida en tamaño con la versión original. Esto genera dificultades añadidas:

- **No hay espacio suficiente:** puede darse el caso de que no haya espacio en el binario para encajar los nuevos opcodes.
- **Se está desplazando alguna referencia relativa:** es posible que una instrucción haga referencia a una dirección posterior está siendo desplazada.

Es por esta limitación que los traductores deben devolver los mismos bytes que consumen. Es decir, las instrucciones propuestas deben ocupar los mismos bytes que las instrucciones originales.

[4] https://en.wikipedia.org/wiki/NOP_(code)

La solución a estas dificultades está fuera del alcance de este trabajo y se analizará en el capítulo de Trabajos futuros.

## 5.5 Seguridad de la información

Para el cifrado de la información antes de almacenarla se utiliza un cifrado de flujo ChaCha20 [5]. Primero se computa el *hash* SHA256 [6] de la clave seleccionada por el usuario y ésta se utiliza para cifrar con ChaCha20.

[5] https://en.wikipedia.org/wiki/Salsa20#ChaCha_variant
[6] https://es.wikipedia.org/wiki/SHA-2

# CAPÍTULO 6
# Pruebas

Las técnicas descritas durante el trabajo se han llevado luego a la práctica sobre dos corpus de ejecutables para medir su efectividad. El objetivo del análisis es medir la cantidad de información que se puede albergar en un ejecutable medio.

## 6.1 Corpus

### 6.1.1. VirusShare

Se han extraído 8049 muestras (12GB) de una de las publicaciones de VirusShare [1]. Se trata de muestras de malware desarrolladas en múltiples lenguajes. Es una muestra algo sesgada ya que es más común en ciertas familias de malware implementar ofuscaciones para evitar análisis estáticos. Eso hace que el conjunto de instrucciones utilizadas no sea una muestra de lo que un compilador tradicional generaría.

### 6.1.2. SysWOW64

Se trata de la carpeta de librerías de IA32 del sistema de Windows x64. La muestra se ha extraído de una versión de Windows 10 (1809 17763.678).

Se han extraído 2312 muestras (1GB). Estas muestras son más representativas del resultado de un proceso de compilación tradicional puesto que están en su gran mayoría compiladas desde código en C++.

## 6.2 Estadísticas

Para la obtención de estadísticas se intentará esconder 1024 bits de información en todos los ejecutables del corpus.

### 6.2.1. VirusShare

De las 8049 muestras únicamente 2298 pudieron almacenar al menos 1024 bits. En la figura 6.1 puede observarse el histograma. Se extrae del histograma que un gran volumen de los ficheros no han podido albergar más de 50 bits. Esto es debido principalmente al mecanismo de desensamblado lineal que se deja instrucciones por analizar.

[1] https://virusshare.com/

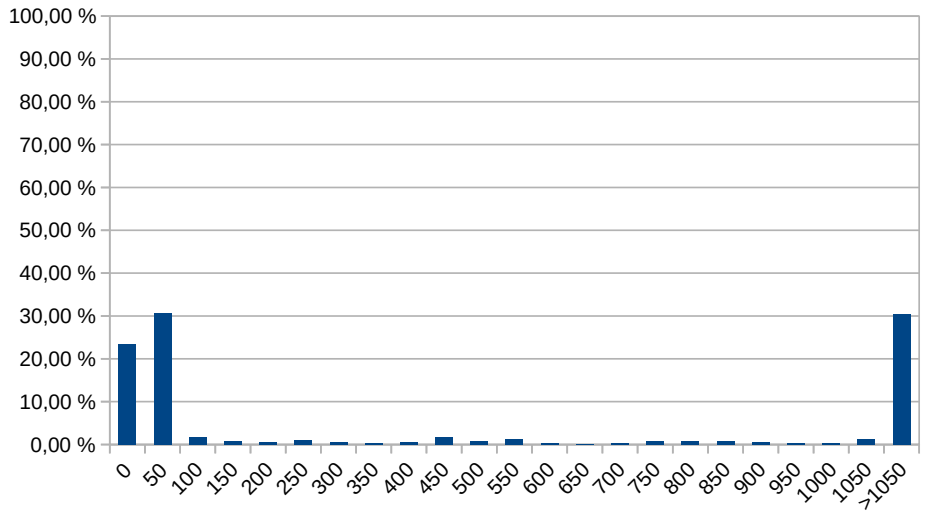


**Figura 6.1:** Histograma de bits por fichero en VirusShare

La figura 6.2 analiza la cantidad de instrucciones detectadas por byte de sección de instrucción. Ninguna instrucción IA32 puede ocupar más de 15 bytes. Viendo el histograma se revela que muchas instrucciones han pasado desapercibidas.

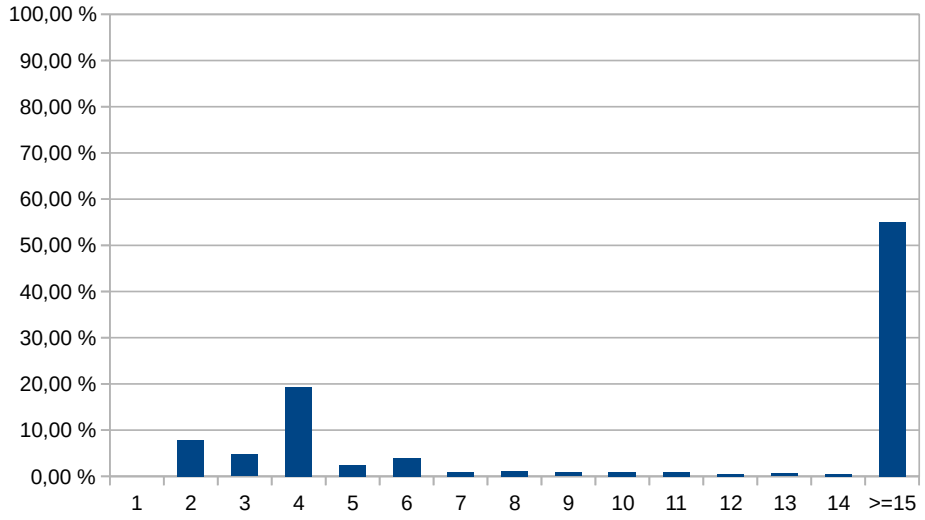


**Figura 6.2:** Histograma de instrucciones por byte en VirusShare

En la tabla 6.1 se analiza qué algoritmo de traducción es el más utilizado dentro de las instrucciones que acaban teniendo una traducción.

Puede extraerse de los datos que el algoritmo más utilizado es *DirectionalBit*, esto es algo previsible puesto que es el que más variedad de instrucciones puede traducir como se vió en la descripción de DirectionalBit.

Otra observación relevante es que *ZeroRegister* no se ha aplicado a ninguna de las traducciones hechas. Esto es porque colisiona con *DirectionalBit* y éste tiene prioridad. Es decir, los dos pueden afectar a las mismas instrucciones pero *DirectionalBit* es preguntado antes si puede usarla por lo que *ZeroRegister* jamás llega a aplicarse.

| Algoritmo de traducción | Instrucciones | Porcentaje |
|---|---|---|
| TestInmediate | 29582 | 1.03 % |
| SwapUnscaledIndexForBase | 116356 | 4.06 % |
| DirectionalBit | 2087648 | 72.92 % |
| ZeroRegister | 0 | 0 % |
| MoveInmediateByteIntoDword | 629363 | 21.98 % |
| Total | 2862949 | 100 % |

Tabla 6.1: Instrucciones traducidas por algoritmo de traducción en VirusShare

### 6.2.2. SysWOW64

Si se aplica el mismo análisis sobre este corpus se pueden observar resultados muy similares en la figura 6.3 y 6.4.

En la tabla 6.2 puede verse de nuevo que *DirectionalBit* es el responsable de traducir la mayoría de las instrucciones traducidas.

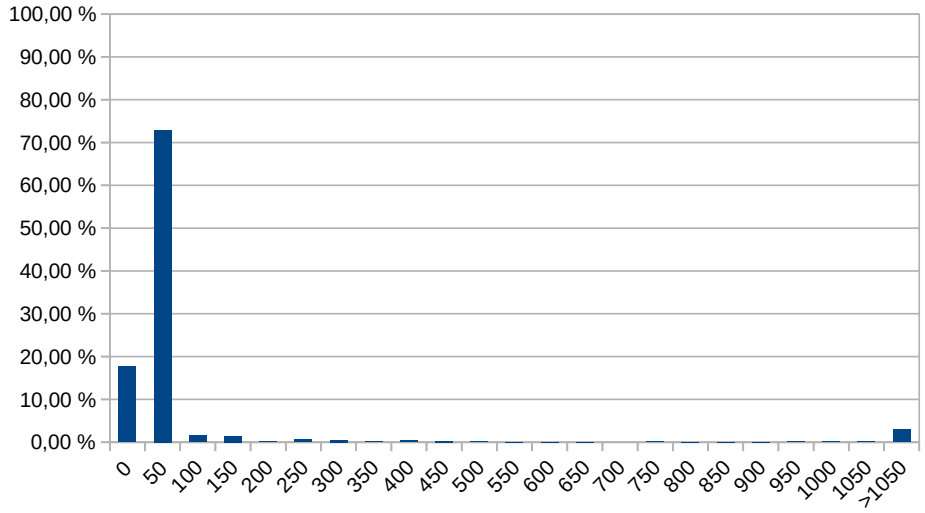


**Figura 6.3:** Histograma de bits por fichero en SysWOW64

| Algoritmo de traducción | Instrucciones | Porcentaje |
|---|---|---|
| TestInmediate | 860 | 0.78 % |
| SwapUnscaledIndexForBase | 5986 | 5.42 % |
| DirectionalBit | 91932 | 83.17 % |
| ZeroRegister | 0 | 0 % |
| MoveInmediateByteIntoDword | 11763 | 10.64 % |
| Total | 110541 | 100 % |

Tabla 6.2: Instrucciones traducidas por algoritmo de traducción en SysWOW64

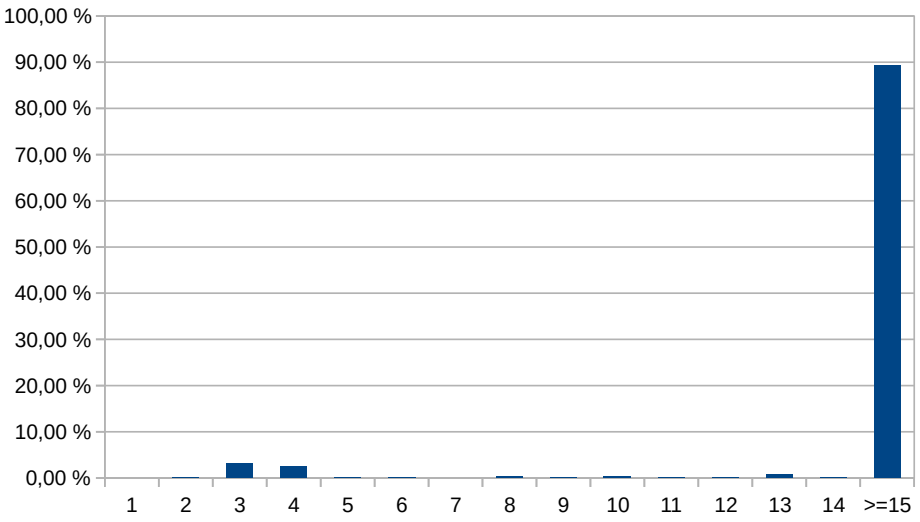


**Figura 6.4:** Histograma de instrucciones por byte en SysWOW64

# CAPÍTULO 7
# Conclusiones

Es momento de revisar si los objetivos planteados en la sección Objetivos han sido logrados.

Se ha diseñado e implementado una herramienta capaz de ocultar de forma segura un conjunto de bits en un fichero PE de Windows para IA32. Se ha diseñado permitiendo añadir nuevos traductores a la misma para así aumentar la cantidad de bits por instrucción que se pueden codificar.

Se ha analizado también la efectividad del trabajo realizado y se han detectado las deficiencias. Además se han planteado posibles soluciones a estas limitaciones.

Se concluye que tanto los objetivos principales como los secundarios han sido logrados. No dejando de lado los Trabajos futuros que permitirían cubrir estos objetivos con mayor profundidad.

## 7.1 Relación del trabajo desarrollado con los estudios cursados

Como se puede observar la base sobre la que se construye este trabajo es la de la arquitectura de computadores. Los mecanismos mediantes los cuales los procesadores comprenden las instrucciones y cómo éstas son codificadas desde mnemónicos. Durante los estudios cursados ha habido múltiples materias tratando en diferentes niveles de profundidad las bases sobre las que se sustenta el trabajo.

# CAPÍTULO 8
# Trabajos futuros

Durante el diseño y la implementación de la solución se han identificado mejoras o descubierto casos de uso que quedarían fuera del ámbito del trabajo. En este capítulo se tratarán con detalle estas cuestiones.

## 8.1 Limitaciones

### 8.1.1. Desensamblado

Como ya se ha mencionado el algoritmo *linear sweep* tiene limitaciones que hace que las instrucciones detectadas por byte de sección de código sean insuficientes en algunos casos. Tal y como ha sido corroborado en el capítulo de Pruebas

Se debería implementar un algoritmo recursivo de desensamblado cuyo objetivo es recorrer realmente las instrucciones que se acabarían ejecutando del ejecutable. El algoritmo escogería como primera instrucción la que es apuntada por el **AddressOfEntrypoint** del fichero PE y continuaría linealmente hasta encontrar una instrucción de cambio de flujo, como un `jmp` o un `call`, y entonces almacenaría la dirección para continuar su análisis en esas ramas de la ejecución.

Este algoritmo hace un recorrido exhaustivo de las instrucciones. Se trata de un algoritmo más complejo de implementar y es susceptible a errores. Dada la complejidad de la IA32 existen mecanismos para controlar el flujo fuera de las instrucciones tradicionales.

Construcciones como la de la tabla 8.1 alteran la pila de ejecución para retornar la ejecución a una dirección que no es referenciada de forma directa. Un algoritmo recursivo que no contemplase estos controles de flujo complejos podría dejar instrucciones sin revisar.

| Dirección | Opcode | Instrucción |
|---|---|---|
| 0x00000000 | E8 01 00 00 00 | call 0x6 |
| 0x00000005 | C3 | ret |
| 0x00000006 | 83 04 24 06 | add DWORD PTR [esp],0x6 |
| 0x0000000A | C3 | ret |

Tabla 8.1: Ejemplo de control de flujo complejo de emular

### 8.1.2. Ensamblado

Una de las mayores restricciones con las que tienen que trabajar los algoritmos de traducción es que la salida debe de ser del mismo tamaño que la entrada. Esto reduce las técnicas de redundancia semántica que se pueden ampliar y obliga a añadir instrucciones `NOP` para hacer de relleno.

Una mejora evidente sería permitir cualquier variabilidad en el tamaño. Los problemas que esto supone ya fueron descritos en la sección Limitaciones. La solución a estos problemas obliga a reensamblar prácticamente por completo el ejecutable. Habría que recorrer las instrucciones con referencias relativas y corregirlas si ha habido un cambio de tamaño en su zona de actuación.

La complejidad de esta implementación es similar a la de la sección previa. Detectar esas direcciones relativas es complejo ya que existen formas variadas de alterar el flujo de ejecución.

La conclusión a la que se llega después de encontrar estos problemas es que la emulación u otras técnicas de análisis estático avanzado son necesarias.

## 8.2 Nuevas funcionalidades

### 8.2.1. Más arquitecturas

La solución podría fácilmente ampliarse para soportar diferentes reensambladores según la arquitectura del ejecutable.

### 8.2.2. Más ejecutables

Actualmente solo existe soporte para los ficheros PE pero existen otros contenedores con instrucciones IA32 para otros sistemas operativos.

### 8.2.3. Reordenado de instrucciones

Una forma de ampliar los bits por instrucción podría ser la capacidad de ordenar las instrucciones. Tener la capacidad de reordenar instrucciones obliga a hacer un análisis más profundo de toda la ejecución ya que los efectos de una instrucción pueden ser comprobados cientos de instrucciones después. Se debe hacer un análisis de dependencias para evitar que el reordenado cambie el resultado de la ejecución.

### 8.2.4. Detección de equivalencias automática

El resto de nuevas funcionalidades dejan entrever que la ejecución estática de las instrucciones podría revelar información extra que permitiría codificar más información. Una vez esa capacidad está implementada pueden abstraerse las instrucciones que participan y centrarse en los resultados.

Habría un proceso de decompilación mediante el cual las instrucciones se reinterpretan a un lenguaje intermedio que luego volvería a ser compilado a ensamblador añadiendo los bits correspondientes.

### 8.2.5. Trabajar a nivel de compilación

Una posibilidad es aplicar estos mismos algoritmos como un paso extra en la compilación de los ejecutables. En lugar de tratar de reensamblar un ejecutable ya finalizado, se podrían aplicar los algoritmos de traducción dentro del *pipeline* LLVM [1]. Esto daría visibilidad sobre estructuras más complejas del código que sería complejo de extraer del ejecutable resultante como se ha tratado en las secciones previas.

[1] https://llvm.org/

# Bibliografía


[1] Shahid Alam, R.Nigel Horspool, Issa Traore, and Ibrahim Sogukpinar. A framework for metamorphic malware analysis and real-time detection. *Computers & Security*, 48(C):212–233, 2015.

[2] M Ben Khadra, Dominik Stoffel, and Wolfgang Kunz. Speculative disassembly of binary code. In *Proceedings of the International Conference on compilers, architectures and synthesis for embedded systems*, CASES '16, pages 1–10. ACM, 2016.

[3] Jorge Blasco Alís. Information leakage and steganography: detecting and blocking covert channels, 2012.

[4] Yang-Seo Choi, Ik-Kyun Kim, Jin-Tae Oh, and Jae-Cheol Ryou. Pe file header analysis-based packed pe file detection technique (phad). In *International Symposium on Computer Science and its Applications*, pages 28–31. IEEE, 2008.

[5] S.P. Choudhary and Miss Deepti Vidyarthi. A simple method for detection of metamorphic malware using dynamic analysis and text mining. *Procedia Computer Science*, 54(C):265–270, 2015.

[6] Christian Collberg. *Surreptitious Software: Obfuscation, Watermarking, and Tamperproofing for Software Protection*. Addison-Wesley Professional.

[7] Rakan El-Khalil and Angelos D. Keromytis. Hydan: Hiding information in program binaries, 2004.

[8] Peter Kankowski. Redundancy of x86 machine code - special encoding for eax and byte immediates. https://www.strchr.com/machine_code_redundancy.

[9] Alireza Khalilian, Amir Nourazar, Mojtaba Vahidi-Asl, and Hassan Haghighi. G3md: Mining frequent opcode sub-graphs for metamorphic malware detection of existing families. *Expert Systems With Applications*, 112:15–33, 2018.

[10] Ajit Kumar and G Aghila. Portable executable scoring: What is your malicious score? In *2014 International Conference on Science Engineering and Management Research (ICSEMR)*, pages 1–5. IEEE, 2014.

[11] Xufang Li, P. K. K Loh, and F Tan. Mechanisms of polymorphic and metamorphic viruses. In *2011 European Intelligence and Security Informatics Conference*, pages 149–154. IEEE, 2011.

[12] Chen Liang, Yu-an Tan, Xiaosong Zhang, Xianmin Wang, Jun Zheng, and Quanxin Zhang. Building packet length covert channel over mobile voip traffics. *Journal of Network and Computer Applications*, 118:144–153, 2018.

[13] Philip O'kane, Sakir Sezer, and Kieran McLaughlin. Detecting obfuscated malware using reduced opcode set and optimised runtime trace. *Security Informatics*, 5(1):1–12, 2016.

[14] Jordi Serra. Esteganografía y estegoanálisis : técnicas para crear y descubrir covert channels, 2014.

[15] P.V. Shijo and A. Salim. Integrated static and dynamic analysis for malware detection. *Procedia Computer Science*, 46:804–811, 2015.

[16] C+ Visual and Business Unit. Microsoft portable executable and common object file format specification, 1999.

[17] Donghong Zhang, Zhenyu Zhang, Bo Jiang, and T.H Tse. The impact of lightweight disassembler on malware detection: An empirical study. In *2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC)*, volume 1, pages 620–629. IEEE, 2018.

# APÉNDICE A
# Glosario

C | E | F | M | O | P

## C

**canal encubierto**

Un canal encubierto, es un canal que puede ser usado para transferir información desde un usuario de un sistema a otro, usando medios no destinados para este propósito por los desarrolladores del sistema. Para que la comunicación sea posible suele ser necesario un preacuerdo entre el emisor y el receptor que codifique el mensaje de una forma que el receptor sea capaz de interpretar. 1

**compilador**

Un compilador es un tipo de traductor que transforma un programa entero de un lenguaje de programación (llamado código fuente) a otro. Usualmente el lenguaje objetivo es código máquina. 1

**conjunto de instrucciones**

Un conjunto de instrucciones o ISA (del inglés *instruction set architecture*),es una especificación que detalla las instrucciones que una unidad central de procesamiento puede entender y ejecutar, o el conjunto de todos los comandos implementados por un diseño particular de una CPU. El término describe los aspectos del procesador generalmente visibles para un programador, incluyendo los tipos de datos nativos, las instrucciones, los registros, la arquitectura de memoria y las interrupciones, entre otros aspectos. 2, 39

**contador de programa**

El contador de programa (en inglés Program Counter o PC), es un registro del procesador de un computador que indica la posición donde está el procesador en su secuencia de instrucciones. 8, 39

## E

**estego-algoritmo**

Estego-algoritmo es el algoritmo esteganográfico que indica cómo realizar el procedimiento de incorporación del mensaje que queremos mantener en secreto en el portador. 9, 10, 22

## F

**flag**

En programación, la bandera o flag se refiere a uno o más bits que se utilizan para almacenar un valor binario o código que tiene asignado un significado. 8, 17

**framework**

En el desarrollo de software, un *framework* o entorno de trabajo es una estructura conceptual y tecnológica de asistencia definida, normalmente, con artefactos o módulos concretos de software, que puede servir de base para la organización y desarrollo de software. 5, 15, 16

**M**

**malware**

El término malware es utilizado para referirse a una variedad de software hostil, intrusivo o molesto. 5, 8, 15, 25

**mnemónico**

Un mnemónico es una palabra que sustituye a un opcode (lenguaje de máquina), con lo cual resulta más fácil la programación. 1, 16, 22, 29

**O**

**opcode**

Opcode (operation code) o código de operación, es la porción de una instrucción de lenguaje de máquina que especifica la operación a ser realizada. 1, 8, 9, 19–22, 38

**P**

**Portable Executable**

El formato Portable Executable es un formato de archivo para archivos ejecutables, de código objeto, bibliotecas de enlace dinámico (DLL), archivos de fuentes FON, y otros usados en versiones de 32 bit y 64 bit del sistema operativo Microsoft Windows. 2, 7, 39

# APÉNDICE B
# Siglas

C | I | O | P | R

**C**

**CISC**

Computador con Conjunto de Instrucciones Complejas 1

**I**

**IA32**

Intel Architecture, 32-bit 1, 2, 8, 15, 19, 22, 25, 26, 29, 31, 32

**ISAs**

conjuntos de instrucciones 2

**O**

**OOP**

Programación Orientada a Objetos 1

**P**

**PC**

contador de programa 8

**PE**

Portable Executable 2, 7, 13, 15, 17, 29, 31, 32

**R**

**RISC**

Computador con Conjunto de Instrucciones Reducidas 1

# APÉNDICE C
# Código fuente

Puedes obtener la última versión del código fuente de la implementación del trabajo en el siguiente repositorio:

https://carlos.mele.ro/git/carlos.melero/TFG