Universidad de Deusto

Nuevo Enfoque para la Detección de Malware basado en Métodos de Recuperación de Información

Igor Santos Grueiro

Bilbao, junio de 2011

Universidad de Deusto

Nuevo Enfoque para la Detección de Malware basado en Métodos de Recuperación de Información

Tesis doctoral presentada por Igor Santos Grueiro
dentro del Programa de Doctorado en
Sistemas de Información

Dirigida por el Dr. Pablo García Bringas

El doctorando El director

Bilbao, junio de 2011

A Tamara.
Eres y serás siempre
la luz que ilumina
mi camino en la vida.

Resumen

Malware es todo código malicioso que tiene potencial para dañar cualquier ordenador, red o información. La cantidad de malware se está incrementado cada año y constituye un serio problema de seguridad. Por lo tanto, la detección de malware se ha convertido en un tema crítico dentro del campo de la seguridad de la información.

Actualmente, el método más extendido dentro de las compañías antivirus es la detección basada en firmas. A pesar de que esta técnica aún se utiliza en la mayoría de soluciones antivirus comerciales, sólo es capaz de conseguir la detección una vez que el ejecutable ha causado daño y están documentados. Además los creadores de software malicioso utilizan técnicas de ofuscación para ocultar sus creaciones a estos detectores de malware. Por ello, los sistemas de detección de malware simplemente fallan al detectar, tanto nuevas variantes de malware conocido como familias de malware desconocidas.

En esta tesis doctoral proponemos un nuevo sistema de representación de ejecutables basado en *recuperación de la información* que es capaz de detectar variantes ofuscadas de malware y detectar malware desconocido, reduciendo los esfuerzos del etiquetado mientras mantiene un alto nivel de precisión. En concreto, utilizamos la frecuencia de aparición de secuencias de códigos operacionales (*opcodes*). En base a esta representación, hemos unificado la detección de variantes ofuscadas de familias de malware con la detección de malware desconocido.

Para la detección de variantes, hemos propuesto un sistema completo de *recuperación de variantes* que se compone de: (i) una base de datos de malware, (ii) una representación de ejecutables, (iii) un sistema para generar consultas que transforma un ejecutable en su representación y (iv) un generador de modelos filogenéticos para monitorizar la evolución de una familia de malware.

Para la detección de malware desconocido, hemos usado *aprendizaje automático*. Hemos empleado *aprendizaje supervisado* que, aunque requiere una gran cantidad de software etiquetado en clases (*benigno* o *malware*), es capaz de ofrecer un alto nivel de precisión. Para reducir el elevado número de ejecutables necesario en el entrenamiento propio del aprendizaje automático, hemos adaptado un enfoque de *aprendizaje de clase única*, en el que sólo es necesario etiquetar un subconjunto de especímenes de una de las clases de software.

Abstract

Malware is every malicious code that has the potential to harm any computer, network or information. The amount of malware is increasing faster every year and poses as a serious security threat. Hence, malware detection has become a critical topic in computer security.

Currently, signature-based detection is the most extended technique within commercial antivirus. Although this method is still used on most popular commercial antivirus software, it can only achieve detection once the virus has already caused damage and it has been registered. Further, malware writers use code obfuscation techniques to hide their executables from these malware detection systems. Therefore, signature-methods fail to detect new variants of known malware families and unknown malware.

In this dissertation, we propose a new method for the representation of executables that is based on *information retrieval* able to detect malware variants and unknown malware, reducing the effots of labelling executables while it maintains a high level of accuracy. In particular, we utilise the frequency of appearance of sequences of operational codes (*opcodes*). Through this representation, we unified both malware variant and unknown malware detection.

Regarding the variant detector, we propose a full *variant retrieval* system which consists of: (i) a malware database; (ii) a representation of executables; (iii) a system able to generate queries, which transforms an executable to the proposed representation; and (iv) a phylogeny model generator to monitor the evolution of malware families.

We applied *machine learning* to detect unknown malware. On one hand, we have employed *supervised machine learning* that,

although it requires a high amount of labelled executables (i.e., previously classified as malware or benign software), it is very accurate in the detection. To reduce the number of executables required for the training phase of the models, we adapted *single-class learning*, which only requires labelling the specimens within one class.

Laburpena

Malware-a ordenagailuei, sareei edo informazioei kalte egin dezakeen edozein ordenagailuko programa da. Urtero, malwareen kopurua handitzen ari da eta honek, segurtasun arazo larriak dakartza. Honi aurre egiteko, informazioaren segurtasun arloan, helburu kritikoa bihurtu da malware-a egoki detektatzea.

Gaur egun, munduko segurtasun enpresa gehienek sinadura oinarritutako teknika erabiltzen dute. Teknika honen desabantaila handiena ezagutzen dituen kode maltzurrak bakarrik detektatzen dituela da. Gainera, malware sortzen duen pertsonek, kodea nahasteko teknikak erabiltzen dituzte jatorrizko intentzioak ezkutatzeko. Hau dela eta, detekzio teknika hauek huts egiten dute malware ezaguneko aldaerekin eta malware ezezagunekin.

Tesi honetan, ordenagailuko programen errepresentazioa proposatzen dugu *informazioa berreskuratze*ko tekniketan oinarrituta malware-en aldaerak eta malware ezezaguna detektatzen duena. Metodo honek esfotzua murrizten du, zehaztasun maila mantenduz. Hain zuzen ere, kode operazionaletako («opcode»-ak) sekuentzien maiztasuna erabiltzen dugu. Errepresentazio honetan oinarrituta, malware-en aldaerak eta malware ezezaguna detektatzeko metodo berriak garatu ditugu.

Malware-en aldaerari dagokionez, aztertzen ari den programa baten antzeko mutazioak berreskuratzen dituen sistema bat proposatu dugu. Sistema honek ondoko osagaiak ditu: malwarea biltzen duen datu-base bat; programak irudikatzeko sistema bat; programa bat kontsulta batean transformatzen duen teknika bat; eta malware-eko familia baten jarraipena erakusten duten modelo filogenetikoak egiteko sistema bat.

Beste alde batetik, *ikaskuntza automatiko*ko teknikak erabili ditugu malware ezezaguna detektatzeko. Lehendabizi, *ikaskuntza gainbegiratua* erabili dugu. Teknika hau erabiltzeko software

guztia behar bezala etiketaturik egon behar luke baina gure esperimentuen arabera, bere zehaztasuna handia da. Software etiketatuaren zenbatekoa murrizteko, *mota bakarreko ikaskuntza*ko metodoa egokitu dugu.

Agradecimientos

Esta tesis doctoral no se habría conseguido si no hubiera contado con un grupo de personas con tanta calidad humana y técnica a mi lado. Esta sección está dedicada a todas esas personas o entidades, sin las cuales este camino no habría sido posible de recorrer.

Me gustaría darle mis más sentidos agradecimientos a mi director de tesis, el Dr. Pablo García Bringas. Pablo me consiguió un trabajo en el S^{3}lab (*Laboratory for Smartness, Semantics and Security*) cuando yo todavía estaba en el último curso de Ingeniería Informática. Gracias a ello, comencé a investigar en técnicas para la detección de malware, despertándose mis inquietudes científicas, que sus valiosos comentarios y su capacidad de abstracción terminaron por forjar definitivamente. *Muchas gracias por todo, Pablo*.

De entre todas las personas que deberían aparecer en esta sección, tanto Félix Brezo Fernández como Xabier Ugarte Pedrero son las que más han contribuido a completar esta investigación. Tengo que agradecerles su ayuda tanto en la implementación de los diferentes módulos del prototipo resultante de esta investigación, como en sus interesantes revisiones a este manuscrito. Ambos sois la prueba fehaciente de que el buen juicio no tiene porqué estar reñido con la edad. *Gracias, Félix y Xabi!*

No puedo olvidarme de la gente del S^{3}lab, especialmente de Javier Nieves Acedo, de Borja Sanz Urquijo y de Carlos Laorden Gómez, que me han provisto de interesantes y fructíferos debates que, sin duda, han influido muy positivamente en esta tesis doctoral. Jamás olvidaré todos esos momentos, tantos los buenos como los malos, que hemos pasado juntos trabajando en tantos y tantos proyectos de investigación. Pocas veces he tenido la enorme suerte de estar acompañado de gente con tanto talento y creatividad. Entre todos vosotros habéis creado un entorno de

trabajo y de verdadera amistad en el que me siento como en casa y en el que espero poder seguir estando por muchos años. *Gracias, Copi, Booja y Lentejas. El espíritu del Brownie Team siempre seguirá vivo allá donde estemos.*

Me gustaría también expresar particularmente mis más sentidos agradecimientos a Mikel Salazar González por los diseños gráficos de este documento. Eres un artista, y así se ve reflejado en cada una de las figuras que haces. *Gracias, System.* De igual modo, agradezco a Iker Pastor López su ayuda en la configuración del experimento del algoritmo de clase única. *Gracias, Iker.*

He de agradecer al Ministerio de Industria (CENIT-CDTI) de España por financiar parte de esta investigación mediante el proyecto *Segur@: Seguridad y Confianza en la Sociedad de la Información*. Esta financiación me ha servido para colaborar con dos grandes empresas de seguridad como son Panda Security y Optenet. De Panda, me gustaría agradecer al personal de Panda Research, en particular, a Zaira Vicente Adame, Etor Llona Llamosas, Luis Regel Fernández y Kenneth Lobato Lastra, por sus interesantes comentarios. De Optenet, al Dr. José María Gómez-Hidalgo, por sus valiosos comentarios acerca de esta y otras investigaciones.

Me gustaría agradecer a mi familia más cercana su comprensión y su apoyo incondicional; ¡qué valiosos son todos esos momentos que hemos pasado todos juntos y que seguiremos pasando en el futuro! A mi padre, porque a pesar de nuestras diferencias de opinión, es una fuente inagotable de sabiduría y experiencia, en forma de útiles consejos que, en alguna ocasión, he tenido a bien seguir. A mi madre, que siempre ha estado a mi lado, proveyéndome de su opinión en los aspectos que realmente importan en la vida. A mi hermana Nuria, que siempre ha mostrado interés por lo que estaba haciendo y porque me enseñó lo que era estudiar informática. *Gracias, Pepe, Chelo, y Nuria.*

Igualmente, me gustaría agradecer a mi familia política su inmutable apoyo y el interés que han mostrado por el transcurso de este periodo de mi vida. *Gracias Mari, Aurelio, Mari Carmen, Rozas e Iri.*

Dejando lo mejor para el final, me gustaría expresar mi gratitud

a Tamara Alonso Rozas por su completa y total comprensión, y sus revisiones en busca de las erratas de este manuscrito y, en el aspecto personal, por enseñarme cada día lo que es el amor incondicional, permaneciendo a mi lado pese a mis numerosos defectos que, sin duda, también pueden ser objeto de estudio e investigación. Esta tesis va dedicada a su persona.

¡Gracias a todo el mundo!

Índice general

Índice de figuras **xvii**

Índice de tablas **xxi**

1 Introducción **3**
1.1 Malware, una amenaza creciente 4
1.1.1 Antes de que el malware ni siquiera existiese 4
1.1.2 La primera oleada de amenazas 4
1.1.3 El malware comienza a evolucionar 5
1.1.4 La profesionalización del malware 6
1.2 Retos a superar 7
1.3 Hipótesis y objetivos 9
1.4 Arquitectura global de la solución 11
1.5 Metodología de investigación 12
1.6 Estructura del documento 15

2 Estudio del estado del arte **17**
2.1 Conceptos y definiciones 18
2.1.1 El ecosistema del malware 18
2.1.2 Ofuscación de software 21
2.1.2.1 Conceptos básicos 22
2.1.2.2 Métodos de ofuscación de malware 24
2.1.2.3 Metamorfismo y polimorfismo 26
2.2 Detección de malware 27
2.2.1 Clasificación general de detectores de malware 28
2.2.2 Los métodos basados en firmas 29
2.2.3 Clasificación de técnicas de detección de malware . . 31
2.2.3.1 Clasificación de Idika et ál. 31

2.2.3.2 Clasificación en base al objetivo del detector 32
2.3 Métodos para la detección de variantes basados en el conocimiento . . . 34
2.3.1 Métodos estáticos para la detección de variantes . . . 34
2.3.1.1 Comparación de patrones . . . 34
2.3.1.2 Llamadas a la API o al sistema para la detección de malware . . . 35
2.3.1.3 Análisis del flujo de control . . . 36
2.3.2 Métodos dinámicos para la detección de variantes . . . 38
2.3.3 Métodos híbridos para la detección de variantes . . . 40
2.3.4 Resumen de los métodos para la detección de variantes 40
2.4 Métodos para la detección de malware desconocido . . . 41
2.4.1 Métodos basados en anomalías . . . 41
2.4.1.1 Detección estática de malware basada en anomalías . . . 41
2.4.1.2 Detección dinámica de malware basada en anomalías . . . 43
2.4.1.3 Detección híbrida de malware basada en anomalías . . . 45
2.4.2 Métodos basados en aprendizaje automático . . . 45
2.4.3 Resumen de los métodos para la detección de malware desconocido . . . 47
2.5 Aspectos relacionados con la detección de malware . . . 48
2.5.1 Formalización teórica . . . 48
2.5.2 Desempaquetado de ejecutables . . . 49
2.5.3 Generación automática de firmas . . . 52
2.6 Sumario . . . 53

3 Una representación de ejecutables 55
3.1 Capacidades de los *opcodes* para la detección de malware . . . 56
3.1.1 Descripción del lenguaje máquina . . . 56
3.1.2 *Opcodes* como características relevantes para la detección de malware . . . 59
3.2 Representación de ejecutables . . . 63
3.2.1 Definición formal de recuperación de información . . . 64
3.2.2 Modelo espacio vectorial . . . 64
3.2.3 Encontrando la relevancia de los *opcodes* . . . 67
3.2.4 Aplicación del VSM para representar ejecutables . . . 69
3.3 Evaluación . . . 71

3.4 Discusión de los resultados 76
3.5 Sumario 77

4 Detección de variantes de malware 79
4.1 Requerimientos 80
4.2 Descripción del método 81
4.2.1 Visión general 81
4.2.2 Base de datos de malware 82
4.2.3 Generador de consultas 84
4.2.4 Recuperación de variantes de malware 85
4.2.5 Generador de modelos filogenéticos 85
4.3 Evaluación empírica 88
4.3.1 Metodología general 88
4.3.2 Similitud de variantes 89
4.3.3 Detección de variantes 105
4.3.4 Modelos filogenéticos 111
4.4 Discusión de los resultados 117
4.5 Sumario 119

5 Detección de malware desconocido 121
5.1 Aprendizaje supervisado 122
5.1.1 Árboles de decisión 123
5.1.1.1 Formalización 123
5.1.1.2 Algoritmo C4.5 124
5.1.1.3 El método *random forest* 126
5.1.2 El método *Support Vector Machines* (SVM) 127
5.1.2.1 Formalización 127
5.1.2.2 Funciones de *kernel* 130
5.1.3 El algoritmo *K-Nearest Neighbours* (KNN) 132
5.1.4 Redes bayesianas 133
5.1.4.1 Formalización 134
5.1.4.2 Método de aprendizaje *hill climbing* 134
5.1.4.3 Algoritmo de aprendizaje K2 135
5.1.4.4 Clasificador *Naïve Bayes* 136
5.1.4.5 El método *Tree Augmented Naïve* (TAN) . . . 137
5.2 Aprendizaje de clase única 138
5.3 Validación empírica 141
5.3.1 Metodología general 141

5.3.2 Aprendizaje supervisado para la detección de malware desconocido . . . 143
5.3.3 Aprendizaje de clase única para la detección de malware desconocido . . . 148
5.4 Discusión de los resultados . . . 151
5.5 Sumario . . . 153

6 Conclusiones **155**
6.1 Síntesis de la validación del sistema . . . 155
6.2 Resumen de los resultados . . . 157
6.3 Aplicaciones de la investigación . . . 160
6.4 Limitaciones . . . 162
6.5 Trabajo futuro . . . 162
6.5.1 Mejora en la representación de ejecutables . . . 162
6.5.2 Ampliación del sistema de detección . . . 164
6.5.3 Defender los medios actuales de propagación de malware . . . 164
6.6 Consideraciones finales . . . 165

Bibliografía **167**

Apéndices **193**

A Desarrollo del prototipo de detección de malware NOA **195**
A.1 *Framework* de representación . . . 196
A.2 Desarrollo de la base de datos . . . 196
A.3 Diseño de la aplicación . . . 203
A.4 Interfaz gráfica . . . 208

B Fichero de relevancias de códigos operacionales **213**

C Muestras empleadas para la validación del detector de variantes **225**

Índice alfabético **229**

Índice de figuras

1.1 Crecimiento del nuevo malware en los últimos años. 7
1.2 Arquitectura de la solución planteada. 11

2.1 Código original del virus Chernobyl/CIH. 29
2.2 Firma para Chernobyl/CIH virus. 30
2.3 Los métodos de firmas no son capaces de detectar variantes de malware. 30
2.4 Clasificación de Idika et ál. de las técnicas de detección de malware. 31
2.5 Nuestra clasificación de técnicas de detección de malware. . . 33
2.6 Virus *contradictorio*. 49

3.1 Extracto de código máquina desensamblado. 57
3.2 Distribución de *opcodes* para el conjunto de software benigno. 60
3.3 Distribución de *opcodes* para el conjunto de malware. 61
3.4 Ejemplo de la aplicación de la *similitud del coseno* en un espacio bidimensional. 66
3.5 Métodología para obtener la relevancia de los *opcodes*. 67
3.6 Ejemplo de código máquina desensamblado. 70
3.7 Gráfico de dispersión de tiempos para nuestro método de representación con una longitud de secuencia $n = 1$ y las funciones de resumen básicas. 73
3.8 Gráfico de dispersión de tiempos para nuestro método de representación con una longitud de secuencia $n = 2$ y las funciones de resumen básicas. 73
3.9 Gráfico de dispersión de tiempo para nuestro método de representación con una longitud de secuencia $n = 3$ y las funciones de resumen básicas. 74

3.10 Gráfico de dispersión de tiempo para nuestro método de representación con una longitud de secuencia $n = 4$ y las funciones de resumen básicas. 74
3.11 Gráfico de dispersión de tiempos para nuestro método de representación con una longitud de secuencia $n = 5$ y las funciones de resumen básicas. 75
3.12 Evolución del rendimiento dependiendo del tamaño de la secuencia de *opcodes*. 76

4.1 Arquitectura general de nuestro método para la detección de variantes. 81
4.2 Modelo de datos propuesto. 83
4.3 Histograma de similitudes para un tamaño de secuencia de 1 para variantes de malware. 90
4.4 Histograma de similitudes para un tamaño de secuencia de 1 cuando se compara con software benigno. 90
4.5 Histograma de similitudes para un tamaño de secuencia de 2 para variantes de malware. 91
4.6 Histograma de similitudes para un tamaño de secuencia de 2 cuando se compara con software benigno. 93
4.7 Histograma de similitudes para una representación combinada de tamaños 1 y 2 para variantes de malware. 94
4.8 Histograma de similitudes para una representación combinada de tamaños 1 y 2 cuando se compara con software benigno. 94
4.9 Histograma de similitudes para un tamaño de secuencia de 3 para variantes de malware. 95
4.10 Histograma de similitudes para un tamaño de secuencia de 3 cuando se compara con software benigno. 96
4.11 Histograma de similitudes para una representación combinada de tamaños 1, 2 y 3 para variantes de malware. 96
4.12 Histograma de similitudes para una representación combinada de tamaños 1, 2 y 3 cuando se compara con software benigno. 97
4.13 Histograma de similitudes para un tamaño de secuencia de 4 para variantes de malware. 98
4.14 Histograma de similitudes para un tamaño de secuencia de 4 cuando se compara con software benigno. 98

4.15 Histograma de similitudes para una representación combinada de tamaños 1, 2, 3 y 4 para variantes de malware. 99
4.16 Histograma de similitudes para una representación combinada de tamaños 1, 2, 3 y 4 cuando se compara con software benigno. 100
4.17 Histograma de similitudes para un tamaño de secuencia de 5 para variantes de malware. 101
4.18 Histograma de similitudes para un tamaño de secuencia de 5 cuando se compara con software benigno. 101
4.19 Histograma de similitudes para una representación combinada de tamaños 1, 2, 3, 4 y 5 para variantes de malware. . . 102
4.20 Histograma de similitudes para una representación combinada de tamaños 1, 2, 3, 4 y 5 cuando se compara malware con software benigno. 102
4.21 Los resultados de tiempo para cada valor de tamaño de secuencia, expresados en milisegundos, requeridos para analizar un *megabyte* de datos. 108
4.22 El porcentaje del análisis dedicado a cada fase. 109
4.23 Relación entre el tamaño del fichero y el tiempo necesario para el análisis. 110
4.24 Resultados de FPR y FNR. 110
4.25 El modelo filogenético de la familia *Antilam*. 113
4.26 El modelo filogenético de la familia *Cmjspy*. 113
4.27 El modelo filogenético de la familia *Bancodor*. 114
4.28 El modelo filogenético de la familia *Gruel*. 115
4.29 El modelo filogenético de la familia *Zokrim*. 116
4.30 Método para etiquetar variantes basado en modelos filogenéticos. 118

5.1 Un ejemplo de árbol de decisión. 123
5.2 Pseudo-código de C4.5. 125
5.3 Generación de los árboles de decisión con el algoritmo *random forest*. 126
5.4 Ejemplo de un clasificador SVM en un espacio bi-dimensional.128
5.5 Ejemplo de un KNN en un espacio bi-dimensional. 132
5.6 Ejemplo de red bayesiana. 133
5.7 Algoritmo *hill climbing*. 135
5.8 Algoritmo K2. 136

5.9 Selección a través de Rocchio de las instancias negativas de $\mathscr{U}$ a $\mathscr{N}$. 139
5.10 Generando el clasificador mediante Roc-SVM. 140
5.11 Los resultados de detección de malware desconocido cuando etiquetamos la clase compuesta de ejecutables maliciosos. . . 149
5.12 Los resultados de detección de malware desconocido cuando etiquetamos la clase compuesta por ejecutables benignos. . . 150

A.1 Diseño final de la base de datos. 197
A.2 El producto de módulos implementado en *Transact-SQL*. . . . 201
A.3 El producto escalar implementado en *Transact-SQL*. 202
A.4 La función de similitud del coseno implementada en *Transact-SQL*. 202
A.5 El procedimiento almacenado que se utiliza para recuperar una lista de variantes similar al ejecutable en inspección, implementado en *Transact-SQL*. 203
A.6 Diagrama de clases para los proyectos *DataAccess* y *WeightReader*. 204
A.7 Diagrama de clases para los proyectos *UnknownMalwareDetection*, *DissassembleHandler*, *GUI* y *OpcodeAnalyzer*. 205
A.8 Diagrama de secuencia para el análisis de un fichero ejecutable mediante el método de detección de variantes. 206
A.9 Diagrama de secuencia para el análisis de un fichero ejecutable mediante el método de detección de malware desconocido..207
A.10 Interfaz gráfica de NOA. Pestaña de análisis de variantes. . . 208
A.11 Interfaz gráfica de NOA. Proceso de análisis de variantes. . . 209
A.12 Interfaz gráfica de NOA. Pestaña de análisis de malware desconocido. 210
A.13 Interfaz gráfica de NOA. Pestaña de actualización de la base de datos de variantes. 210
A.14 Interfaz gráfica de NOA. Pestaña de configuración de la aplicación. 211

Índice de tablas

2.1 Comparación de los diferentes esquemas de desempaquetado dinámico. 52

3.1 Tipos de operandos según Christodorescu. 58
3.2 Tipos de operando genéricos. 58
3.3 Comparación de los 14 *opcodes* más frecuentes (%). 62
3.4 Variación de Cramer en los *opcodes* más frecuentes. 62
3.5 Comparación de los *opcodes* menos utilizados (en partes por millón). 63
3.6 Variación de Cramer en los *opcodes* menos frecuentes. 63
3.7 Descripción del conjunto de datos para la evaluación de tiempos de NOA. 71

4.1 Descripción del conjunto de datos de familias de malware para la comparación de similitud. 89
4.2 Las 5 familias de malware con los ratios de similitud más altos para una longitud de secuencias de 1. 91
4.3 Las 5 familias de malware con los ratios de similitud más altos para una longitud de secuencias de 2. 93
4.4 Las 5 familias de malware con los ratios de similitud más altos para longitudes de secuencia de 1 y 2. 95
4.5 Las 5 familias de malware con los ratios de similitud más altos para un tamaño de secuencia de 3. 95
4.6 Las 5 familias de malware con los ratios de similitud más altos para longitudes de secuencia de 1, 2 y 3. 97
4.7 Las 5 familias de malware con los ratios de similitud más altos para un tamaño de secuencia de 4. 99
4.8 Las 5 familias de malware con los ratios de similitud más altos para longitudes de secuencia de 1, 2, 3 y 4. 99

4.9 Las 5 familias de malware con los ratios de similitud más altos para un tamaño de secuencia de 5. 100
4.10 Las 5 familias de malware con los ratios de similitud más altos para longitudes de secuencia de 1, 2, 3, 4 y 5. 103
4.11 Los parámetros utilizados para el primer experimento de detección. 105
4.12 Los parámetros utilizados para el segundo experimento de detección. 106
4.13 Descripción del conjunto de datos para el experimento de generación de modelos filogenéticos. 111

5.1 Resultados de detección de malware desconocido para un tamaño de secuencia de 1. 145
5.2 Resultados de detección de malware desconocido para un tamaño de secuencia de 2. 146
5.3 Resultados de detección de malware desconocido para una combinación de secuencias de *opcodes* de tamaño 1 y 2. 147

A.1 Especificación de la tabla de la BBDD *Settings*. 197
A.2 Especificación de la tabla de la BBDD *MaliciousExecutable*. . . 198
A.3 Especificación de la tabla de la BBDD *Opcode*. 198
A.4 Especificación de la tabla de la BBDD *ExecutableModel*. 199
A.5 Especificación de la tabla de la BBDD *TempAnalysis*. 199

C.1 Muestras de la familia *Antilam* utilizadas en la validación. . . 225
C.2 Muestras de la familia *Bancodor* utilizadas en la validación. . 226
C.3 Muestras de la familia *Gruel* utilizadas en la validación. . . . 226
C.4 Muestras de la familia *Hupigon* utilizadas en la validación. . . 227
C.5 Muestras de la familia *PcClient* utilizadas en la validación. . . 228
C.6 Ejecutables benignos para la validación del sistema de detección de variantes. 228

«Debemos recordar que estamos en las fases preliminares de una de las grandes batallas de la historia, que nosotros estamos actuando en muchos puntos de Noruega y Holanda, que estamos preparados en el Mediterráneo, que la batalla aérea es continua y que muchos preparativos tienen que hacerse aquí y en el exterior. En esta crisis, espero que pueda perdonárseme si no me extiendo mucho al dirigirme a la Cámara hoy. Espero que cualquiera de mis amigos y colegas, o antiguos colegas, que están preocupados por la reconstrucción política, se harán cargo, y plenamente, de la falta total de ceremonial con la que ha sido necesario actuar. Yo diría a la Cámara, como dije a todos los que se han incorporado a este Gobierno: "No tengo nada más que ofrecer que sangre, esfuerzo, lágrimas y sudor". Tenemos ante nosotros una prueba de la más penosa naturaleza. Tenemos ante nosotros muchos, muchos, largos meses de combate y sufrimiento. Me preguntáis: ¿Cuál es nuestra política? Os lo diré: hacer la guerra por mar, por tierra y por aire, con toda nuestra potencia y con toda la fuerza que Dios nos pueda dar; hacer la guerra contra una tiranía monstruosa, nunca superada en el oscuro y lamentable catálogo de crímenes humanos. Ésta es nuestra política.

Me preguntáis; ¿Cuál es nuestra aspiración? Puedo responder con una palabra: Victoria, victoria a toda costa, victoria a pesar de todo el terror; victoria por largo y duro que pueda ser su camino; porque, sin victoria, no hay supervivencia. Tened esto por cierto; no habrá supervivencia para todo aquello que el Imperio Británico ha defendido, no habrá supervivencia para el estímulo y el impulso de todas las generaciones, para que la humanidad avance hacia su objetivo. Pero yo asumo mi tarea con ánimo y esperanza.

Estoy seguro de que no se tolerará que nuestra causa se malogre en medio de los hombres. En este tiempo me siento autorizado para reclamar la ayuda de todas las personas y decir: Venid, pues, y vayamos juntos adelante con nuestras fuerzas unidas.»

Discurso de Winston Churchill a la Cámara de los Comunes.
13 de mayo de 1940

«Es un hecho desafortunado que sólo podamos asegurar la paz preparándonos para la guerra.»

John F. Kennedy (1917–1963)

Capítulo

1

Introducción

Técnicamente, el término *malware* se designa para nombrar a cualquier software con intenciones maliciosas. Si estos ejecutables se han identificado y registrado con anterioridad, es posible enfrentarse fácilmente a ellos. El método clásico de firmas, que es el método más utilizado contra el malware, genera *firmas* de los patrones dentro de los ejecutables maliciosos con el fin de detectar malware.

Desafortunadamente, la cantidad de malware se está incrementando cada año [Kar10], así como su capacidad para evitar cualquier tipo de barrera de seguridad [SBN+10, SPDB09]. Además, los creadores de malware utilizan actualmente técnicas de ofuscación de código buscando ocultar las amenazas de seguridad ante los métodos clásicos de detección.

Estos hechos han llevado a una situación en la que los creadores de malware desarrollan nuevos ejecutables maliciosos, mientras que los investigadores diseñan nuevos métodos y estrategias para detectarlos, actuando como los personajes principales de un juego de ofuscación y des-ofuscación [CJS+05]. Es por esto que el nuevo malware y las variantes ofuscadas de software malicioso conocido sobrepasan fácilmente a los métodos clásicos de detección.

El resto del capítulo está organizado como sigue. La sección 1.1 describe de un modo conciso la historia del software malicioso. La sección 1.2 describe cuáles son los retos en la detección actual de malware. La sección 1.3

establece la hipótesis de la tesis doctoral y enumera los objetivos a cumplir con esta investigación. La sección 1.4 muestra una visión general de la solución que se ha planteado. La sección 1.5 describe la metodología de investigación que se ha utilizado a lo largo de esta investigación. Finalmente, la sección 1.6 detalla la organización de este documento.

1.1 Malware, una amenaza creciente

Esta sección revisa la historia del software malicioso, comenzando por las primeras aplicaciones con fines malignos hasta las amenazas actuales.

1.1.1 Antes de que el malware ni siquiera existiese

No está claro cuando apareció la primera aplicación maliciosa. A pesar de que el primer ordenador de la historia no tenía ningún ejecutable malicioso, los ordenadores IBM 360/370 y Univax 1008, a mediados de los años 1970, sí que los tenían [Vir10d].

La idea de los virus informáticos (un tipo de malware que se replica a sí mismo) apareció mucho antes. Por ejemplo, en los trabajos de John von Neumann, acerca de un autómata auto-replicante en los años 50 que completó en 1966 [NB66] o, en 1959, en la visión de Lionel Penrose acerca de la auto-replicación automática [Pen59].

En 1962, un grupo de ingenieros de los laboratorios de *America's Bell Telephone*, compuesto por V. Vyssotsky, G. McIlroy y R. Morris[1], crearon un juego llamado *Darwin*. El juego consistía en un árbitro dentro de la memoria del ordenador que establecía las reglas del juego y el orden de una batalla entre programas creados por los jugadores. Estos programas podían rastrear y destruir los programas de los oponentes y multiplicarse. El objetivo del juego era borrar los programas del adversario y conseguir el control sobre el campo de batalla (la memoria del ordenador rival) [Vir10d].

1.1.2 La primera oleada de amenazas

Las ideas creativas de los científicos y el entretenido juego de los ingenieros fueron ensombrecidos en el momento en el que el mundo se dio cuenta de

[1] Robert Morris es el padre de Robert Morris Jr., el creador del famoso *gusano de Morris* que puso en jaque a ARPANET (la precursora de la actual Internet).

que la teoría detrás de los ejecutables auto-replicantes podía utilizarse con intenciones mucho más oscuras.

En la década de 1970 se detectó el ejecutable *Creeper* dentro de ARPANET[1]. El objetivo principal de *Creeper* era obtener acceso a través de un módem y copiarse a sí mismo en el sistema remoto. Los sistemas infectados mostraban el mensaje: «Soy la enredadera: cógeme si puedes». Al poco tiempo se creó, de un modo anónimo, el programa *Reaper* para eliminar a *Creeper*. *Reaper* también se puede considerar malware ya que se distribuía a través de redes de equipos y borraba el virus *Creeper* [Vir10a].

En 1974 aparece el virus *Rabbit*, denominándose jocosamente de este modo, pues no hacía otra cosa que multiplicarse y propagarse a otras máquinas. *Rabbit* era capaz de saturar la máquina infectada con copias de sí mismo, destrozando el rendimiento del sistema. Una vez que *Rabbit* llegaba a cierto nivel de reproducción, el propio ejecutable fallaba [Vir10a].

Pervading Animal, otro juego, aparece en el año 1975. Los analistas aún debaten si se trataba de otro virus o del primer caballo de Troya (un programa malicioso que consigue acceso al equipo de una víctima). Las reglas del juego eran simples, el jugador pensaba en un animal y el programa le formulaba preguntas para intentar adivinarlo. El juego venía equipado con una función de auto-corrección: si el programa era incapaz de adivinar el animal, *Pervading Animal* se actualizaba y añadía nuevas preguntas. Una versión más moderna del juego incluía una función para reemplazar a la versión antigua y se copiaba a través de los directorios [Vir10a].

1.1.3 El malware comienza a evolucionar

Los primeros incidentes registrados de auténticos virus datan de 1981 y de 1982, en el caso del sistema Apple II [Fer92]. Se considera a *Elk Cloner* el primer ejemplo documentado de un virus [SLRT91].

De igual modo, el primer virus para ordenadores personales compatibles (PC) era un virus del sector de arranque llamado *Brain*. En esta época, Robert Morris Jr. creó un gusano (un programa que se propaga a través de la red), que se propagaba a través de la red y cambiaba la configuración de dispositivos y redes.

En 1990, se observaron importantes cambios en el desarrollo y creación de malware [Vir10c]. Aparecieron los primeros ejecutables polimórficos (el

[1] ARPANET (Advanced Research Projects Agency Network) fue la primera red de intercambio de paquetes del mundo y la predecesora de Internet.

polimorfismo consiste en cifrar el código para ocultar las verdaderas intenciones): la familia *Chamaleon* compuesta por las versiones 1260, V2P1, V2P2 y V2P6. Esta familia era la evolución de dos virus anteriores: *Vienna* y *Cascade*. El segundo acontecimiento importante es la aparición de *Bulgarian Virus Producing Factory* [Vir10b].

En esos días, el malware era creado por jóvenes programadores, niños que acababan de aprender a programar y que, buscando poner a prueba sus habilidades [Vir10e], eventualmente, decidían escribir software malicioso. Los creadores de malware creaban amenazas por una combinación de aburrimiento, fama o curiosidad [Gor95]. Además, aquellos primeros autores jamás tenían objetivos específicos y no lanzaban el malware al público [Gor94].

1.1.4 La profesionalización del malware

Actualmente, la creación de malware está profesionalizada. La compañía española de soluciones antivirus *Panda Software*[1] publicó en el año 2005 un informe sobre los cambios en las tendencias de la creación de malware:

> «*Los tiempos en los que los creadores de malware buscaban notoriedad y reconocimiento de sus malignas creaciones hace tiempo que pasaron. (...) Los ciber-delincuentes, motivados por el beneficio económico, se han adaptado bien al nuevo entorno: ahora buscan infecciones silenciosas para que sus creaciones funcionen sin hacer demasiado ruido en los medios de comunicación. (...) Esto significa que las posibles víctimas no están en guardia y son más susceptibles al robo de dinero y de todo tipo de información.*»

PandaLabs[2] explicó la tendencia en la creación de nuevo malware, remarcando la mayor profesionalización en el sector, y subrayó uno de los puntos clave en la seguridad de la información en el año 2005: el aumento de los ataques dirigidos. Una de las observaciones más importantes del informe es la profesionalidad de los creadores de malware, que tienen ahora como motivación el beneficio económico [Lab05].

Se ha creado un nuevo modelo de negocio basado en el malware, con una alianza entre desarrolladores, distribuidores y compañías, que hacen uso de esa infraestructura maliciosa.

[1] Panda Software cambio su nombre en el año 2007 a Panda Security.
[2] PandaLabs es el laboratorio de análisis de Panda Security.

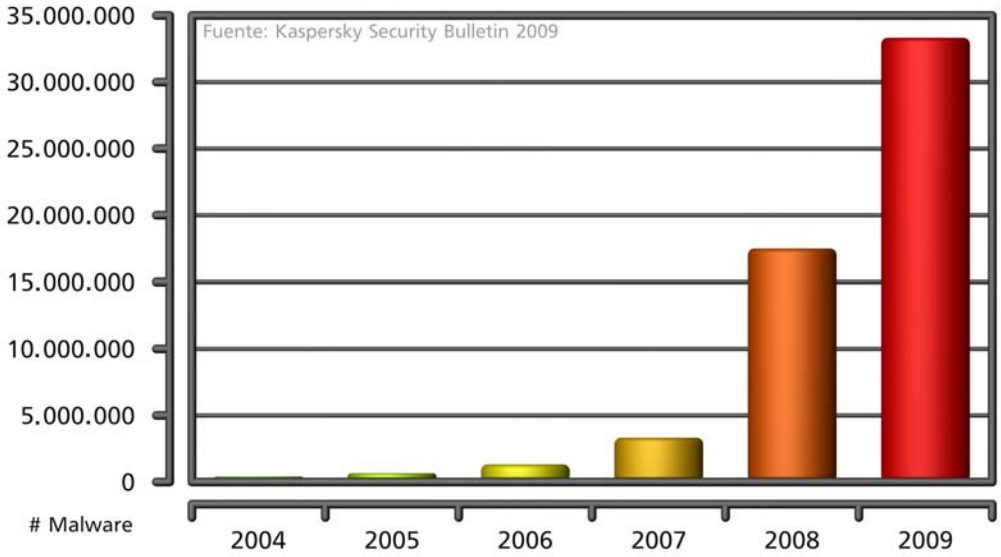

Figura 1.1: Crecimiento del nuevo malware en los últimos años.

Desafortunadamente, las técnicas que se utilizan en los antivirus comerciales requieren inspeccionar el ejecutable, decidir si es malware o no y desarrollar una firma para él. Como consecuencia de este hecho y del aumento exponencial de la cantidad de malware, estas soluciones para detectar malware no son capaces de enfrentarse ni al nuevo malware ni a las nuevas variantes ofuscadas del mismo [SYS+08, SPDB09].

1.2 Retos a superar

Existen ciertos problemas que aún no tienen solución y que constituyen los retos de la detección actual de malware [Sub08, SZ03, SYS+08, ME08, IM07]. Específicamente, nuestra investigación se centra en los siguientes:

- **Detección de variantes de malware.** Como hemos mencionado, los creadores de malware utilizan técnicas para ocultar el comportamiento real de sus creaciones y generan variantes de los ejecutables una vez que éste ha sido identificado y registrado, para así, evitar que puedan ser detectadas [CJ04, CJ03].

  Por consiguiente, desarrollar un método que sea capaz de detectar la mayoría de las ofuscaciones de malware es un requisito importante para una mejor detección de éstas [Chr07, CKJ+05, CJS+05, JDF08,

SMEG09]. Los métodos de firmas clásicos, a pesar de sus limitaciones, son capaces de etiquetar un ejecutable malicioso con la familia de malware (una familia de malware es un conjunto de ejecutables que comparten el mismo comportamiento malicioso pero no la misma implementación) a la que pertenecen. Consideramos que esta habilidad es una de las más importantes bondades de los sistemas clásicos de firmas. Vamos a desarrollar un método que sea capaz de detectar variantes de malware y, a su vez, mantenga la habilidad para el *etiquetado específico*.

- **Detección de malware desconocido.** Uno de los retos más relevantes a superar en la detección de malware es proveer a los analistas de malware de un método capaz de detectar amenazas desconocidas, que aún no hayan sido documentadas.

  La cantidad de malware se está incrementando cada vez más rápido [Kar10] y no sólo se desarrollan nuevas variantes de malware conocido, sino también malware completamente nuevo. Aunque actualizar las bases de firmas contribuye notoriamente a reducir el impacto de esta realidad, la solución final pasa por afrontar este problema directamente [IM07, JDF08, SMEG09].

- **Optimizar las tasas de falsos negativos y falsos positivos.** Existen varias técnicas de detección de malware que sufren de una alta tasa de falsos positivos (un falso positivo es un ejecutable benigno incorrectamente clasificado como malware); especialmente aquéllas que se centran en la detección de malware desconocido. Además, cuando se trata de identificar ofuscaciones de malware conocido, existe la necesidad de minimizar el número de ejecutables maliciosos no detectados, en otras palabras, minimizar la tasa de falsos negativos (un falso negativo es un ejecutable malicioso incorrectamente clasificado como software benigno). Encontrar un equilibrio entre las tasas de falsos positivos y falsos negativos es un problema a resolver en el futuro para poder asegurar el mejor rendimiento posible de cualquier técnica que se proponga [IM07, JDF08, SMEG09].

- **Adaptación continua del conocimiento.** Aunque los métodos de firmas clásicos han sido capaces hasta ahora de actualizar su base de conocimiento en un tiempo suficiente para proporcionar detección de malware práctica, el actual aumento del crecimiento de malware

amenaza esta situación. Como el malware está en constante evolución, modelar el desarrollo de esta evolución es una tarea necesaria para completar la fase de detección [KWLP05a, KWLP05b].

- **Eficiencia de los detectores de malware.** El método de firmas es una técnica rápida y, por esta razón, aún es la base de las aplicaciones comerciales anti-malware. Los enfoques académicos para la detección no son, por mucho, tan rápidos, y en muchas ocasiones su complejidad los convertiría en un cuello de botella si se usasen dentro de un anti-malware comercial, ante la enorme cantidad de malware que llega cada día a los analistas [Chr07, CKJ+05, CJS+05]. Por tanto, los nuevos detectores de malware, no sólo tienen como objetivo el detectar una gran cantidad de malware, sino que también tienen que ser capaces de hacerlo con celeridad [SJB04, IM07].

1.3 Hipótesis y objetivos

Los retos que hemos enumerado suponen los límites actuales de la detección de malware y es ahí donde la comunidad científica está colocando sus esfuerzos. Por tanto, hemos establecido la siguiente hipótesis:

> «*Existe un método de representación rápido para ejecutables con el que un detector de malware es capaz, tanto de identificar variantes de familias de malware conocidas como de detectar malware desconocido.*»

La representación que se consiga será la base para la construcción de un detector de malware que supere las barreras de la detección de malware actual, detectando tantas variantes como sea posible y, a su vez, identificando nuevo malware antes de que estas amenazas hayan sido capaces de causar daño. Además, ésta tiene que ser bastante rápida, es decir, que no introduzca un esfuerzo demasiado alto en la fase de representación de ejecutables. Por tanto, la representación tiene que compararse con los métodos más rápidos para la representación de ejecutables como: las funciones resumen.

Para la ratificación de esta hipótesis, hemos planteado una investigación con el siguiente objetivo principal:

Objetivo principal 1 *Desarrollar y evaluar un detector de malware capaz de detectar variantes de malware y malware nuevo y desconocido.*

Para lograr el objetivo principal, planteamos, a su vez, los siguientes objetivos específicos.

Objetivo específico 1 *Desarrollar un método de representación rápido que actúe como marco de trabajo para una detección de malware completa.*

Objetivo específico 2 *Desarrollar y evaluar un detector de malware capaz de detectar nuevas variantes del malware actual.*

Objetivo específico 3 *Desarrollar y evaluar un detector de malware capaz de detectar malware desconocido.*

El primer objetivo específico es encontrar un modo de representar ejecutables para mejorar la detección de malware. Esta representación tiene que ser rápida y automática. El segundo objetivo específico es detectar variantes de familias de malware. Finalmente, el tercer y último objetivo específico afronta el problema de detectar malware nuevo y desconocido. La integración de la detección de variantes de malware y de malware desconocido se realiza a través de una representación común de los ejecutables.

A fin de lograr estos objetivos específicos, nos hemos marcado los objetivos operacionales siguientes:

Objetivo Operacional 1 *Diseñar e implementar un método para la representación de ejecutables, así como validarlo en términos de velocidad.*

Objetivo Operacional 2 *Diseñar e implementar un marco de trabajo completo y escalable para la detección de variantes de malware.*

Objetivo Operacional 3 *Desarrollar un método para seguir la evolución de las variantes de malware y organizarlas.*

Objetivo Operacional 4 *Minimizar el número de ejecutables benignos incorrectamente clasificados como malware en la detección de variantes malware, proveyendo de un método para configurar lo estricto del detector.*

Objetivo Operacional 5 *Desarrollar y validar un método para detectar malware desconocido.*

Objetivo Operacional 6 *Desarrollar un método alternativo para la detección de malware desconocido que minimice el esfuerzo previo de etiquetado.*

Objetivo Operacional 7 *Minimizar el número de ejecutables benignos incorrectamente clasificados como malware en la detección de malware desconocido.*

Estos objetivos operacionales describen las actividades que vamos a realizar durante la investigación. Para evaluar las soluciones alcanzadas en torno a la detección de variantes de malware y malware desconocido, evaluaremos ambos componentes en términos de detección de malware y número de ejecutables benignos clasificados erróneamente. La representación de ejecutables se valorará, en primer lugar, en función de su velocidad para, más tarde, valorar su capacidad para detectar tanto variantes de familias de malware conocidas como malware desconocido.

1.4 Arquitectura global de la solución

Una vez enumerados los objetivos de la investigación, hemos planteado una solución global con el fin de desarrollar un sistema final en el que estén agrupados los distintos componentes que se obtengan con el logro de los objetivos.

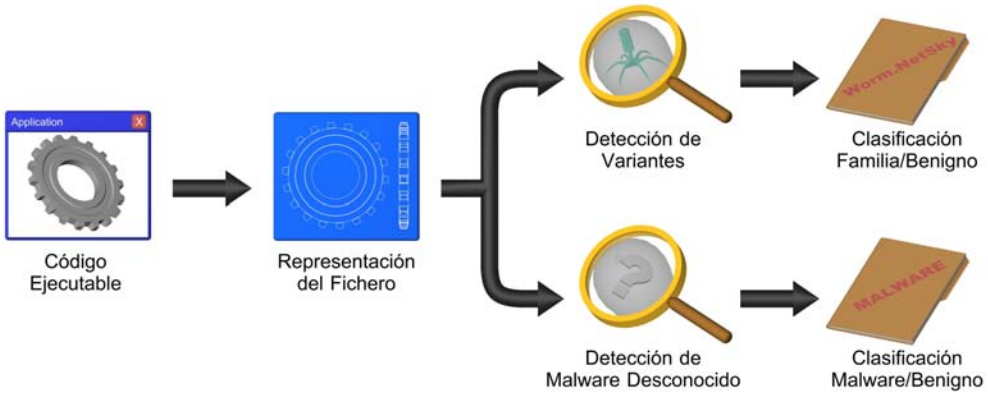

Figura 1.2: Arquitectura de la solución planteada.

La Figura 1.2 muestra la arquitectura global del sistema final, que está formado de los siguientes componentes:

- **Representación de ejecutables.** Existe un componente de representación que, a partir de un ejecutable, extrae una representación del mismo. Esta representación estará basada en técnicas de recuperación de información y ha de ser rápida.

- **Detección de variantes.** El componente de detección de variantes cuenta con una base de datos en la que se almacenan representaciones de familias de malware conocidas. Así, a partir de la representación del ejecutable que el sistema esté analizando, se comparará con las previamente almacenadas en la base de datos para valorar su similitud con éstas. Si se encuentra alguna similitud alta, entonces, el ejecutable se categorizará como la familia de malware con la que coincida.

- **Detección de malware desconocido.** El módulo de detección de malware desconocido empleará un modelo de aprendizaje automático entrenado con muestras de software benigno y software malicioso. De este modo, este componente clasificará un ejecutable en inspección como legítimo o malicioso.

El sistema funciona como sigue. En un primer lugar, existe un componente de representación de los ficheros ejecutables que obtiene una representación a partir de un fichero ejecutable para que, posteriormente, sea analizado por el resto de componentes. Una vez que se ha obtenido la representación del ejecutable en inspección existen dos maneras de analizarlo. Primeramente, se podrá analizar con el detector de variantes. El detector de variantes comparará la representación del ejecutable con representaciones previamente almacenadas y conocidas para identificar si éste es alguna ofuscación de algún comportamiento de malware conocido. Por otro lado, nuestro sistema también será capaz de detectar malware completamente desconocido. Para ello, se hará uso del componente de detección de malware desconocido, que a partir de un modelo de aprendizaje automático determinará si el ejecutable en inspección es susceptible de ser malware o no.

1.5 Metodología de investigación

Nuestro objetivo en esta tesis doctoral es ampliar el conocimiento en el campo de la detección de malware. Para ello, necesitamos adquirir un conocimiento profundo de las problemáticas que está atravesando la detección de malware actual, así como proponer nuevas soluciones.

Para conseguir estos objetivos, la investigación se llevará a cabo con la siguiente metodología:

1. **Adquisición de conocimiento.** En primer lugar, debemos analizar el entorno actual en cuanto a la detección de malware. Para ello, estudiaremos en profundidad el estado del arte acerca de esta temática para determinar cuáles son sus líneas de investigación en la comunidad y, por tanto, los retos a resolver.

2. **Formulación de preguntas de investigación.** En este paso, formularemos las preguntas a resolver durante fases posteriores, marcando así las líneas a seguir y los retos a resolver por esta investigación.

3. **Desarrollo de nuevos componentes.** Desarrollaremos nuevos componentes capaces de enfrentarse a la actual amenaza del malware que sean capaces de resolver las preguntas de investigación.

4. **Evaluación de los diferentes componentes.** Una vez que los componentes estén desarrollados, los evaluaremos para dar respuesta a los interrogantes formulados en la fase de desarrollo. Para su evaluación, diseñaremos una metodología que, por un lado, establezca los pasos a seguir en la evaluación y, por otro lado, establezca las métricas que midan el grado de consecución de los objetivos y preguntas planteadas.

5. **Obtener respuesta de la comunidad científica.** Presentaremos los resultados de la investigación en congresos internacionales y revistas. De este modo, conseguiremos la opinión de la comunidad científica acerca de nuestro trabajo.

   Además, el acudir a conferencias de este área nos permitirá profundizar en el estudio y comprensión de los avances conseguidos por otros grupos de investigación. Por consiguiente, en función de las respuestas obtenidas por la comunidad científica, modificaremos nuestras soluciones para adecuarlas a los nuevos requerimientos.

6. **Difusión de los resultados.** Una vez consideremos que los objetivos de la investigación se han conseguido plenamente, pasaremos a comunicar los resultados finales de la investigación.

A la hora de desarrollar los diferentes enfoques, utilizaremos la metodología de Investigación-Acción [Ber89]. Investigación-Acción es una metodología cíclica que está compuesta de los siguientes pasos iterativos:

1. **Diagnóstico:** identificamos o definimos un problema.
2. **Planificación de la acción a tomar:** consideramos las diferentes vías de acción para abordar el problema.
3. **Acción:** seleccionamos la acción a tomar.
4. **Evaluación:** estudiamos las consecuencias de la acción.
5. **Especificación del aprendizaje:** identificamos los descubrimientos generales.

En primer lugar, comenzamos identificando un problema y recolectando evidencias que hagan posible el diagnóstico detallado. Entonces, consideramos las soluciones posibles, de las cuales seleccionamos una única acción. Recogemos los datos resultantes de la acción y los analizamos. Finalmente, interpretamos los descubrimientos en términos de cómo de satisfactoria ha sido la acción elegida. Re-formulamos el problema y repetimos el proceso hasta que el problema esté completamente resuelto. Es un metodología inherentemente participativa en donde los actores que están implicados en el desarrollo tiene un papel importante. En concreto, ya que la última parte de la investigación en la que se desarrolla el prototipo funcional del sistema se ha realizado bajo el marco del proyecto Segur@, los actores implicados en esta última fase son el grupo de investigación de DeustoTech y de Optenet.

De cara a la evaluación, diseñaremos una metodología que establezca, por un lado, los pasos a seguir y, por otro lado, las métricas para medir el grado de consecución de los objetivos y preguntas planteadas. Tal y como se ha mencionado en el apartado 1.4, se desarrollarán tres componentes: (i) representación de ejecutables, (ii) detección de variantes y (iii) detección de malware desconocido. El primero se validará, en primera instancia, en términos de velocidad o coste computacional. Las posibles restricciones en la configuración de la representación que se obtengan se arrastrarán a los componentes de detección. La detección de variantes se evaluará utilizando mutaciones en familias conocidas de malware y software benigno. El objetivo es comprobar si nuestro método es o no capaz de identificar la familia de las variantes desconocidas de familias conocidas y verificar el número de posibles falsos positivos. En lo referente a la detección de malware desconocido, utilizaremos un conjunto de aplicaciones en la que la mitad de éstas serán maliciosas. Con un enfoque de validación cruzada en el que dividiremos el conjunto de datos varias veces en entrenamiento y prueba para

así mejorar la bondad de los resultados. Dentro de este componente, para el método de aprendizaje de clase única, hay que evaluar cómo va afectando el número de muestras etiquetadas dentro del conjunto de entrenamiento.

1.6 Estructura del documento

La presente tesis doctoral está organizada en los capítulos que se enumeran a continuación:

- **Capítulo 1. Introducción.** En este capítulo se introduce brevemente la historia del malware y la evolución del mismo durante los últimos años. Se enumeran los retos que esta investigación busca solventar para, así, sentar la hipótesis fundamental de la tesis doctoral. Con ella, se establecen los objetivos a cumplir en la investigación. Seguidamente, se detalla la metodología general para la consecución de los mismos. Por último, se describen las principales aportaciones de la presente tesis doctoral al área de la detección de malware.

- **Capítulo 2. Estudio del estado del arte.** En este capítulo se realiza un exhaustivo estudio de la literatura acerca del análisis y detección del malware. Primeramente, se introducen ciertos conceptos básicos del malware, como sus tipos, métodos de ofuscación y otro tipo de terminología relacionada con el malware. Después, se clasifican los diferentes métodos de detección de malware en categorías para, a partir de esta clasificación, describir las principales aportaciones a cada una de las categorías de la misma. Además, se detallan aportaciones a aspectos relacionados con la detección de malware.

- **Capítulo 3. Una representación de ejecutables basada en recuperación de información.** En este capítulo se introduce un método de representación de ejecutables que está basado en técnicas de representación de documentos, propias del área de la recuperación de información. El enfoque extrae las frecuencias de aparición de secuencias de códigos operacionales del lenguaje máquina en los que se especifican los ejecutables. Hemos evaluado el rendimiento del modelo comparándolo con funciones de resumen criptográfico que son los métodos más rápidos posibles, siendo nuestra representación entre 8 y 16 veces más lenta que éstas — mucho más rápida que otros enfoques para la detección de malware [Chr07] — y, por tanto, tan rápida como para no convertirse en un *cuello de botella*.

- **Capítulo 4. Detección de variantes de malware.** En este capítulo se introduce y detalla nuestra solución para la detección de variantes de malware. En base a la representación descrita en el capítulo 3, construimos un sistema de detección de variantes que consta de: una base de datos de malware, un sistema que transforma un ejecutable a una consulta y una métrica de similitud que recupera las variantes de la base de datos. Además, hemos realizado un sistema de generación de modelos filogenéticos capaz de evaluar y monitorizar la evolución de una familia de malware. Hemos evaluado los enfoques con diferentes tamaños de secuencia, demostrando que nuestro sistema es capaz de detectar variantes de malware y que es capaz de representar a través de modelos filogenéticos la evolución de una familia de malware.

- **Capítulo 5. Detección de malware desconocido.** En este capítulo se describen varios métodos para detectar malware desconocido. En primer lugar, hemos evaluado un modelo de aprendizaje automático supervisado, demostrando que utilizando la representación de ejecutables propuesta, se puede detectar malware desconocido. Además, hemos adaptado un modelo de aprendizaje de clase única, con el fin de reducir los esfuerzos a la hora de etiquetar software. Hemos evaluado este último enfoque y si bien no es tan preciso como el supervisado, se consigue reducir los esfuerzos de etiquetado de la industria antimalware, manteniendo un alto grado de precisión.

- **Capítulo 6. Conclusiones.** En este último capítulo, se comienza evaluando el grado de consecución de la hipótesis y los objetivos. Se discuten las aplicaciones de esta investigación y sus campos de dominio. Seguidamente, se debaten las principales limitaciones de las soluciones propuestas para, más tarde, centrar las líneas de investigación futuras. Finalmente, se concluye la disertación con una breve opinión personal acerca del futuro de este área de investigación.

«Hay muchas hipótesis en ciencia que son erróneas. Esto es perfectamente correcto, estos errores son la base para averiguar lo que es cierto. La ciencia es un proceso de auto-corrección. Para ser aceptadas, las nuevas ideas deben sobrevivir a los más rigurosos niveles de prueba y escrutinio.»

Carl Sagan (1934–1996)

Capítulo 2

Estudio del estado del arte

A lo largo de los últimos años, se ha dedicado una gran cantidad de esfuerzo a la detección de malware y a problemas derivados. Debido al incremento tanto en cantidad como en capacidad del malware en los últimos años [Kar10] y, a su vez, a la incapacidad del método clásico de firmas de hacer frente a esta realidad; la investigación en detección de malware se ha convertido en un tema más activo últimamente, buscando enfrentarse de manera eficaz a estas amenazas. Este capítulo presenta varios conceptos básicos acerca del malware, tales como los diferentes tipos existentes o las técnicas de ofuscación que los creadores de malware utilizan para ocultar el comportamiento de sus ejecutables. Además, se presenta la investigación en este área para encontrar una solución al problema de la detección de malware, proveyendo, a su vez, de una clasificación de las mismas.

Lo que resta de capítulo está organizado como sigue. La sección 2.1 presenta conceptos y definiciones acerca de la detección de malware. La sección 2.2 describe la detección de malware y provee de una clasificación de estas técnicas. La sección 2.3 revisa los métodos propuestos para la detección de variantes de malware en base al conocimiento. La sección 2.4 detalla los diferentes enfoques para la detección de malware desconocido. La sección 2.5 describe y detalla aspectos relacionados con la detección de malware: formalización teórica, empaquetadores y generación automática de firmas o perfiles. Finalmente, la sección 2.6 resume los principales aspectos de este capítulo.

2.1 Conceptos y definiciones

Esta sección introduce los conceptos y definiciones fundamentales relacionados con el malware y su detección. El objeto de proveer dichas definiciones es acercar al lector la terminología que se utilizará a lo largo de este documento. En particular, se definirá el propio malware y conceptos directamente relacionados con éste. A continuación, se introducirán los diferentes tipos de software malicioso, así como las técnicas de ofuscación utilizadas por éstos. Por último, se detallará el estado actual de la detección de malware y su exponente más clásico: la detección en base a firmas.

2.1.1 El ecosistema del malware

El término malware es la combinación de las voces inglesas *malicious* (malicioso) y *software*, y se refiere a cualquier programa informático diseñado con la intención de dañar ordenadores, redes o información [SPDB09, Vas07, IM07]. Christodorescu y Jha [CJS+05] definen malware como:

Definición 2.1 ***Malware (Christodorescu y Jha)***
«Un programa cuyo objetivo es malévolo.»

Por su parte, McGraw y Morrisett [MM00] centran su definición en la peculiar forma en la que el malware suele comportarse:

Definición 2.2 ***Malware (McGraw y Morrisett)***
«Cualquier trozo de código añadido, cambiado, o borrado de un sistema software para causar intencionalmente daño o subvertir la función del sistema.»

Por contra, Karim et ál. [KWLP05b] proponen una definición genérica de malware:

Definición 2.3 ***Malware (Karim et ál.)***
«El término malware es el uso común para nombrar código malicioso de todos los tipos (...)»

Aunque actualmente se utilice el término malware para nombrar estas amenazas, en el pasado, la forma más común de nombrarlas era la expresión *virus informático*[1] que Fred Cohen fue el primero en utilizar en 1983 [Coh84].

[1] El término virus informático fue utilizado por primera vez en una novela de ciencia ficción escrita por David Gerold en el año 1972: *When Harley was one* [Ger72]. En esta obra se describía un programa de ordenador capaz de replicarse a sí mismo.

Ya que el vocablo malware se refiere a cualquier tipo de software con intenciones maliciosas, podemos distinguir diversos tipos de malware incluidos en esta definición. Dependiendo de su objetivo primario, se le dan diferentes nombres a estos ejecutables [Chr07].

Definición 2.4 ***Virus (Christodorescu)***

«Un virus es un programa que se replica infectando a otro programa; un sector de arranque o de partición; o un documento que permita macros. Normalmente, añade una copia de sí mismo a los ficheros de la víctima [Chr07].»

Los virus informáticos fueron una vez los ejecutables maliciosos más extendidos porque las primeras amenazas fueron de este tipo. Ahora la situación ha cambiado y, aunque son un tipo popular de malware, ya no son los más comunes. El término virus fue también normalmente utilizado — aunque de forma errónea — para denominar a otros tipos de malware que no tenían la capacidad reproductora característica de los virus. Siendo estrictos, un virus puede propagarse de una máquina a otra, sólo cuando su portador lo traslada a esa otra máquina.

Definición 2.5 ***Gusano (Dalla)***

«Un programa malicioso que utiliza la red para enviar copias de sí mismo a otros sistemas se denomina gusano. Al contrario que los virus, los gusanos no necesitan que su portador les lleve a otro sistema, ellos mismos se encargan de propagarse a través de la red (...) [Dal07].»

Los gusanos informáticos son muy similares en funcionamiento a los virus ya que también son programas auto-replicantes. Utilizan una red de ordenadores para enviar copias de sí mismos a otros ordenadores o redes. Al contrario que los virus, no necesitan añadirse a un programa existente. Los gusanos causan daño a las redes consumiendo su ancho de banda, mientras que los virus modifican ficheros dentro del sistema objetivo.

Definición 2.6 ***Caballo de Troya (Dalla)***

«Como los virus, los caballos de Troya se ocultan dentro de programas que pueden parecer útiles o, al menos, inofensivos para un usuario. Los caballos de Troya pueden ser tanto programas legítimos corruptos que ejecutan código malicioso al ponerse en funcionamiento; como programas que ejecutan código malicioso directamente, enmascarados como otra cosa, para conseguir el descuido del usuario y, así, alcanzar la complicidad necesaria para llevar a cabo sus objetivos [Dal07].»

También llamados troyanos, los caballos de Troya son un tipo de malware no auto-replicante que parece desarrollar una funcionalidad deseable para el usuario pero que, en vez de eso, facilita el acceso no autorizado a los sistemas de la víctima. Actualmente, los caballos de Troya representan la forma más común de malware. Los troyanos bancarios son, a su vez, un tipo de caballos de Troya cuya intención es obtener un beneficio ilícito mediante transacciones con la cuenta bancaria de una víctima [Stå08].

Definición 2.7 ***Spyware (Christodorescu)***

«Un programa de spyware es un tipo de software que envía información personal recogida del sistema víctima a otra entidad [Chr07].»

El *spyware* es un tipo de malware instalado en los ordenadores que recolecta pequeños esbozos de información sobre los usuarios en cada momento. La presencia del *spyware* está normalmente oculta para el usuario, secretamente instalado en el sistema personal de la víctima. Generalmente, el *spyware*, por ejemplo un *keylogger*[1], se instala por el dueño de un ordenador (personal, corporativo o compartido) para, secretamente, monitorizar a otros usuarios del mismo [HF06].

Definición 2.8 ***Puerta trasera (Dalla)***

«Una puerta trasera es un programa informático diseñado para superar las políticas de seguridad con el fin de permitir a entidades externas tener control sobre una máquina o una red de forma remota. Las puertas traseras pueden ser programas únicos o alojarse dentro de versiones corruptas de programas benignos [Dal07].»

Las intenciones de las puertas traseras son similares a las de los troyanos o los gusanos, pero su funcionamiento interno es diferente. Una puerta trasera en un sistema informático es un método para superar la autenticación habitual, asegurando el acceso remoto a una máquina, intentando no ser detectado. La puerta trasera puede tomar la forma de un programa previamente instalado o puede ser una modificación de un programa o dispositivo hardware existente. Las puertas traseras habilitan un método para acceder a un sistema, proporcionando una conexión remota a *hackers* o a otro malware para que se distribuyan virus o *spam* (correos comerciales no solicitados).

[1] Un *keylogger* es una aplicación que registra las pulsaciones de teclado para guardarlas y/o enviarlas a través de Internet.

Definición 2.9 ***Bomba lógica (Christodorescu)***

«Una bomba lógica es un malware cuyo cuerpo se activa en una situación particular, en un instante concreto de tiempo o cuando ciertas condiciones se satisfacen [Chr07].»

Más que un tipo de malware, las bombas lógicas normalmente forman parte de ejecutables maliciosos. El software malicioso, como los virus o gusanos, generalmente contienen bombas lógicas que ejecutan cierto cuerpo de código (en otras palabras, cierta función) en un instante de tiempo predefinido con anterioridad o cuando alguna condición concreta se activa[1]. Los troyanos que se activan en fechas concretas se denominan *bombas de tiempo*.

Definición 2.10 ***Bot (Christodorescu)***

«Un bot es un tipo de puerta trasera que participa en un sistema de control a gran escala de máquinas víctima. Las máquinas infectadas con el programa bot se conectan a un canal de control, formando una botnet o red de bots, una red de máquinas controladas fácilmente al unísono por un solo atacante [Chr07].»

La forma más común para la expansión de *bots* en una *botnet* (red de *bots*) consiste en la utilización de otro tipo de malware para controlar a la víctima. De este modo, una vez ejecutado el programa, es posible escanear las conexiones de red, discos duros y, por consiguiente, intentar propagarse mediante vulnerabilidades conocidas del sistema operativo, aplicaciones, etc. [LWM09].

No obstante, el malware actual ya no se puede clasificar fácilmente en estas categorías porque ahora combinan más de una de las características que hemos presentado. Esta situación, junto con el incentivo del beneficio económico, ha contribuido a la explosión en el crecimiento del malware [Chr07, Stå08].

2.1.2 Ofuscación de software

Los métodos de ofuscación de software se crearon con la intención de proteger la integridad del software y asegurar su privacidad [KSVZ07]. Estos métodos son capaces de proteger una aplicación software de la piratería.

[1] Históricamente, muchos ejecutables maliciosos atacaban a sus víctimas en fechas específicas, tales como el viernes 13 o el *April Fools Day* (la versión anglosajona del día de los Santos Inocentes).

Generalmente, la prevención de la ingeniería inversa se realiza para manejar riesgos derivados del acceso no autorizado al código fuente. Estos riesgos incluyen la pérdida de la propiedad intelectual, facilidad para explotar vulnerabilidades y pérdida de ingresos resultantes de las víctimas de la ingeniería inversa, modificaciones para eludir la medida o el control del uso. La ofuscación de código es una medida de compensación contra estos riesgos [Dal07].

Así, los creadores de malware utilizan este tipo de técnicas de ofuscación de código a fin de ocultar el comportamiento real de sus ejecutables [Chr07].

2.1.2.1 Conceptos básicos

Un ofuscador es un programa informático que realiza ciertas transformaciones a un ejecutable. El código resultante es equivalente, en cuanto a funcionalidad, al código original pero más difícil de comprender. Collberg et ál. [CTL97, HF98, CTL98] definen la ofuscación como:

Definición 2.11 ***Ofuscación (Collberg)***

«Supongamos que $\mathcal{T} : \mathcal{P} \rightarrow \mathcal{P}'$ es una transformación de un programa de un código original $\mathcal{P}$ a un código resultante $\mathcal{P}'$. $\mathcal{T} : \mathcal{P} \rightarrow \mathcal{P}'$ es una transformación ofuscadora si:

- *La transformación $\mathcal{T}$ es potente y*
- *$\mathcal{P}$ y $\mathcal{P}'$ tienen el mismo comportamiento observable, es decir, si $\mathcal{P}$ no termina o termina con error ante una condición dada entonces $\mathcal{P}'$ puede terminar con error o no terminar,*
- *en el caso de que termine, $\mathcal{P}'$ tiene que terminar y producir la misma salida que $\mathcal{P}$.»*

Sin embargo, sólo se considera *potente* a una transformación en caso de que el programa resultante sea más difícil de comprender que el original [DG05]. Esta definición de ofuscación de código se basa en la noción de *potencia* de una transformación, que es una métrica que mide la complejidad de un programa. No obstante, definir un métrica para la complejidad no es una tarea trivial [Hal77, Mar94].

Existen algunas características que ayudan a medir la complejidad de un programa: la longitud del mismo (el número de instrucciones y argumentos) [Hal77], el nivel de anidamiento (el número de condiciones anidadas),

el flujo de datos (p. ej., el número de referencias a variables locales) [GL06] o la complejidad de las estructuras de datos [Jon08]. Así, dada una métrica para la complejidad de un programa, es posible medir — o estimar — la potencia de una ofuscación.

Dalla [Dal07] definió potencia en términos de complejidad de comprensión del programa resultante:

Definición 2.12 ***Potencia de una ofuscación (Dalla)***
«La potencia de una ofuscación define cuánto es un programa transformado más difícil de comprender que el original.»

Collberg et ál. [CTL97] propusieron una definición que expresaba de un modo formal y preciso los conceptos que se han mencionado:

Definición 2.13 ***Potencia de una ofuscación (Collberg)***
«Supongamos $\mathscr{T}$ como una transformación que conserva el comportamiento, tal que $\mathscr{T}(\mathscr{P}) \rightarrow \mathscr{P}'$ transforma un programa origen $\mathscr{P}$ a un programa $\mathscr{P}'$. Supongamos $\mathscr{E}(\mathscr{P})$ como la complejidad de $\mathscr{P}$. La potencia de $\mathscr{T}$ respecto al programa $\mathscr{P}$, $\mathscr{T}_{pot}(\mathscr{P})$, se define como:

$$\mathscr{T}_{pot} = \frac{\mathscr{E}(\mathscr{P}')}{\mathscr{E}(\mathscr{P}) - 1}$$

siendo $\mathscr{T}$ una transformación potente si y sólo si $\mathscr{T}_{pot}(\mathscr{P}) > 0$»

En otras palabras, cuanto más alta sea la potencia del ofuscador, mejor es el propio ofuscador [Dal07].

Por otro lado, la *resistencia de una transformación* mide cómo de difícil es superar la ofuscación a partir de un *desofuscador* automático. Borello y Mé [BM08] definen resistencia como:

Definición 2.14 ***Resistencia de una ofuscation (Borello y Mé)***
«La resistencia mide la complejidad de la operación inversa (desofuscación) mediante herramientas automáticas (...)»

Además, el *coste de una ofuscación* mide los recursos añadidos al tiempo y espacio de ejecución como consecuencia directa de la ofuscación. Dalla [Dal07] define coste como:

Definición 2.15 ***Coste de una ofuscación (Dalla)***
«El coste de una transformación de ofuscación mide el procesamiento computacional añadido al programa ofuscado respecto del programa original.»

Otra medida importante es el *sigilo de transformación* que mide cómo de oculta está una transformación. Dolz y Parra [DP08] definen sigilo como:

Definición 2.16 ***Sigilo de una ofuscación (Dolz y Parra)***
«Cómo de bien se mezcla el código introducido por el ofuscador con el código original.»

En resumen, las transformaciones realizadas con el fin de ofuscar código ejecutable — a veces malicioso — se suelen evaluar con las métricas que hemos descrito en esta sección. Sin embargo, estas métricas resultan muy difíciles de medir directamente, haciendo que los valores finales acostumbren a ser una mera estimación más que una medida precisa [CTL97, Dal07, DP08, BM08].

2.1.2.2 Métodos de ofuscación de malware

En esta parte, describimos los métodos que se utilizan habitualmente para ofuscar código malicioso. Se puede realizar una clasificación de estos métodos en base al tipo de información que tengan como objetivo [CTL97]. Específicamente, las ofuscaciones más comunes son:

- **Transformaciones de capa.** Estas ofuscaciones actúan en información que no es realmente necesaria para ejecutar un programa. Habitualmente, estos métodos son triviales y reducen la información disponible para un analista de malware. Estas ofuscaciones incluyen borrar los comentarios y cambiar los nombres de las variables. Por ejemplo, si cambiamos los identificadores de un programa (p. ej., métodos y variables) con identificadores sin sentido, la información funcional se reduce drásticamente [Dal07, CTL97].

- **Ofuscaciones de datos.** Estas ofuscaciones actúan en las estructuras de datos de un programa. Se pueden clasificar en función de la operación de transformación que realicen en los datos:

  - *Transformaciones de almacenamiento y codificación.* Estas transformaciones afectan al almacenamiento y también a los métodos utilizados para interpretar los datos [CT98].
  - *Ofuscaciones de agregación.* Estas ofuscaciones alteran el modo en el que los datos se agrupan, haciendo más difícil restaurar la estructura interna de un programa [CTL97, ZTW06].

  - *Ofuscaciones de orden.* Estas transformaciones cambian el modo de ordenar los datos. Por ejemplo, pueden reordenar listas en memoria aplicando una función $f(i)$ que determine la posición del elemento i-ésimo de la lista, mientras que el elemento j-ésimo se suele almacenar en la posición i-ésima de la lista que quedaría libre [CTL97, ZTW06].

- **Transformaciones de control.** Estas ofuscaciones tienen como objetivo confundir o enmarañar el flujo de control de un programa. Afectan a la agregación, al orden e incluso a los cálculos del flujo de control [Dal07].

  - *Transformaciones de agregación.* Estos métodos cambian el modo de agrupar las sentencias de un programa uniendo y separando diferentes fragmentos de código. Por ejemplo, un tipo de ofuscación de agregación en el control es hacer procedimientos *inline*. En otras palabras, reemplazar una llamada a función con las sentencias que componen la función que se llama. Otro tipo de ofuscaciones de agregación en el control son las transformaciones de bucles, como p. ej. *desenrollar el bucle*, que consiste en replicar el cuerpo de bucle una o más veces [Dal07, CTL97].
  - *Ofuscaciones de orden de control.* Estos métodos modifican al azar la situación de algún elemento en el código original (en otras palabras, reordenamiento de las sentencias independientes). Por ejemplo, invertir bucles consiste en reordenar los bucles y recorrerlos al revés [Dal07, CTL97].
  - *Transformaciones de cómputo.* Insertan código, que puede ser redundante o inservible, para ocultar el flujo de control real. Por ejemplo, se observa que hay una fuerte correlación entre la complejidad percibida de un trozo de código y los números de predicados (sentencias) que contiene.

    Estas transformaciones suelen basarse en la existencia de *predicados opacos*, que se definen como: «predicados cuyo valor es conocido previamente por la ofuscación pero que resulta difícil de deducir para un *desofuscador* [Dal07]». Insertando estos predicados opacos, es posible romper el flujo de control original de un programa [MKK07b].

2.1.2.3 Metamorfismo y polimorfismo

El malware ofuscado resultante de aplicar las técnicas mencionadas anteriormente se conoce como *malware metamórfico*. Chouchane y Lakhotia [CL06a] definen el metamorfismo como:

Definición 2.17 ***Metamorfismo (Chouchane y Lakhotia)***
«El metamorfismo es la habilidad del malware de transformar su propio código.»

El objetivo principal del malware metamórfico es cambiar la apariencia del codigo malicioso manteniendo intacta su funcionalidad. Por tanto, la mayoría del malware metamórfico se basa en un comportamiento o funcionalidad maliciosa (*payload*) unido a un motor metamórfico. Este motor cambia su código en cada infección del ejecutable [Dal07, LKK04, KWLP05a, WMCL07]. Cuando se llama al motor, su algoritmo muta el cuerpo del malware (y en algunas ocasiones, incluso el propio cuerpo del motor metamórfico) aplicando transformaciones al mismo.

En contraste, cuando el cuerpo real de un ejecutable malicioso se oculta (o empaqueta), al malware resultante se le llama *polimórfico*. Jacob et ál. definen el polimorfismo como:

Definición 2.18 ***Malware polimórfico (Jacob, Debar y Filiol)***
«El malware polimórfico oculta su cuerpo completo para suprimir la posibilidad de encajar con ninguna firma conocida. Una simple variación en una clave de cifrado modifica totalmente sus secuencias de bytes.»

Por lo tanto, se requiere una rutina de desempaquetado para recuperar el código original y ejecutarlo. Esta rutina también sufre mutaciones para evitar ser reconocida.

Además, los creadores de malware normalmente utilizan empaquetadores (*packers* en inglés) para ocultar sus creaciones. Christodorescu define [CKJ+05] los empaquetadores como:

Definición 2.19 ***Empaquetadores (Christodorescu et ál.)***
«Herramientas para comprimir el código (...)»

Los empaquetadores son aplicaciones informáticas que ocultan un ejecutable reduciendo su tamaño y su comprensibilidad. Al principio, los creadores de malware sólo utilizaban esquemas de cifrado relativamente sencillos y, por cada infección, la única mutación que realizaban era la que

llevaban a cabo con la clave de descifrado, mientras que los algoritmos de cifrado permanecían igual. Más tarde, se empezaron a utilizar técnicas más sofisticadas de cifrado y los ejecutables maliciosos comenzaron a mutar su algoritmo de cifrado [Ste05].

Al nuevo malware creado mediante el uso de diferentes técnicas de ofuscación se le da el nombre de *variante de malware*:

Definición 2.20 ***Variante de malware***

«Una variante de malware es el resultado de aplicar cualquier transformación que se pueda realizar a un ejecutable malicioso. Estas transformaciones incluyen transformaciones de ofuscación y métodos de empaquetado que mantienen la funcionalidad del código malicioso intacta.»

Al grupo de malware que comparte la misma funcionalidad se le llama *familia de malware* [KWLP05a]:

Definición 2.21 ***Familia de malware***

«Definimos una familia de malware como el conjunto de diferentes mutaciones de un ejecutable original y de todas las variantes con la misma funcionalidad.»

Una familia de malware se compone por diversas variantes de malware, que son las versiones ofuscadas (metamórficas y polimórficas) de una funcionalidad original. Una familia generalmente evoluciona; con la adición, en algunas ocasiones, de nuevas funcionalidades. Sin embargo, el objetivo principal de las nuevas variantes de la familia de malware suele mantenerse inmutable [Zha08, Vas07].

2.2 Detección de malware

Un detector de malware se define como la implementación de diversas técnicas que, conjuntamente, pueden detectar algún tipo de malware. La detección de malware ha sido un problema desde la aparición de las primeras amenazas hasta ahora. Sin embargo, no ha sido hasta que los creadores de malware cambiaron sus motivaciones y métodos, cuando la comunidad se ha dado cuenta de la necesidad de realizar mayores avances en este área.

En concreto, Christodorescu [Chr07] define detector de malware como:

Definición 2.22 ***Detector de malware (Christodorescu)***

«Un detector de malware $\mathscr{D}$ es una función cuyo dominio y rango son el conjunto de programas ejecutables $\mathscr{P}$ y el conjunto $\{maligno, benigno\}$, respectivamente. En otras palabras, un detector de malware $\mathscr{D}$ es una función $\mathscr{D} : \mathscr{P} \rightarrow \{maligno, benigno\}$ definido como:

$$\mathscr{D}(\mathscr{P}) = \begin{cases} maligno & si\ \mathscr{D}\ cree\ que\ \mathscr{P}\ es\ maligno \\ benigno & en\ otro\ caso \end{cases}$$

»

2.2.1 Clasificación general de detectores de malware

Los detectores de malware se pueden dividir en dos categorías diferentes: método clásico y método basado en el comportamiento.

El método clásico de firmas o *detector sintáctico de malware* [CKJ+05, Chr07, CJ04] constituye la primera técnica empleada para hacer frente al malware. Este método aún se utiliza en los antivirus comerciales [SPDB09]. Buscan ficheros para encontrar patrones de bytes sospechosos que se han almacenado anteriormente en una base de datos (p. ej., base de datos de firmas [Mor01]). A pesar de que los detectores de malware que emplean el método clásico de firmas son capaces de detectar malware, su efectividad viene limitada sólo a ejecutables maliciosos conocidos previamente, por lo que variantes relativamente simples de malware pueden superar estos métodos.

La *detección de malware basada en el comportamiento* [CKJ+05, Chr07, CJ04] no se basa en simples patrones de bytes sino en complejas estructuras e información estadística para representar un comportamiento.

Aunque pueda parecer que la detección basada en el comportamiento es una reciente actividad investigadora, sus cimientos no son realmente tan nuevos como uno pueda pensar de antemano. Fred Cohen [Coh86, Coh87] defendía que los virus (entendiendo como tal todo tipo de malware) tenían que utilizar los recursos provistos por un sistema, de igual modo que cualquier otro programa que estuviese ejecutándose en el mismo. Siguiendo esta hipótesis, predijo que una vía para modelar el comportamiento de un ejecutable y, por tanto, predecir el comportamiento malicioso del mismo, era equivalente a definir qué era un uso legítimo del sistema y qué no lo era.

Cohen estableció que este problema se podía atacar de dos modos dife-

rentes. Por un lado, se podía modelar el comportamiento del software benigno y, así, medir las desviaciones de estos modelos de comportamiento. La ventaja de este enfoque es que puede afrontar la detección de malware desconocido. El enfoque contrario sería modelar y detectar los comportamientos sospechosos. Este enfoque es el más utilizado dentro de la investigación en detección de malware debido a la necesidad de minimizar la aparición de falsos positivos (en otras palabras, software legítimo incorrectamente detectado como malicioso).

2.2.2 Los métodos basados en firmas

Los detectores actuales de malware utilizan firmas de bytes[1] para identificar malware. Una firma de bytes es un extracto o una función resumen[2] del cuerpo del malware.

```
E8 0000000      call 0h
5B              pop ebx
8D 4B 42        lea ecx, [ebx + 42h ]
51              push ecx
50              push eax
50              push eax
0F01 4C 24 FE   sidt [esp - 02h]
5B              pop ebx
83 C3 1C        add ebx 1Ch
FA              cli
8B 2B           mov ebp, [ebx]
```

Figura 2.1: Código original del virus Chernobyl/CIH.

Por ejemplo, consideremos un extracto de código del virus Chernobyl/CIH[3] (mostrado en la Figura 2.1). La parte izquierda de la Figura 2.1 contiene el código máquina IA-32 mientras que la parte derecha contiene el código ensamblador (código más legible por las personas que el código máquina).

[1] Las firmas de bytes también se conocen como *scan strings*.

[2] Una función de resumen, o función *hash* es cualquier función o procedimiento matemático que convierte una gran cantidad de datos en un único datos, generalmente un número entero, que sirve de clave para los datos originales.

[3] W95.CIH, también conocido como Chernobyl, es un virus parasitario destructivo. Se mantiene residente en memoria e infecta a otros ejecutables que estén usándose [Yam10].

Una de las posibles firmas para Chernobyl/CIH es usar la secuencia de bytes del código máquina (véase la Figura 2.2) como firma. Cuando un fichero contiene esos bytes, el sistema determina que el fichero está infectado con el virus Chernobyl/CIH.

E800 0000 005B 8D4B 4251 5050 0F01 4C24 FE5B 83C3 ACFA 8B2B

Figura 2.2: Firma para Chernobyl/CIH virus.

La mayor ventaja de los métodos basados en firmas es que, debido al uso de técnicas de comparación de patrones, tienen una complejidad controlada y, por tanto, son realmente rápidos. Además, tienen una tasa de falsos positivos baja.

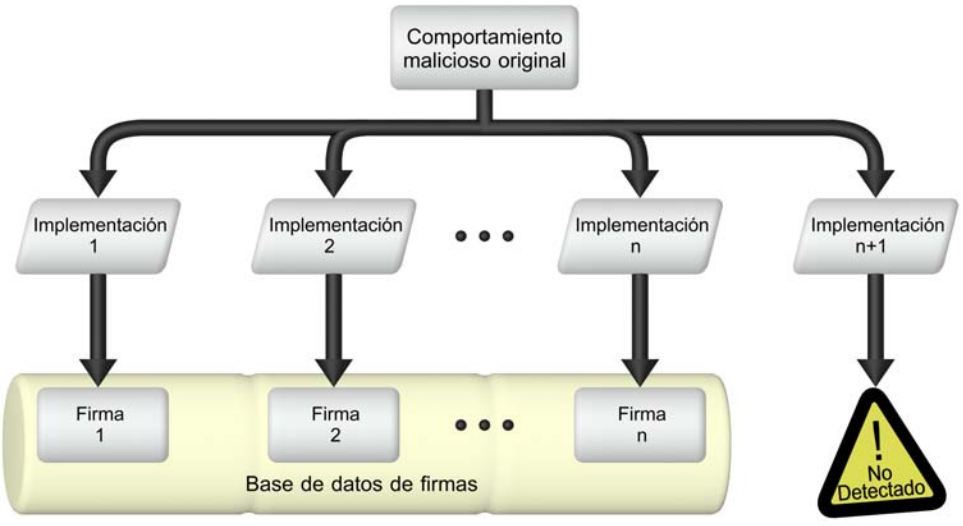

Figura 2.3: Los métodos de firmas no son capaces de detectar variantes de malware (Figura adaptada de la versión de Christodorescu [Chr07]).

No obstante, estos métodos necesitan desarrollar una firma para cada variante de malware (ver Figura 2.3) y, por consiguiente, no detectan variantes ofuscadas de malware [CJ04, SBN+10, SPDB09, JDF08]. Debido a que la generación de firmas se realiza manualmente, este proceso requiere un análisis del código que consume muchos recursos computacionales y, además, una vez generada la firma, ésta se tiene que distribuir a todos los objetivos potenciales [SBN+10, SPDB09]. Mientras tanto, se pueden haber

desarrollado nuevas variantes de malware que eviten la detección mediante esa firma recién creada.

Por otro lado, la detección de malware polimórfico es un problema complejo ya que la detección de los virus que cambian de tamaño es un problema NP-completo[1] [Spi03]. De igual modo, cuando se trata de malware metamórfico, unas buenas reglas de reescritura de código pueden llevar a la indecisión en la detección [Fil07].

2.2.3 Clasificación de técnicas de detección de malware

Las técnicas de detección de malware actuales se pueden dividir en diferentes grupos. Este apartado clasifica las técnicas de detección de malware propuestas tanto por la industria como por la comunidad científica. Las herramientas disponibles para la detección de malware abarcan desde métodos que se centran en la detección de ofuscaciones de malware hasta la detección de anomalías o malware desconocido.

2.2.3.1 Clasificación de Idika et ál.

De acuerdo con Idika et ál. [IM07] (véase la Figura 2.4), los métodos de detección de malware se pueden dividir en dos categorías: detección de malware basada en anomalías y detección de malware basada en firmas. Los métodos basados en anomalías se pueden dividir a su vez en: (i) métodos basados en anomalías *puros* y (ii) métodos basados en especificaciones.

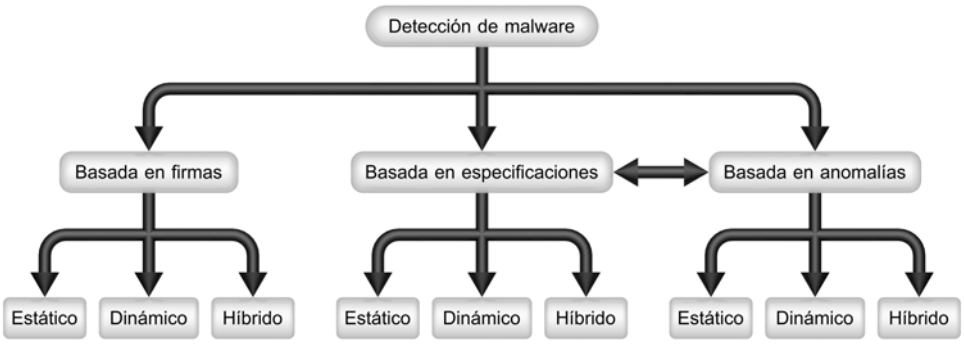

Figura 2.4: Clasificación de Idika et ál. de las técnicas de detección de malware.

[1] Los problemas de decisión más difíciles de resolver.

- **Detección basada en anomalías.** Estas técnicas conforman su conocimiento sobre qué constituye un comportamiento benigno con la intención de decidir cuándo un programa es malicioso y cuándo no.
- **Métodos basados en especificaciones.** Estos enfoques son un tipo especial de detección basada en anomalías. En ellos se genera algún tipo de especificación o conjunto de reglas acerca de lo que es un comportamiento benigno para decidir la legitimidad de un ejecutable.
- **Métodos basados en firmas**[1]**.** Estos métodos utilizan la información sobre el comportamiento del malware para decidir si un ejecutable que se esté inspeccionando es malicioso o no. La caracterización o *firma* del comportamiento malicioso es la clave de la efectividad de los métodos basados en firmas.

La detección de malware también puede clasificarse en otras tres categorías: estática, dinámica e híbrida. El enfoque concreto o análisis de una técnica basada en firmas o anomalías se determina por cómo el sistema recolecta la información necesaria para detectar malware.

El análisis estático recupera información de las propiedades de los programas mientras que el análisis dinámico monitoriza la ejecución de un programa para determinar si es malicioso o no. Por ejemplo, un enfoque estático interpreta información estructural (estructura de datos, flujo de control, secuencias de bytes, etc.) mientras que los enfoques dinámicos recolectan la información en tiempo de ejecución. Además, existen técnicas híbridas que intentan combinar los dos enfoques [RKLC03], pero no es una práctica habitual.

2.2.3.2 Clasificación en base al objetivo del detector

Hemos desarrollado nuestra propia clasificación de técnicas para la detección de malware (véase la Figura 2.5).

En esta clasificación los métodos también pueden ser estáticos, dinámicos o híbridos. Hemos reemplazado el término *métodos basados en firmas*, que puede evocar y confundir al lector con los métodos clásicos de firmas, por *métodos basados en conocimiento* para referirnos a los métodos cuyo objetivo es detectar variantes de malware conocido. Además, debido al gran

[1] La terminología utilizada por Idika et ál. puede llevar al lector incorrectamente a pensar en el método clásico de firmas, sin embargo, estos métodos van más allá del enfoque simple de firmas.

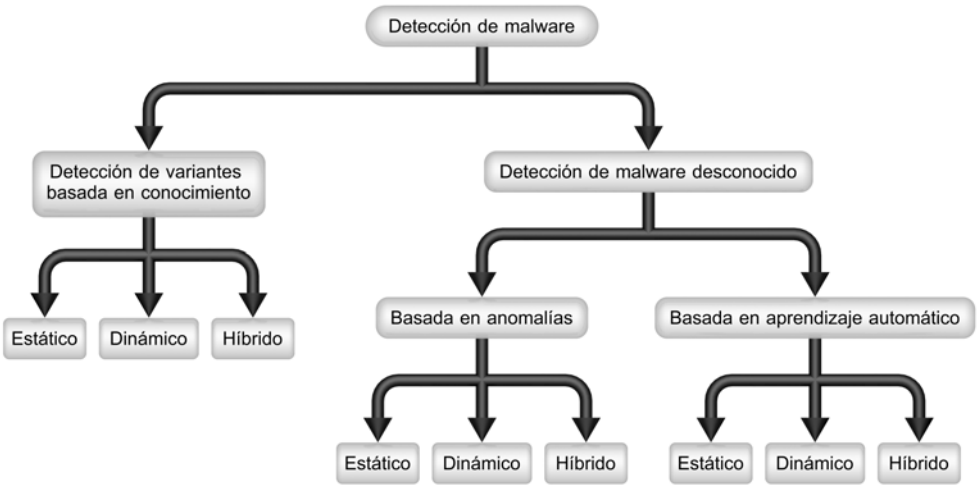

Figura 2.5: Nuestra clasificación de técnicas de detección de malware.

número de métodos de detección de malware basados en técnicas de aprendizaje automático (p. ej., [SEZS01, KM04b, MSF+08, SPDB09, RHW+08, SMEG09]), hemos incluido una categoría exclusiva para este tipo de detección de malware.

De este modo, nuestra clasificación de técnicas de malware se organiza como sigue:

- **Métodos para la detección de variantes basados en el conocimiento.** El objetivo de este tipo de métodos para la detección de malware es detectar variantes de familias de malware conocidas mediante la representación de comportamientos maliciosos. En cierto modo, esta categoría es muy similar a los *métodos basados en firmas* que Idika et ál. proponían en su clasificación.

- **Detección de malware desconocido.** Su objetivo es detectar familias desconocidas de malware (que aún no han sido documentadas, y por tanto, tampoco registradas). Esta categoría puede, a su vez, dividirse en dos sub-categorías diferentes:

  – *Métodos basados en el aprendizaje automático.* Estos métodos utilizan una representación tanto de software benigno como de software malicioso. Su intención es proveer de una base de conocimiento suficiente para entrenar algoritmos de aprendizaje automático [Bis06].

– *Métodos basados en anomalías.* Estos métodos conforman una representación acerca de la legitimidad del software únicamente a través de software legítimo. De esta forma, cualquier desviación de este perfil de comportamiento *normal* se considera, al menos, sospechoso.

Estas categorías para la detección de malware representan y organizan el trabajo realizado en el área de la detección de malware. Las siguientes secciones describirán los enfoques que se han realizado por la comunidad científica en dichas categorías.

2.3 Métodos para la detección de variantes basados en el conocimiento

En esta sección repasaremos los diferentes enfoques que se han propuesto y evaluado para la detección de variantes en base al conocimiento. Como ya hemos mencionado con anterioridad, los métodos basados en el conocimiento utilizan una representación de comportamientos maliciosos con el fin de detectar nuevas variantes de malware conocido.

2.3.1 Métodos estáticos para la detección de variantes

Se ha dedicado mucho esfuerzo recientemente a mejorar los enfoques tradicionales para la detección de variantes de malware. Estos enfoques suelen utilizar el análisis estático del código para detectar ejecutables maliciosos, empleando técnicas semánticas para inferir el comportamiento subyacente de los ejecutables ofuscados.

Christodorescu [CJ04] et ál. investigaron las principales limitaciones de las técnicas de escaneo de malware, demostrando cómo las ofuscaciones más simples, tales como transposición de código, reasignación de registros, inserción de código inútil y sustitución de instrucciones, podían engañar a los antivirus comerciales.

2.3.1.1 Comparación de patrones

La primera generación de detectores de malware utilizaba una representación de los datos e instrucciones en forma de bytes como la *firma* a comparar. De este modo, *string scanning* es el enfoque más sencillo para detectar

malware. Utiliza una secuencia de bytes que es típica de un ejecutable malicioso pero que no se da en el software legítimo. Estas secuencias se organizan en bases de datos que los motores de detección emplean para analizar áreas predefinidas de los ejecutables y de los sistemas, a fin de detectar malware en un tiempo limitado [Szö05].

Cuando los creadores de malware empezaron a renombrar registros e insertar instrucciones basura, las *wildcards* extendieron el método *string scanning* [Szö05]. Las *wildcards* permitían que los argumentos de una función se especificaran mediante el uso de expresiones regulares [KM07]. Además, otra técnica adicional, conocida con el nombre *mismatches* (en castellano, *no parejos*) permitía, a su vez, que un número n de bytes en la cadena de texto tomase cualquier valor.

Smart scanning es un enfoque que mejora las *wildcards* en las cadenas de bytes mediante el uso de expresiones regulares para la instrucción completa y no sólo los parámetros de ésta. Este enfoque permite detectar las instrucciones basura que se hayan insertado en un programa [Szö05].

Kumar y Spafford [KS92] propusieron un detector genérico que detectaba malware basándose en la comparación de expresiones regulares. Para cada fichero a inspeccionar, el sistema de comparación de patrones contrastaba y analizaba todos los ejecutables maliciosos conocidos. Debido a que el sistema estaba implementado para SunOS, no resulta fácil compararlo con otros enfoques.

El mayor inconveniente de los métodos basados en la comparación de patrones, es que operan a nivel de byte y pueden generar falsos positivos debido a que no se conocen las instrucciones reales de los ficheros a inspeccionar. Una mejor opción es desensamblar el ejecutable, identificar las instrucciones y, después, comparar el flujo de instrucciones con expresiones regulares [SF01].

2.3.1.2 **Llamadas a la API o al sistema para la detección de malware**

Existe un gran número de enfoques para la detección de malware que se han basado en las llamadas al sistema o a la API del sistema operativo. Esta elección de características parece apropiada ya que si un fichero se ejecuta dentro de un sistema operativo, parece obvio pensar que también debe utilizar sus recursos [Coh87, Coh84].

Sung et ál. [SXCM04] propusieron un método llamado *Static Analysis for Vicious Executables* (SAVE). Este método representaba los ejecutables como una secuencia de llamadas a la API de Windows donde cada llamada a la

API se representaba, a su vez, por un número de 32 bits. Los 16 bits más significativos correspondían a la categoría o módulo donde se encontraba la función, mientras que el resto correspondían a la posición de la función en una lista de funciones de la API. Para detectar variantes de malware, aplicaban varias medidas de similitud: similitud del coseno, coeficiente de *Jaccard* y la función de correlación de Pearson; entre las representaciones de malware conocido y el número de 32 bits del fichero en inspección. En su validación, compararon SAVE con los detectores comerciales: *Norton*, *McAfee Unix Scanner*, *McAfee*, *Dr. Web*, *Panda*, *Kasperksy*, *F-Secure* y *Anti Ghostbusters*, frente a variantes de los ejecutables *W32.Mydoom*, *W32.Bika*, *W32.Beagle* y *W32.Blaster.Worm*. SAVE fue el único detector capaz de detectar todas las variantes de malware.

Zhang y Reeves [ZR07] introdujeron un nuevo método basado principalmente en los patrones de llamadas a librería o sistema. El objetivo era determinar si dos fragmentos de código eran suficientemente similares como para ser funcionalmente equivalentes. Su experimento demostró que su método poseía tres capacidades importantes: (i) podía distinguir fácilmente entre programas que estaban relacionados, (ii) era capaz de detectar variantes de malware y (iii) podía identificar variaciones en los programas.

Sathyanarayan et ál. [SKB08] diseñaron un modelo de especificación para una clase de malware, en vez de para un único espécimen de malware. Su intención era que esa representación fuese capaz de detectar variantes de malware e incluso nuevo malware que fuese de esa clase. Extrajeron de un ejecutable las llamadas a la API que consideraban críticas para clasificarlo como malware o no. A las funciones extraídas se les realizaba una prueba de probabilidad estadística para determinar la clase de malware.

Sin embargo, estos enfoques no han sido capaces de identificar variantes de malware sin sacrificar la tasa de falsos positivos, debido principalmente a que existen métodos de ofuscación que pueden afectar el número y el orden de las llamadas al sistema [Chr07].

2.3.1.3 **Análisis del flujo de control**

El análisis del flujo de control (en inglés *Control Flow Graph Analysis*, CFG) [All70, Shi88] construye un árbol con la secuencia de instrucciones de un ejecutable. Los orígenes de esta técnica son los sistemas de verificación de software [CJS+05].

Lo et ál. [LLO95] fueron los primeros en utilizar CFG para la detección de malware. Su método partía un programa en bloques para identificar sig-

nos de *tell-tale* (que *contaran historias*) como operaciones que cambian el estado de un programa, p. ej. eventos de acceso a la red u operaciones de fichero.

Crocker y Pozzo [CP89] propusieron un sistema de filtrado de malware que empleaba técnicas de verificación para comprobar si un programa actuaba de un modo malicioso. Su sistema comparaba la especificación de un comportamiento malicioso, representado en forma de árbol de flujo de control, contra el ejecutable para determinar si éste era malware o no.

Bergerson et ál. [BDEK99] presentaron varios métodos para desensamblar ficheros binarios, ayudando en la representación de la ejecución de un programa. Además, mejoraron la partición de programas usando *idioms* (secuencias de instrucciones). Estos bloques se utilizaban para deshacer ciertas transformaciones y, así, recuperar construcciones de alto nivel. Como una extensión a su trabajo, aplicaron los *idioms* para construir un grafo de flujo de control de las llamadas al sistema (sólo las relevantes desde el punto de vista de la seguridad) [BDD+01]. Finalmente, planeaban verificar su enfoque junto con un autómata de seguridad (p. ej. [AN08, SCH08, Cho08]), pero nunca llegaron a presentarlo.

Kinder et ál. [KKSV05] propusieron un paradigma de comprobación de modelos para la detección de malware. Presentaron un lenguaje de especificación llamado *Computation Tree Predicate Logic* , que extendía el conocido como CTL [KLT+01], mediante la adición de cuantificadores sobre variables. El modelo CTPL describía los comportamientos maliciosos, representando secuencias a nivel de ensamblador para intentar detectar más variantes que con una única representación por fichero. No obstante, no se realizó una validación empírica del modelo.

Christodorescu y Jha [CJ03] presentaron *SAFE: Static Analyser for Executables* , cuyo objetivo era el de detectar variantes ofuscadas de malware conocido. SAFE era capaz de enfrentarse a la transposición de código, la inserción de instrucciones basura, la reasignación de registros y la sustitución de instrucciones. Su técnica requería la especificación manual de comportamientos maliciosos. El fichero ejecutable se cargaba a través del componente *Executable Loader*, que utilizaba las herramientas de análisis estático IDA Pro [Dat10a] y Code Surfer [Gra10]. Utilizando la representación de los patrones de comportamiento y el grafo de flujo de control, se anotaban en el grafo las coincidencias entre el ejecutable y el patrón. En función del número de coincidencias se determinaba si un fichero era malicioso o no. SAFE fue capaz en los experimentos de detectar todas las variantes de malware que los propios autores crearon para ello. Sin embargo, SAFE tenía un alto

consumo de recursos computacionales [Chr07] y requería, además, de un proceso manual para la especificación de los patrones de comportamiento.

Más tarde, Christodorescu et ál. [CJS+05] mejoraron SAFE introduciendo patrones semánticos[1]. Afirmaron que la mayor deficiencia de los detectores de malware era que éstos eran puramente sintácticos e ignoraban las semánticas de las instrucciones. Por consiguiente, presentaron un enfoque que incorporaba las semánticas de las instrucciones para detectar malware. Con este fin, formalizaron las semánticas y presentaron un algoritmo que sí las consideraba para complementar a SAFE. En su experimentación, definieron dos plantillas semánticas: un bucle de descifrado y un envío masivo de correos electrónicos. Compararon estas plantillas con 21 gusanos de las siguientes familias: *Netsky*, *Beagle* y *Sober*. El algoritmo no produjo ningún falso positivo y sólo falló al detectar el comportamiento de envío masivo de *Sober*. Sin embargo, tal y como ocurría con SAFE, el sistema tenía un alto consumo de recursos computacionales y requería de un proceso manual para la creación de los patrones, que esta vez eran semánticos [Chr07].

2.3.2 Métodos dinámicos para la detección de variantes

El análisis dinámico ejecuta un programa en un entorno controlado[2] para encontrar el comportamiento real de un ejecutable [IM07]. A pesar de que estos métodos no son exhaustivos[3], tienen la importante ventaja de que las instrucciones que se ejecutan en los entornos controlados son reales. Por consiguiente, el análisis dinámico es inmune a los intentos de ofuscación y no tiene problemas con los ejecutables que se auto-modifican.

Cuando se utilizan este tipo de técnicas, una pregunta habitual suele ser: «¿qué entorno es el más adecuado?». No existe una respuesta correcta a esta cuestión pero, desde luego, se pueden dar ciertas indicaciones que sirvan para tomar una mejor elección.

Por ejemplo, ejecutar el malware directamente en el equipo de un analista es probablemente una opción poco inteligente y que puede resultar completamente desastrosa, pues el equipo estará conectado a Internet y el malware puede escapar a otras máquinas [BMKK06]. Por contra, la utilización de una máquina que en cada ejecución se reinstale no es una opción

[1] Christodorescu et ál. consideraban semántica al hecho de sustituir bloques de instrucciones por intención o comportamiento.

[2] Al entorno controlado se le suele denominar *sandbox*.

[3] Los enfoques dinámicos sólo inspeccionan un flujo de ejecución, omitiendo las ramas alternativas que no se cumplan en un instante determinado.

eficiente debido al gran consumo de procesador que involucra el análisis [BKK06].

Bayer et ál. [BKK06] presentaron *TTAnalyze*, una herramienta para analizar el comportamiento de binarios de Windows dinámicamente. El programa se ejecutaba en un entorno emulado y las acciones importantes para la seguridad se monitorizaban. El sistema almacenaba las llamadas a funciones nativas de Windows que el ejecutable utilizaba. Empleaba QEMU [QEM10] en vez de máquinas virtuales, haciendo más difícil detectar el entorno.

Willems et ál. [WHF07] propusieron *CWSandbox*, que ejecutaba el binario malicioso en un entorno simulado, monitorizaba todas las llamadas al sistema y, automáticamente, generaba una traza de la ejecución.

Dinaburg et ál. [DRSL08] presentaron *Ether*, que se basaba en una novedosa aplicación de extensiones de virtualización como *Intel VT* , en la que el sistema residía fuera del sistema operativo. Por consiguiente, este enfoque no sufría de vulnerabilidades en los componentes software que hicieran posible la detección del entorno controlado. En su experimento con 25.000 muestras recientes de malware, demostraron que el sistema permanecía transparente y que era capaz de enfrentarse a casi todas las variantes de malware. A pesar de todo, Intel VT sufre de algunas limitaciones arquitectónicas que pueden permitir que Ether sea detectado en algunas circunstancias.

Rieck et ál. [RHW+08] propusieron una metodología para detectar dinámicamente variantes de familias de malware. En primer lugar, utilizaban un entorno controlado para monitorizar la ejecución de malware. Seguidamente, etiquetaban el malware con un antivirus. Más tarde, se entrenaba un clasificador de aprendizaje automático. Finalmente, las características se ordenaban utilizando herramientas estadísticas, permitiendo una posterior clasificación del software.

Kolbistsh et ál. [KCK+09] utilizaban un entorno dinámico para modelar el grafo de flujo de control de un ejecutable. El sistema generaba automáticamente modelos que capturaban la información sobre el comportamiento observado en los miembros de una familia de malware.

No obstante, el análisis dinámico tiene ciertas limitaciones: (i) la información que estos enfoques obtienen se limita al entorno controlado. En otras palabras, si el ejecutable se da cuenta de que está en un entorno controlado, puede no mostrar su comportamiento. Además, (ii) la ejecución de malware puede suponer que el malware escape [CDKT09]. Finalmente, (iii) el análisis suele estar limitado a un único flujo de ejecución.

Para resolver este último problema, Moser et ál. [MKK07a] desarrollaron un método que permitía la ejecución de un binario por múltiples caminos de su código. Sin embargo, sólo podían garantizar pequeñas y sencillas resoluciones de variables para cambiar el estado actual del programa y forzar la exploración por otras ramas.

2.3.3 Métodos híbridos para la detección de variantes

Los enfoques híbridos emplean tanto información obtenida de una manera estática como de una manera dinámica. El trabajo de Mori et ál. [MISI06] se puede considerar como uno de los primeros enfoques híbridos para la detección de variantes de malware. Propusieron un método para detectar virus polimórficos en dispositivos móviles. El código se descifraba en un emulador dentro del sistema operativo y, una vez obtenido, se aplicaba análisis estático para identificar llamadas al sistema *peligrosas*.

Perdisci et ál. [PLL08] propusieron un esquema similar, donde el análisis dinámico se empleaba para obtener el código oculto de un ejecutable. Una vez recuperado, realizaban un análisis estático para determinar si el ejecutable era malicioso o no.

A pesar de que los enfoques híbridos pueden evitar las limitaciones del análisis estático y del dinámico, existe muy poca investigación en este área y los enfoques estáticos sólo se utilizan para decidir qué programas necesitan una inspección dinámica posterior, sin que se utilice la información estática para la inspección dinámica [Chr07].

2.3.4 Resumen de los métodos para la detección de variantes

Este capítulo se ha centrado en revisar el trabajo que se ha realizado a lo largo de los años en el área de la detección de variantes de malware. En primer lugar, hemos revisado los enfoques estáticos que no ejecutan el malware. Estos enfoques son los primeros que se utilizaron para la detección de malware y, por tanto, en los que más se ha avanzado. Después, hemos estudiado los enfoques dinámicos que ejecutan el malware en un entorno controlado y la forma en que éstos pueden superar algunas de las limitaciones de los enfoques estáticos. Por último, hemos revisado algunos enfoques que utilizan una combinación de ambos.

En particular, hemos identificado la necesidad de los enfoques actuales de, o bien una mayor profundidad en la detección de variantes, o bien una

mayor celeridad a la hora de realizar el análisis correspondiente. Por lo tanto, consideramos que lograr un equilibrio entre la velocidad y la eficacia de los métodos para la detección de variantes, puede suponer una contribución importante a este área.

2.4 Métodos para la detección de malware desconocido

Detectar malware completamente desconocido se ha convertido en un importante problema debido principalmente al crecimiento exponencial en la creación de nuevo malware [IM07, JDF08, SMEG09]. En este capítulo repasaremos los enfoques cuyo objetivo es precisamente detectar malware desconocido. En concreto, estos métodos se pueden dividir en dos categorías: basados en anomalías y basados en aprendizaje automático.

Los enfoques basados en anomalías utilizan únicamente la información recuperada del software legítimo para generar un perfil acerca de lo que constituye un uso legítimo del sistema. De este modo, cualquier desviación se considera, por lo menos, sospechosa. Por contra, los métodos basados en aprendizaje automático utilizan un conjunto de datos tanto de malware como de software benigno con la intención de ser la base de entrenamiento para métodos de aprendizaje automático capaces de detectar malware completamente desconocido [IM07, JDF08, SMEG09].

2.4.1 Métodos basados en anomalías

Los métodos basados en la detección de anomalías, como se ha mencionado con anterioridad, parten de la idea de que es posible representar lo que es un comportamiento *normal* o benigno para los ejecutables. Así, cualquier desviación de esta representación se considerará maligna. Una de las mayores ventajas que supone la utilización de estos métodos es su capacidad para detectar familias desconocidas de malware [SKF08, BRB+09].

El resto de esta sección revisa estos enfoques empezando por los estáticos para, seguidamente, tratar los enfoques dinámicos y terminar con los enfoques híbridos.

2.4.1.1 Detección estática de malware basada en anomalías

Uno de los primeros intentos para la detección de anomalías se basaba en la comprobación de integridad. Esta técnica compara los contenidos de un

binario ejecutable contra una versión anterior del mismo [Chr07]. Normalmente, esta comparación se realiza sobre los valores de resumen criptográfico entre la versión actual y las versiones anteriores del ejecutable [Coh94]. La eficiencia de este tipo de métodos suele ser extremadamente alta [Rad92]. No obstante, es bastante común que tengan lugar una gran cantidad de falsos positivos debido a cambios en los ejecutables como nuevas actualizaciones o nuevas instalaciones. Otra desventaja de estos modelos es que requieren que la primera función de resumen criptográfico se haya calculado a partir de un programa sin infectar y, por ello, normalmente, sólo se realizan resúmenes parciales (v. gr., en los documentos de Microsoft Office tan sólo se verifican las macros) [Chr07].

Li et ál. [LWSH05] propusieron un enfoque en base a n-gramas[1]. En este enfoque se construía un modelo o conjunto de modelos con los que se intentaban caracterizar diversos tipos de ficheros en función de su distribución de bytes. La principal suposición de los autores era que los programas legítimos estaban compuestos por distribuciones de bytes predecibles para cada uno de sus tipos de fichero. La técnica consiguió unas tasas de detección de entre el 72,1 % y el 94,5 % para ficheros PDF[2] que tenían malware incrustado. El mayor inconveniente del trabajo de Li et ál. era que únicamente evaluaron su método añadiendo el malware al principio o al final del documento.

Debbadi et ál. [DGK+00] introdujeron un enfoque basado en compiladores para asegurar la seguridad del código. La entrada del compilador de certificaciones era el código fuente de la aplicación. El sistema generaba el código binario además de anotaciones que consistían en información relativa al comportamiento del ejecutable. Por ejemplo, un tipo de comportamiento importante sería el acceso o no a información sensible. De esta forma, el verificador se aseguraba de que los tipos de comportamiento en el certificado estuviesen en concordancia con los que manifestaba el ejecutable en inspección. Basándose en políticas de seguridad, el verificador decidía si el ejecutable era seguro. No obstante, no reportaron resultados empíricos en su trabajo.

Cai et ál. [CTG05] utilizaban frecuencias de secuencias de bytes para detectar malware desconocido. Su objetivo era recoger únicamente información de software benigno y medir lo que se desviaban los ejecutables de ese perfil. Concretamente, emplearon unos modelos de probabilidad gaussiana

[1] Un n-grama es una sub-cadena de texto de una cadena de texto mayor. Estas secuencias se calculan solapando una ventana de tamaño fijo n.

[2] PDF es un formato de almacenamiento de documentos.

[BEd08, RW06], que se ajustaron con un análisis de componentes principales[1]. Sin embargo, su enfoque mostraba una alta tasa de falsos positivos (desde un 5 % hasta un 70 %).

Por otro lado, algunos autores han sugerido que si los productores de software firmaran su código de un modo verificable, entonces los usuarios serían capaces de protegerse más fácilmente del malware [Chr07]. Este enfoque es algo diferente ya que la decisión acerca de la validez de un ejecutable la toma una compañía de certificaciones. De este modo, una posible aproximación es la aplicación de criptografía basada en la identidad [Sha84].

Okomoto y Masumoto [OM90] utilizaban este enfoque proponiendo un sistema en el que todos los productores de software se registrasen utilizando una autoridad central para firmar su software con su identidad como clave. Así, los usuarios podrían verificar la identidad del software comprobando la firma de un ejecutable antes de ejecutarlo.

No obstante, la autoridad central, en la que los usuarios y los productores de software tienen que depositar su confianza, se puede considerar como un único punto central de fallo y, por tanto, un ataque a este nodo podría invalidar las claves secretas almacenadas, haciendo no confiable el proceso de verificación y certificación. Para este fin, Harn et ál. [HLY92] presentaron una extensión del trabajo de Okomoto y Masumoto incluyendo más autoridades de certificación.

El lanzamiento del sistema de firmas de código Microsoft Authenticode [Mic09] mostró que los usuarios tendían a ignorar la firma de los ejecutables siempre que se careciese de una infraestructura de clave pública [TSA08].

2.4.1.2 **Detección dinámica de malware basada en anomalías**

Sekar et ál. [SBS99] propusieron un método para monitorizar el comportamiento, donde un programa se traducía de un modo manual a un lenguaje de especificación de auditoria ASL [GZ06]. El código ASL se compilaba entonces a una clase C++. El ejecutable resultante se pasaba al sistema de detección de llamadas al sistema. Cada llamada al sistema se interceptaba en tiempo de ejecución y se enviaba al sistema de detección. El motor de detección comparaba la llamada que se había realizado contra la espe-

[1] El análisis de componentes principales involucra un proceso matemático que transforma un número de variables posiblemente correlacionadas a un número más pequeño de variables no relacionadas llamadas componentes principales [Jol02].

cificación. No obstante, los autores no proporcionaron ninguna validación empírica de su modelo.

Hofmeyr et ál. [HFS98] desarrollaron otra técnica que también monitorizaba llamadas al sistema. En primer lugar, se desarrollaban unos perfiles que representaban el comportamiento normal de los servicios en el sistema. En este caso, se definió *normal* como «secuencias de llamadas al sistema de una longitud corta». Se empleó la distancia de Hamming[1] [BD92] para determinar cuánto una secuencia de llamadas al sistema se parecía a otra. Sin embargo, el método no fue capaz de detectar intrusiones que intentaban explotar vulnerabilidades de varios programas UNIX como `sendmail`, `lpr` y `ftpd`.

Sato et ál. [SOG02] desarrollaron un nuevo método de detección que también se basaba en la frecuencia de llamadas al sistema en la que el perfil era un conjunto de secuencias de llamadas al sistema. El perfil inicial de un proceso se ordenaba en función de la frecuencia de cada una de las llamadas al sistema: cuanto más baja sea la frecuencia de una llamada al sistema, más baja será su posición en el perfil. Así, seleccionaron el número de llamadas al sistema para actuar como el perfil. Después, comparaban perfiles para determinar la legitimidad de una aplicación. Su método fue incapaz de detectar intrusiones en ciertos servicios de Unix.

Wang y Stolfo [WS04] presentaron *PAYL*, una herramienta que calculaba el comportamiento esperado para cada servicio dentro de un sistema. El método creaba una distribución de las frecuencias de bytes, permitiendo desarrollar un modelo de legitimidad para cada servicio del sistema. El modelo requería una fase de entrenamiento, tras la cual, era capaz de medir la distancia Mahalanobis[2] [DMJRM00] entre el modelo y los diversos eventos que ocurrían en los servicios.

Milenkovic et ál. [MMJ05] presentaron una técnica que dedicaba recursos a garantizar que sólo las instrucciones seguras se ejecutaran en un equipo. Su enfoque verificaba la firma de cada bloque de instrucciones en tiempo de ejecución. Además, estas firmas se cifraban utilizando una clave única por cada tipo de procesador. Crearon tres variantes de su método: *SIGT*, *SIGC* y *SIGE*. *SIGT* y *SIGE* verificaban que los bloques de instrucciones básicos en el código correspondían con la firma de estos bloques básicos. La diferencia en los enfoques era cómo se almacenaban las firmas. La premisa

[1] Se llama distancia de Hamming a la efectividad de los bloques de código y depende de la diferencia entre un bloque de código válido y otro.

[2] La distancia Mahalanobis es una forma de medir la distancia entre dos puntos o vectores en el espacio.

era que el código malicioso habría inyectado código que cambiaba el flujo de control de una aplicación y, por tanto, su firma. Milenkovic et ál. determinaron que su técnica era viable y que no consumiría muchos recursos de computación. Sin embargo, no demostraron empíricamente esta afirmación.

Masri et ál. [MP05] describieron una herramienta llamada *Dynamic Information Flow Analysis* (DIFA). DIFA se diseñó específicamente para aplicaciones Java [GJSB05], monitorizando las llamadas a los métodos a partir de los bytecodes[1]. Masri et ál. evaluaron su técnica con un caso de estudio. En éste, insertaron un *exploit*[2] en el servidor de aplicaciones *Apache Tomcat* [WGC+02]. Una vez configurado Tomcat, eran capaces de detectar las peticiones anómalas a partir de gráficos de los perfiles del flujo de información.

2.4.1.3 **Detección híbrida de malware basada en anomalías**

No hay mucha investigación acerca de la combinación de técnicas estáticas y dinámicas para la detección de malware en base a anomalías.

El trabajo de Wang et ál. [WBV+05] para detectar *ghostware*[3] se puede considerar híbrido. Utilizaban dos métodos para analizar el malware, uno dinámico y otro estático para construir perfiles de uso y medir lo que se desviaban los ejecutables de éstos. Aunque los resultados parecían prometedores, es posible que el sistema produjera una tasa alta de falsos positivos.

2.4.2 **Métodos basados en aprendizaje automático**

En los últimos años ha existido una intensa investigación en la dirección del uso de técnicas de aprendizaje automático para la detección de malware desconocido.

Schultz et ál. [SEZS01] fueron los primeros en introducir el aprendizaje automático para la detección de malware desconocido. Utilizaron tres categorías diferentes para extraer características: cabecera del programa, características de cadenas de texto y secuencias de bytes. Estas características las

[1] Un bytecode es un conjunto de instrucciones para la ejecución eficiente de programas interpretados, sin necesidad de compilar un programa en lenguaje máquina.

[2] Un exploit es una pieza de software, un fragmento de datos o una secuencia de comandos que aprovecha un error, fallo o vulnerabilidad, para causar un comportamiento no deseado o imprevisto en los programas informáticos.

[3] Un *ghostware* o *rootkit* es un ejecutable que oculta la existencia de procesos ejecutándose — que pueden ser malware — al sistema operativo. No se le considera malware estrictamente hablando.

utilizaron para entrenar cuatro clasificadores diferentes: un método clásico de firmas; *Ripper* [Coh95], un algoritmo basado en reglas; un clasificador bayesiano ingenuo (*Naïve Bayes*)[1] y un *Multi-Naïve Bayes*. Concluyeron que todos los algoritmos de aprendizaje automático eran, por lo menos, dos veces más precisos que los métodos de firmas.

Abou-Assaleh et ál. [ACKS04a, ACKS04b] introdujeron un método de detección de malware que se basaba en el método de n-gramas común (en inglés *Common N-gram*, CNG) y el algoritmo *k-nearest neighbour* KNN[2]. Para cada clase (código malicioso o benigno) construían un perfil basado en los n-gramas más comunes. Estos perfiles se comparaban con un perfil del fichero a analizar y se elegía la clase que tuviese el perfil más similar. En sus experimentos, obtuvieron los mejores resultados para una longitud de n-grama entre $n = 3$ y $n = 6$.

Kolter et ál. [KM04a] mejoraron los resultados obtenidos previamente por Schultz et ál. [SEZS01] mediante el uso de n-gramas, en vez de secuencias simples de bytes — la diferencia era que los n-gramas se solapan y las secuencias simples no. Su método empleaba varios algoritmos de clasificación y los mejores resultados los dio un árbol de decisión *Boosted*[3].

Más tarde, Kolter junto a Maloof [KM06] publicaron una extensión al trabajo anterior en el que clasificaban el malware en familias basándose en las funciones que se utilizaban en los ejecutables. Las categorías en las que el enfoque tuvo un mejor rendimiento fueron: el envío masivo de e-mails, las puertas traseras y los virus. Intentaron también detectar código malicioso en función de la fecha de obtención de los ejecutables, es decir, entrenaron sus métodos con ficheros previos a julio de 2003 para tratar de detectar malware de agosto de 2004. Los resultados fueron, tal como se esperaba, bastante peores. La relevancia de este estudio es que estableció la verdadera importancia de mantener la base de entrenamiento al día para la detección de malware desconocido.

Henchiri y Japkowicz [HJ06] presentaron un enfoque basado en una selección jerárquica de características. Este modelo hacía plausible elegir los

[1] Un clasificador bayesiano ingenuo es un clasificador probabilístico simple que se basa en aplicar el teorema de Bayes y que supone que todas las variables son independientes [Pea82].

[2] El algoritmo KNN es un método de clasificación que determina la clase de una instancia basándose en los especímenes más cercanos en el espacio de entrenamiento a ésta [FH52].

[3] El *boosting* es una técnica de aprendizaje automático para la combinación de un gran número de clasificadores débiles [Sch03a].

n-gramas del contenido de bytes del fichero binario más relevantes para la representación. Comparando este enfoque Los algoritmos de clasificación mejoraron su rendimiento en comparación con otros enfoques más clásicos de selección de características. Sin embargo, no queda claro si estos mejores resultados son el fruto de su algoritmo de selección, o del número de características final — que no era el mismo que en los enfoques tradicionales.

Santos et ál. [SPDB09] propusieron un enfoque basado en n-gramas en el que se modelaba un ejecutable como una secuencia de n-gramas de las cadenas de texto contenidas en el ejecutable. Se utilizaba un algoritmo KNN modificado con una medida de similitud basada en el número de coincidencias de los n-gramas. Además, los autores introdujeron un método para minimizar los falsos positivos: un parámetro que establecía lo riguroso que era el sistema para clasificar un ejecutable como malware. El sistema era capaz de detectar malware desconocido. No obstante, las características que se utilizaban eran fácilmente modificables por los creadores de malware.

Devesa et ál. [DSC+10] presentaron un sistema en el que las características para el entrenamiento de los modelos de aprendizaje automático se extraían de un modo dinámico. En concreto, el sistema utilizaba *QEMU* [QEM10] y *WINE* [WH10] para monitorizar la ejecución de binarios. Eligieron como características un conjunto de llamadas al sistema que consideraron relevantes desde el punto de vista de la seguridad (v. gr., operaciones de E/S, etc.) para obtener características binarias, donde un 1 indica que esta llamada aparece en la ejecución y un 0 que no aparece. El enfoque obtuvo unas tasas de detección altas aunque, debido a la muestra de malware seleccionada, no queda realmente claro si realmente se estaba detectando malware desconocido o, tan sólo, variantes ofuscadas de malware.

2.4.3 **Resumen de los métodos para la detección de malware desconocido**

A lo largo de las páginas de esta sección hemos estudiado los diferentes métodos para la detección de malware desconocido. Teniendo en consideración el ingente volumen de nuevo malware que las casas antivirus reciben a diario, estos métodos son ahora más relevantes que nunca.

En concreto, en este apartado, hemos visto cómo estos métodos pueden basarse en identificar un perfil *normal* y medir la desviación de un ejecutable respecto a ese perfil en los *enfoques basados en anomalías* o la utilización

de herramientas de inteligencia artificial como el *aprendizaje automático*. La principal ventaja de los sistemas basados en anomalías es, a su vez, su principal desventaja: al medir lo que es *normal* y considerar malicioso todo lo que se desvía del perfil, detecta prácticamente todo el malware pero, por contra, también produce una gran cantidad de falsos positivos. Por contra, los métodos basados en aprendizaje automático, pese a la desventaja clara de necesitar datos etiquetados, parecen más prometedores, consiguiendo un buen equilibrio entre falsos positivos y tasa de detección.

2.5 Aspectos relacionados con la detección de malware

Dentro del área de la detección de malware existen ciertos aspectos que no se pueden clasificar como herramientas de detección, pero que generan inquietudes importantes para el avance del área. En este apartado se tratan: (i) los formalismos teóricos de la detección de malware, (ii) los enfoques para *desempaquetar* ejecutables cifrados y (iii) métodos para la generación automática de firmas.

2.5.1 Formalización teórica

Ha habido un esfuerzo enorme en la literatura científica para formalizar la detección de malware, especialmente, los límites de esta detección. En este ámbito, la comunidad científica ha dedicado un esfuerzo especial a los virus. Cohen [Coh87] fue el primero en definir virus:

Definición 2.23 ***Virus informático (Fred Cohen)***
«Un virus es un programa que puede infectar a otros modificándolos o incluyendo una copia de sí mismo posiblemente evolucionada.»

Además, Cohen [Coh87] también fue el primero en darse cuenta de que la detección de malware era *indecidible*[1]. Para ello, imaginemos un programa $\mathscr{P}$ que es un virus. Cohen estableció que es necesario saber si $\mathscr{P}$ infectaba a otros programas para clasificarlo como virus. Este problema es *indecidible* porque $\mathscr{P}$ puede invocar cualquier proceso de decisión $\mathscr{D}$ acerca de su naturaleza e infectar a otros programas si y sólo si $\mathscr{D}$ determinaba que

[1] Un problema *indecidible* es un problema que no puede decidirse por ningún algoritmo.

$\mathscr{P}$ era benigno (véase la Figura 2.6). Por tanto, generar un procedimiento $\mathscr{D}$ que estuviera al tanto de este tipo de virus *contradictorios* era inviable.

```
// Si un detector de malware 𝒟 determina que el virus 𝒫
   no es malware, entonces 𝒫 muestra su verdadero
   comportamiento.
if 𝒟(𝒫) ≠ malicioso then
    infectar un ejecutable();
    actuar maliciosamente ;
end
```

Figura 2.6: Virus *contradictorio*. El ejecutable $\mathscr{P}$ sólo realiza acciones maliciosas en el caso de que un detector $\mathscr{D}$ determine que no es malicioso. Por consiguiente, el detector $\mathscr{D}$ nunca será capaz de detectar el comportamiento malicioso de $\mathscr{P}$.

Chess y White [CW00] determinaron que era imposible crear un detector perfecto de malware que no obtuviese ningún falso positivo. A pesar de que un detector hipotético clasificase un ejecutable como malicioso, la misma *indecibilidad* puede aparecer para los programas que solamente en ocasiones actúan maliciosamente.

Estas investigaciones han demostrado que, lamentablemente, no es *decidible* si un ejecutable es malicioso o no. No obstante, tal y como se ha visto en los capítulos anteriores, es posible ofrecer una detección que, a la vista de la teoría, jamás será perfecta.

2.5.2 Desempaquetado de ejecutables

Los ejecutables *empaquetados* suponen un problema importante para la detección y el análisis de malware. El malware empaquetado utiliza rutinas de descompresión o descifrado que extraen el *payload* (comportamiento) real del binario y lo ejecutan.

Los ejecutables empaquetados pueden invocar a la rutina de desempaquetado una vez, en cuyo caso todo el código se extrae a memoria en un único paso, o en varios, cuando el algoritmo divide la funcionalidad real del programa en diferentes partes [MCJ07].

Debido a que los *empaquetadores* pueden evadir la detección de los métodos basados en firmas [LH07], los creadores de malware los utilizan para ocultar las intenciones reales de sus programas [GFC08]. El porcentaje de programas empaquetado, tanto maliciosos como benignos, es complicado

de medir correctamente [GFC08]. Sin embargo, se acepta que el porcentaje de nuevo malware que está empaquetado se está incrementando según las casas comerciales de anti-malware [Mas10, Pan10, McA10]: de un 29% en 2003, al 35% en 2005 y a más del 80% en 2007.

Los detectores comerciales de malware utilizan rutinas concretas de desempaquetado para recuperar el código real (en otras palabras, una rutina para cada algoritmo de cifrado o empaquetado). Estas rutinas se utilizan antes de escanear el código malicioso empaquetado [Szö05]. Este enfoque está obviamente limitado a un conjunto fijo de empaquetadores previamente conocidos.

Los sistemas comerciales anti-malware también utilizan técnicas de *Rayos X* para superar enfoques de compresión conocidos o cifrados débiles [PF04]. Estas técnicas no pueden enfrentarse a la cantidad creciente de métodos de cifrado y empaquetado y, por consiguiente, se necesitan enfoques más genéricos para desempaquetar el malware [STF09].

Se han propuesto enfoques tanto estáticos como dinámicos por la comunidad científica [BHB+09]. Coogan et ál. [CDKT09] propusieron analizar el ejecutable mediante análisis estático para extraer la rutina de desempaquetado del ejecutable. Después, empleaban esa rutina para desempaquetar ellos mismos los ejecutables. No obstante, este enfoque solamente funciona para una capa de empaquetado y es insuficiente para malware que emplea varias capas de empaquetado (como *Conficker* [Por09]).

Universal PE Unpacker [Dat10b] y *OllyBonE* [Ste10] son intentos de desarrollar desempaquetadores genéricos. Ambas herramientas identifican ejecutables empaquetados y sus *entry point*[1] mediante ciertas heurísticas. Por ejemplo, *Universal PE Unpacker* funciona bajo la suposición de que la función del sistema `GetProcAdress`[2] se utiliza siempre después de desempaquetar un ejecutable y antes de que el programa llegue al *entry point*. Aunque en ocasiones los sistemas operativos no satisfacen estas suposiciones, generalmente, estos enfoques producen resultados rápidos y correctos. Sin embargo, cualquier malware puede evadir estos enfoques [RHD+06]. Para este fin, se han propuestos otros métodos más verificables y seguros.

PolyUnpack [RHD+06] es un enfoque genérico para extraer el código real sin utilizar ninguna suposición. *PolyUnpack* comienza desensamblando el

[1] Un *entry point* es el punto en el que un ejecutable ha terminado su carga a memoria y procede a ejecutarse.

[2] GetProcAdress es una función de la API del sistema que devuelve la dirección de una función exportada de una variable de una librería enlazada dinámicamente (*Dynamic-link library*, DLL).

ejecutable empaquetado y, así, consigue una visión estática del código. Tras esto, ejecuta el binario en un entorno controlado, instrucción por instrucción, comprobando si la secuencia a ejecutar está o no en el código obtenido previamente. Los resultados mostraron que esta herramienta podía identificar y extraer el código oculto con éxito. No obstante, la fase de desensamblado y la ejecución *paso a paso* incrementan la complejidad computacional del análisis. Además, *PolyUnpack* no puede manejar ejecutables que se hayan empaquetado varias veces.

Renovo [KPY07] utiliza un enfoque similar a *PolyUnpack*. Se diferencia en que *Renovo*, además de monitorizar cada instrucción, comprueba si la memoria era sobrescrita. Si una porción de la memoria que hubiese sido sobrescrita con anterioridad se ejecutaba, entonces se le trataba como otra capa de empaquetado. Este modelo instrumentaba las instrucciones y también incurría en un gasto computacional significante.

OmniUnpack [MCJ07], basado en *OllyBonE* [Ste10], identifica las páginas de memoria de ejecución e invoca a un anti-malware antes de cualquier llamada al sistema *peligrosa*. Incorpora además dos optimizaciones para reducir el número total de escaneos. Por un lado, únicamente invoca al anti-malware si se traspasa el control a una página que haya sido modificada previamente de un modo dinámico en un intervalo comprendido entre la llamada al sistema *peligrosa* actual y la anterior. Por otro lado, si se llama al anti-malware, éste sólo analiza las páginas de memoria que se han modificado desde la llamada al sistema anterior. A pesar de todo, *OmniUnpack* no es ni seguro ni portable. El hecho de analizar segmentos de memoria no es compatible con la mayoría de sistemas anti-malware comerciales.

Eureka [SYS+08] incluye una metodología estadística para evaluar dinámicamente si un proceso ha alcanzado un estado estable para el desempaquetado. *Eureka* aplica un servicio para superar automáticamente algunas estrategias de ofuscación a nivel de código que ocultan las llamadas al sistema. La contribución más importante de *Eureka* es que es el primer sistema de desempaquetado que evalúa al concluir si el código resultante se puede analizar estáticamente o no.

La Tabla 2.1 compara los diferentes enfoques para el desempaquetado de ejecutables. Se han comparado en términos de granularidad, que indica cada cuánto realiza sus transformaciones; velocidad, que indica de una forma cualitativa cómo de rápido es el enfoque en obtener la representación real del código fuente y por último, si los enfoques son resistentes a dos técnicas contra el desempaquetado: anti-debug que consiste en no mostrar su funcionalidad si el ejecutable está siendo depurado y múltiples capas de

Tabla 2.1: Comparación de los diferentes esquemas de desempaquetado dinámico.

Característica	PolyUnpack	Renovo	Omniunpack	Eureka
Granularidad	Instrucción	Instrucción	Página	Llamada al Sistema
Velocidad	Lento	Lento	Rápido	Rápido
Resiliencia a anti-debug	No	No	Sí	Sí
Resiliencia a múltiple empaquetado	No	Sí	Sí	Sí

empaquetado. Los mejores enfoques para el desempaquetado son *OmniUnpack* y *Eureka*. Esto no resulta especialmente sorprendente ya que ambos son los más recientes y han superado las limitaciones de los enfoques anteriores. Sin embargo, existen ciertos retos que aún no han sido superados como la revelación parcial de código [Bur09, GFC08].

2.5.3 Generación automática de firmas

La eficiencia de los métodos clásicos de firmas se han convertido, gracias a su eficiencia, en el enfoque preferido para las casas comerciales de antivirus [SKB08, HVSC07, VH08]. Por tanto, ha existido una ingente cantidad de actividad científica en torno a la generación de firmas (ya sean firmas clásicas u otras más sofisticadas) para detectar malware. La principal dificultad que estos métodos tienen que superar es conseguir encontrar un equilibrio entre la precisión (tasa de detección de malware), la tasa de falsos positivos y la eficiencia de los métodos [Chr07].

Kephart y Arnold [KA94] propusieron uno de los primeros enfoques para la generación automática de firmas. Su propuesta introducía un método para generar *scan strings*. El sistema extraía las secuencias de bytes de las secciones de código de malware conocido y elegía las secuencias que no producían falsos positivos. No obstante, este enfoque se basaba en bytes, que son fácilmente modificables por los creadores de malware [Chr07].

Deng et ál. [DWS+03] desarrollaron un método de generación de firmas que era capaz de producir firmas de longitud variable. Este enfoque genera varias firmas por cada ejecutable malicioso y aplica varios clasificadores

para reducir la tasa de falsos positivos.

Griffin et ál. [GSHC09], del grupo de investigación de la compañía *Symantec* [Sym10], presentaron el sistema *Hancock*. Este sistema generaba automáticamente firmas como una combinación de características: distribución de bytes, cadenas de texto y diversidad de instrucciones. El sistema no podía detectar un alto número de variantes de malware, hecho que los autores justificaban con un supuesto cambio en las estrategias de distribución de malware.

Christodorescu et ál. [CJK07] presentaron un modelo para generar automáticamente especificaciones de comportamientos maliciosos. El enfoque ejecutaba las muestras en un sistema dinámico y recogía las trazas de la ejecución para construir un grafo de dependencias. De este modo, utilizando los grafos de dependencias de ejecutables maliciosos y de software benigno, podían establecer que aquellas funciones que estaban en el malware pero no en el software benigno eran comportamientos maliciosos, siendo la especificación la diferencia entre los grafos. Sin embargo, el enfoque era bastante costoso computacionalmente hablando.

Estos enfoques automáticos que facilita las tareas manuales de los analistas de malware, tienen que ser capaces de aguantar la creciente cantidad de nuevo malware en los próximos años. Por consiguiente, la celeridad de los mismos a la hora de extraer las firmas o perfiles es un factor importante para garantizar la escalabilidad en la detección del malware.

2.6 Sumario

En este capítulo hemos revisado la literatura acerca de la detección de malware. En concreto, hemos comenzado revisando conceptos básicos relativos al malware: sus tipos, la ofuscación del mismo, la detección clásica y sus limitaciones. En este último punto, teniendo en cuenta los avances de los últimos años, hemos propuesto una clasificación de las técnicas de detección de malware basadas en el objetivo de la detección.

Así, hemos estudiado las técnicas que intentan detectar variantes ofuscadas de malware ya conocido. En particular, hemos comprobado que existen métodos rápidos pero que carecen del nivel de detección necesario, mientras que existen enfoques que pecan precisamente de los contrario; es decir, son capaces de ofrecer una gran precisión en la detección pero son costosos, computacionalmente hablando.

Seguidamente, hemos evaluado los diferentes enfoques para la detec-

ción de malware desconocido. En primer lugar, hemos revisado los enfoques basados en anomalías que tratan de identificar desviaciones de un comportamiento *típico*. En segundo lugar, hemos estudiado los enfoques, más actuales, que utilizan aprendizaje automático a fin de detectar nuevo malware. Estos últimos, parecen ser la opción ideal para la detección de nuevo malware.

Finalmente, hemos visto cómo existen temas dentro de la detección de malware que, si bien no son propiamente detección del mismo, están fuertemente ligados al área y, por ende, necesitan la atención de la comunidad científica. En este capítulo hemos revisado los tres temas que entran en esta categoría: formalización teórica del malware, desempaquetado del ejecutable y generación automática de firmas.

«El poeta está muchos rangos por debajo del pintor en la representación de las cosas visibles y muy lejos del músico en la de las cosas invisibles.»

Leonardo Da Vinci (1452-1519)

Capítulo

3

Una representación de ejecutables basada en recuperación de información

Mediante una representación podemos modelar el comportamiento de los ficheros ejecutables. Los métodos que se utilizan en las casas comerciales para representar este tipo de malware generalmente apoyan su núcleo en una base de datos de firmas clásica [Mor01]. Como los analistas de malware escriben estas firmas de un modo manual, se forma un *cuello de botella* debido a la gran cantidad de malware nuevo [Kar10].

Otro tipo de firma que se suele emplear, es utilizar sencillas funciones de resumen criptográfico que calculan una clave única para el ejecutable malicioso [Szö05]. La principal ventaja de este último método es que estas funciones son realmente rápidas extrayendo las firmas de los ejecutables.

No obstante, el cambio de motivación de los creadores de malware que, actualmente, buscan dinero en vez de fama y gloria, ha hecho que estos ejecutables sean más difíciles de detectar que nunca. Estos hechos han originado una situación en la que los métodos de detección clásicos, en los que se basan los antivirus comerciales, ya no son completamente confiables

debido a que sólo pueden detectar malware que haya sido previamente almacenado y documentado.

A la luz de estos acontecimientos, proponemos un nuevo método para la representación de ejecutables. Utilizamos una representación basada en los códigos operacionales de lenguaje máquina (*opcodes*). Un estudio relativamente reciente [Bil07a, Bil07b], analizó estadísticamente la habilidad de los *opcodes* para la detección de malware mostrando su alta capacidad de predicción para determinar si un ejecutable es malware. Por consiguiente, presentamos un nuevo método basado en la frecuencia de aparición de secuencias de *opcodes* para construir un enfoque de recuperación de información (IR, del inglés *Information Retrieval*) [MS83].

Lo que resta de capítulo está organizado como sigue. La sección 3.1 explica más detalladamente por qué los *opcodes* pueden clasificar malware. La sección 3.2 describe de un modo detallado cómo hemos llevado a cabo la representación de los ejecutables. La sección 3.3 describe la evaluación del método y presenta los resultados. La sección 3.4 debate el método propuesto y posibles modos de mejorarlo. Finalmente, la sección 3.5 resume las principales contribuciones de este capítulo.

3.1 Capacidades de los *opcodes* para la detección de malware

Esta sección justifica la elección de los *opcodes* como características para la representación de ejecutables.

En primer lugar, introducimos brevemente cómo funciona el lenguaje máquina y damos algunas nociones sobre los parámetros que pueden acompañar a los *opcodes*.

Seguidamente, describimos los aspectos más relevante del trabajo anterior de Bilar [Bil07a, Bil07b] que descubrió que los *opcodes* servían para predecir la clase de un ejecutable.

3.1.1 Descripción del lenguaje máquina

Un *opcode* es la parte de una instrucción de lenguaje máquina (representados aquí a través de mnemónicos de *código ensamblador*). Su especificación y formato recaen en la arquitectura del conjunto de instrucciones del procesador de un equipo [Swa10]. En esta investigación nos vamos a centrar

en el conjunto de instrucciones de *Intel-32* porque se trata de la arquitectura más utilizada por los ejecutables maliciosos. Sin embargo, el enfoque que hemos propuesto es fácilmente escalable a cualquier otro conjunto de instrucciones de lenguaje máquina.

```
call  0h
pop   ebx
lea   ecx, [ebx + 42h]
```

Figura 3.1: Extracto de código máquina desensamblado.

La mayoría de instrucciones en código máquina están compuestas de un operador *opcode* que indica la acción a realizar y uno o más operandos, que determina cómo realizar la acción. Por ejemplo, en el extracto de código mostrado en la Figura 3.1 hay tres instrucciones máquina. La primera, `call 0h`, está compuesta de un *opcode* `call` y un operando `0h`. Esta instrucción — como su nombre sugiere — llama a una sub-rutina o función alojada en la posición de memoria `0h`. La instrucción `pop ebx` extrae el contenido del registro `ebx`. Finalmente, `lea ecx, [ebx + 42h]` almacena la dirección efectiva para la localización de memoria apuntada por el contenido de `ebx + 42h`.

Podemos definir una instrucción como un conjunto formado por un código operacional y de 0 a n operandos. Formalmente, supongamos una instrucción como una bi-tupla $\mathscr{I} = \{o, \mathscr{P}\}$ donde o se refiere al *opcode* y $\mathscr{P}$ se refiere al operando. El operando $\mathscr{P}$ se define como una n-tupla de operandos, $\mathscr{P} = \{p_1, p_2, \ldots, p_{n-1}, p_n\}$ y $n \in \mathbb{N}$. Además, el conjunto de operandos dentro de una instrucción $\mathscr{I}$ puede ser dividido en diferentes conjuntos: siendo $\mathscr{E} = \{i_1, i_2, \ldots, i_{m-1}, i_m\}$ el conjunto de los operandos de entrada y $\mathscr{S} = \{o_1, o_2, \ldots, o_{\ell-1}, o_\ell\}$ el conjunto de datos de salida, $\ell \in \mathbb{N}$ y $m \in \mathbb{N}$.

Los operadores se pueden dividir de diferentes modos. Por ejemplo, hay *opcodes* que realizan operaciones aritméticas, *opcodes* para el manejo de datos, *opcodes* para ejecutar operaciones lógicas, para controlar el flujo de control, etc. Por otro lado, los operandos pueden ser valores de registro, valores en la pila, valores de memoria, puertos de entrada y salida, etc.

Christodorescu [Chr07] clasificó detalladamente los operandos del código máquina. Estableció que existían un gran número de expresiones a las que un operando podía pertenecer (véase la Tabla 3.1). Concretamente, Christodorescu separó las localizaciones de datos y almacenamiento y proveyó de un esquema de reemplazo para los operandos que propuso. Estas

Tabla 3.1: Tipos de operandos según Christodorescu.

Tipo de dato	Expresión
byte unsigned int	uint(0:0:8)
word unsigned int	uint(0:0:16)
doubleword unsigned int	uint(0:0:32)
quadword unsigned int	uint(0:0:64)
double quadword unsigned int	uint(0:0:128)
byte signed int	uint(0:1:7)
word signed int	uint(0:1:15)
doubleword signed int	uint(0:1:31)
quadword signed int	uint(0:1:63)
double quadword signed int	uint(0:1:127)
single precision float	uint(0:1:31)
double precision float	uint(0:1:63)
near pointer	$\perp(32)$
far pointer (logical address)	uint(0:0:16) $\times$ uint(0:0:32) $\times \perp(32)$
eax, ebx, ecx, edx	$\perp(32)$
esi, edi, ebp, esp	$\perp(32)$
eip	uint(0:1:31)
cs, ds, ss, es, fs, gs	$\perp(16)$
ax, bx, cx, dx	$\perp(16)$
al, bl, cl, dl	$\perp(8)$
ah, bh, ch, dh	$\perp(8)$

Tabla 3.2: Tipos de operando genéricos.

Operando	Tipo
eax, ebx, ecx, edx, esi, edi, ebp, esp, eip, cs, ds, ss, es, fs, gs, ax, bx, cx, dx, al, bl, cl, dl, ah, bh, ch, dh	*Registro*
number+ h	*Constante*
[expresión]	*Localización de memoria*

especificaciones se llamaban *patrones de abstracción* y le permitieron desarrollar un lenguaje de especificación para crear un detector de malware *semántico* [CJS+05, PCJD08].

Podemos dividir, no obstante, los operandos en tres tipos más genéricos de parámetros: *constantes*, *localizaciones de memoria* o *registros*. La Tabla 3.2 muestra las expresiones y cómo se clasificarían de este modo genérico.

En este tipo de clasificación de los parámetros *Expresión* se refiere a cualquier tipo de operación que conlleve *registros*, *constantes* o, incluso, otras expresiones. Estos tipos de operando son una simplificación de la clasificación de Christodorescu [Chr07] que era parte de un lenguaje de especificación completo que le posibilitó realizar un detector de malware sensible a la semántica. Con esta simplificación, se pueden entender mejor los diferentes tipos de expresiones provistos por el lenguaje máquina.

3.1.2 *Opcodes* como características relevantes para la detección de malware

Nuestro método de representación se basa en la suposición de que no se necesita información de los operandos de las instrucciones de código máquina para detectar malware. En otras palabras, que los *opcodes* por sí mismos son capaces de predecir si un ejecutable es malicioso o no, tal y como se extrae del trabajo previo de Bilar [Bil07a, Bil07b].

Bilar, comparó en su trabajo las distribuciones estadísticas de *opcodes* dentro del software malicioso y del benigno. De esta forma, fue capaz de realizar una evaluación preliminar de las capacidades de los *opcodes* para detectar malware moderno (que puede ser tanto polimórfico como metamórfico).

Para validar su hipótesis inicial: «*los opcodes son unos indicadores significantes para la detección de malware*», llevó a cabo la siguiente metodología:

1. **Extracción de los *opcodes*.** Utilizando un conjunto de malware, extrajo los *opcodes* que se utilizaban mediante el código desensamblado obtenido con *Ida Pro* [Dat10a] y contabilizó el número de veces que éstos aparecían en el ejecutable utilizando el *plugin InstructionCounter* [Ins10]. Siguiendo el mismo procedimiento extrajo los *opcodes* del conjunto de datos benigno.

2. **Análisis estadístico.** Seguidamente, utilizando los códigos operacio-

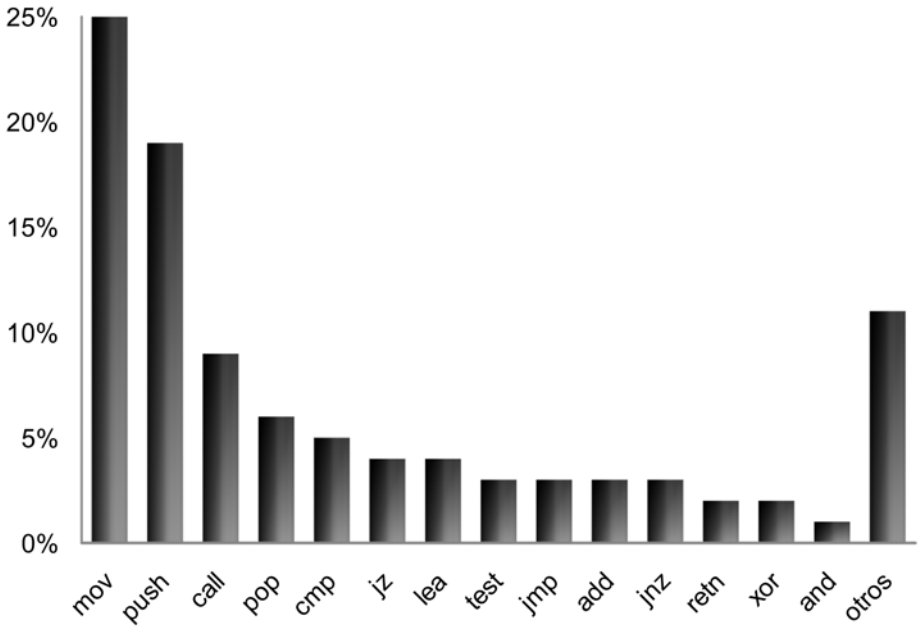

Figura 3.2: Distribución de *opcodes* para el conjunto de software benigno.

nales que había extraído, ejecutó varias pruebas estadísticas. Las preguntas que buscaba contestar eran:

- *¿Existe una diferencia estadísticamente significativa en la frecuencia de aparición de los opcodes entre el malware y el código benigno?*
- *Si existe una diferencia significativa, ¿qué opcode u opcodes son los responsables?*
- *¿Cómo de fuerte es la correlación entre la clase de software y los opcodes?*

Bilar empleó el procedimiento de χ^2 de Pearson [OF79] para la primera pregunta, el procedimiento *análisis de residuos estandarizados post-hoc* (STAR) [Hab73] para la segunda y, para la última pregunta el *test* de χ^2 [MS09].

En cuanto a la extracción de *opcodes*, la Figura 3.2 muestra los *opcodes* más frecuentes para el conjunto de software benigno. Por otro lado, la Figura 3.3 representa la distribución de *opcodes* para el conjunto de malware.

Podemos apreciar que los 5 *opcodes* más utilizados son los mismos para malware y para software benigno: `mov`, `push`, `call`, `pop` y `cmp` aunque

también se puede observar que existen ciertos cambios en el orden de los *opcodes* de menor frecuencia de utilización.

En particular, el código benigno generó 1,5 millones de *opcodes* de los que 192 eran diferentes. De éstos, 72 representaban el 98,8 % de los *opcodes*, 14 el 90 % y, los 5 más utilizados, cerca del 64 % de los mismos.

De igual manera, el malware generó 665.000 *opcodes*, de los cuales 141 eran diferentes, incluyendo 2 que no están documentados: `icebp` y `sa1c`. De éstos, 60 representaban más del 98,8 %, 14 el 92 % y, los 5 más utilizados, el 65 % del total.

La Tabla 3.3 muestra una distribución granular de los *opcodes* entre las diferentes clases de interés: código benigno y 7 clases diferentes de malware. Entre el software malicioso encontramos 2 tipos diferentes de *rootkits*[1]: de usuario (*RK User*) y de sistema operativo (*RK SO*). Además, había herramientas (*Tools*), *bots*, caballos de Troya (*Trojan*), virus y gusanos. Una vez extraída la frecuencia de los *opcodes*, se realizaron las pruebas estadísticas tanto para los *opcodes* más frecuentes como para los menos.

En cuanto a los *opcodes* más frecuentes, cuando se comparó su distri-

[1]Un *rootkit* es un software que consiste de uno o varios programas que ocultan la ejecución de programas maliciosos [HB05].

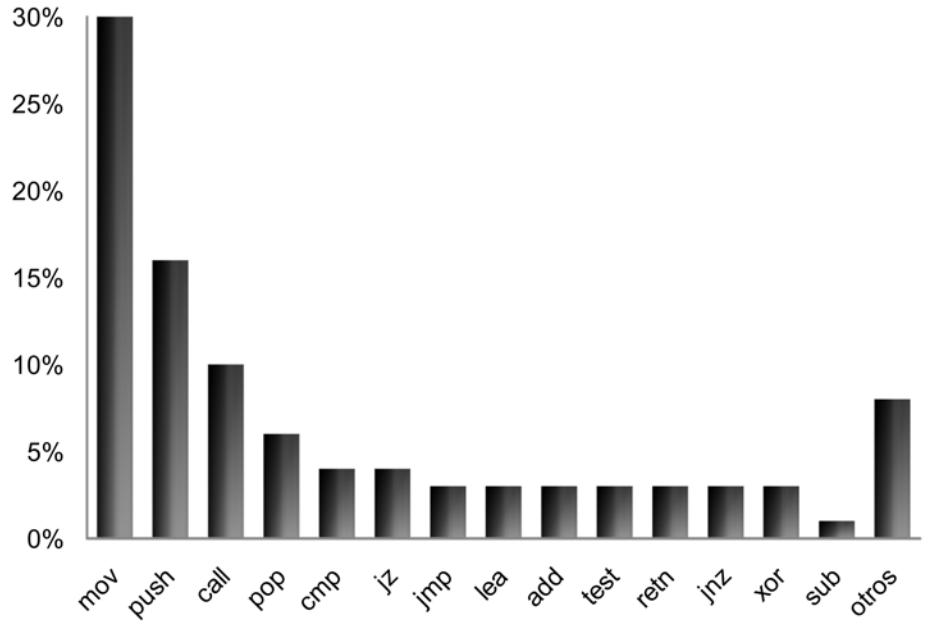

Figura 3.3: Distribución de *opcodes* para el conjunto de malware.

bución entre los grupos de maliciosos y benignos, 1/3 obtuvieron frecuencias similares, 1/3 más altas y otro 1/3 más bajas. La Tabla 3.4 indica que los *opcodes* más frecuentes no parecen ser un fuerte indicador para detectar malware debido a su variación de Cramer, que indica cuánto de una asociación estadística se puede explicar sin necesidad de utilizar factores externos [CL03].

Tabla 3.3: Comparación de los 14 *opcodes* más frecuentes (%).

Op.	Benigno	RK User	RK SO	Tool	Bot	Trojan	Virus	Gusano
mov	25,3	37,0	29,0	25,4	34,6	30,5	16,1	22,2
push	19,5	15,6	16,6	19,0	14,1	15,4	22,7	20,7
call	8,7	5,5	8,9	8,2	11,0	10,0	9,1	8,7
pop	6,3	2,7	5,1	5,9	6,8	7,3	7,0	6,2
cmp	5,1	6,4	4,9	5,3	3,6	3,6	5,9	5,0
jz	4,3	3,3	3,9	4,3	3,3	3,5	4,4	4,0
lea	3,9	1,8	3,3	3,1	2,6	2,7	5,5	4,2
test	3,2	1,8	3,2	3,7	2,6	3,4	3,1	3,0
jmp	3,0	4,1	3,8	3,4	3,0	3,4	2,7	4,5
add	3,0	5,8	3,7	3,4	2,5	3,0	3,5	3,0
jnz	2,6	3,7	3,1	3,4	2,2	2,6	3,2	3,2
retn	2,2	1,7	2,3	2,9	3,0	3,2	2,0	2,3
xor	1,9	1,1	2,3	2,1	3,2	2,7	2,1	2,3

Tabla 3.4: Variación de Cramer en los *opcodes* más frecuentes.

RK SO	RK User	Tool	Bot	Trojan	Virus	Gusanos
10,3	6,1	4,0	15,0	9,5	5,6	5,2

Los *opcodes* menos utilizados (mostrados en la Tabla 3.5) no obtuvieron prácticamente frecuencia alguna: su frecuencia es cero para casi todas las pruebas. Estos *opcodes* se seleccionaron uniformemente al azar entre la población de *opcodes* con una frecuencia menor del 0,2%.

Cuando se comparó la distribución entre la clase benigna de software y las múltiples clases maliciosas, el 70% obtuvo frecuencias similares, un 30% frecuencias más altas y un 10% más bajas. Las variaciones de Cramer, mostradas en la Tabla 3.6, indican que los *opcodes* que son menos utilizados

Tabla 3.5: Comparación de los *opcodes* menos utilizados (en partes por millón).

Op.	Benigno	RK User	RK SO	Tool	Bot	Trojan	Virus	Gusano
bt	30	0	34	47	70	83	0	118
fdvip	37	0	0	35	52	52	0	59
fild	357	0	45	0	133	115	0	438
fstcw	11	0	0	0	22	21	0	12
imul	1182	1629	1849	708	726	406	755	1126
int	25	4028	981	921	0	0	108	0
nop	216	136	101	71	7	42	647	83
pushf	116	0	11	59	0	0	54	12
rdtsc	12	0	0	0	11	0	108	0
sbb	1078	588	1330	1523	431	458	1133	782
setb	6	0	68	12	22	52	0	24
setle	20	0	0	0	0	21	0	0
shld	22	0	45	35	4	0	54	24
std	20	272	56	35	48	31	0	95

son un mejor indicador ya que con ellos se puede explicar entre un 12% y un 63% de la variación obtenida.

En resumen, la distribución de las frecuencias de los *opcodes* contenidos en el malware se desvía significativamente de la del software benigno. Los *opcodes* menos utilizados explican mejor la variación que los *opcodes* que se utilizan más frecuentemente. El estudio de Bilar no fue más allá y no desarrolló un detector de malware utilizando estos importantes descubrimientos.

Tabla 3.6: Variación de Cramer en los *opcodes* menos frecuentes.

RK SO	RK User	Tool	Bot	Trojan	Virus	Gusanos
63	36	42	17	16	10	12

3.2 Representación de ejecutables

Aplicamos *recuperación de información* (IR) para representar ejecutables. IR es un área de investigación muy activa dentro de las ciencias de la computación.

El objetivo principal de este área es la búsqueda de documentos o de información dentro de éstos, tanto en bases de datos como en la Internet [FB92].

3.2.1 Definición formal de recuperación de información

Baeza-Yates et ál. [BR99] definen modelo de IR como una 4-tupla de la forma $[\mathscr{D}, \mathscr{Q}, \mathscr{F}, \mathscr{R}(q_i, d_j)]$ donde:

- $\mathscr{D}$, es el conjunto de representación de documentos;
- $\mathscr{Q}$, es el conjunto de las búsquedas del usuario;
- $\mathscr{F}$, es un marco de trabajo para modelar documentos, búsquedas y sus relaciones;
- $\mathscr{R}(q_i, d_j)$ es una función de ordenamiento que asocia un número real con una búsqueda q_i, $(q_i \in \mathscr{Q})$ y la representación de los documentos d_j, $(d_j \in \mathscr{D})$. Esta función también se llama *función de similitud*.

$\mathscr{D}$ es un conjunto de documentos d, $\{d : \{t_1, t_2, \ldots, t_n\}\}$, cada uno de los cuales está compuesto por un número n de términos t. Consideremos $w_{i,j}$ como un peso para un término dado d_j, tal que si t_i no existe en d_j, entonces $w_{i,j} = 0$. Por tanto, un documento puede representarse como un vector, empezando por el origen del eje de coordenadas cartesianas, de términos que actúan de *índice* tal que $\vec{d_j} = (w_{1,j}, w_{2,j}, \ldots, w_{n,j})$.

3.2.2 Modelo espacio vectorial

El modelo de espacio vectorial (VSM de su denominación inglesa *Vector Space Model*) [Sal89, MS83] es un modelo algebraico utilizado para la recuperación, filtrado, indexación y ordenamiento de información. Este modelo representa documentos en lenguaje natural de un modo matemático utilizando vectores en un espacio multi-dimensional. El espacio está formado únicamente por ejes positivos basados en un origen.

Los documentos se representan formalmente en una matriz *término-por-documento* donde el elemento (i, j) ilustra la relación entre el término i y el documento j. En otras palabras, esta asociación refleja la ocurrencia del término i en el documento j. Los términos pueden ser diferentes unidades textuales (p. ej., una palabra o una frase) y pueden ser ponderados individualmente. Esta ponderación permite que un término, perteneciente a un

documento, se convierta en más o menos relevante para un documento en concreto o para toda la colección de documentos al completo.

En los *modelos booleanos de vectores*, cada una de las coordenadas del vector de un documento es cero (cuando el término no se encuentra en el documento) o uno (cuando el término está presente en el documento). La ponderación de términos se considera como un refinamiento de los modelos booleanos que tiene en cuenta la frecuencia de aparición de los términos, la frecuencia total de un término en el conjunto de datos completo y la localización de cada ocurrencia (v. gr., en el título, en el resumen, en el cuerpo).

Un modo común para la ponderación es el TF-IDF (de las voces inglesas *Term Frequency-Inverse Document Frequency*) donde el peso del término i en el documento j viene dado por:

$$weight(i,j) = tf_{i,j} \cdot idf_i \tag{3.1}$$

donde la frecuencia del término (*term frequency*) $tf_{i,j}$ [MS83] se define como:

$$tf_{i,j} = \frac{n_{i,j}}{\sum_k n_{k,j}} \tag{3.2}$$

donde $n_{i,j}$ es el número de veces que el término t_i aparece en el documento d_j y $\sum_k n_{k,j}$ es el número total de términos que existen dentro del documento d_j. La frecuencia de documento inversa (*inverse document frequency*) idf_i se define como:

$$idf_i = log\left(\frac{|\mathscr{D}|}{|\mathscr{D} : t_i \in d|}\right) \tag{3.3}$$

donde $|\mathscr{D}|$ es número total de documentos y $|\mathscr{D} : t_i \in d|$ es el número de documentos en los que aparece el término t_i.

A pesar de que existen más variaciones de la fórmula TF-IDF, todas ellas se basan en que la ponderación de términos indica la relevancia de un término para un documento y una colección de documentos dados. Además, se asume que la importancia de un término en un documento se refleja por su frecuencia de aparición en los documentos.

Los vectores de términos se pueden representar en un espacio multidimensional, donde cada término es una dimensión y cada documento un vector en el espacio, con componentes booleanas o ponderadas indicando la ocurrencia de los términos en el documento concreto. Es muy común en categorización de textos utilizar este tipo de modelo de espacio vectorial [Ber03, BC07].

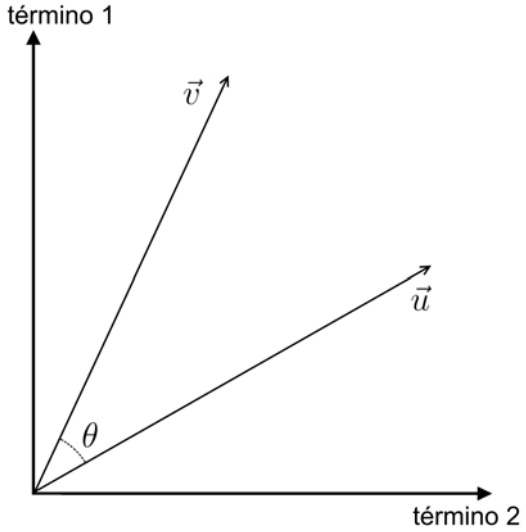

Figura 3.4: Ejemplo de la aplicación de la *similitud del coseno* en un espacio bidimensional.

La similitud en el modelo de espacio vectorial se determina empleando coeficientes asociativos basados en el producto escalar de un vector que representa un documento y un vector que representa una búsqueda, donde el solapamiento de términos (realmente ponderaciones de términos) indica similitud. En concreto, la similitud entre el vector de búsqueda $\vec{v}$ y un vector de documento $\vec{u}$ se computa utilizando una medida de similitud, como la *similitud del coseno* (representada en la Figura 3.4):

$$sim(\vec{v},\vec{u}) = \cos(\theta) = \frac{\vec{v}\cdot\vec{u}}{\|\vec{v}\|\cdot\|\vec{u}\|} \tag{3.4}$$

donde $\vec{v}\cdot\vec{u}$ es el producto escalar de $\vec{v}$ y $\vec{u}$ mientras que $\|\vec{v}\|\cdot\|\vec{u}\|$ es el producto de los módulos de $\vec{v}$ y $\vec{u}$.

Así, tenemos que $\cos(\vec{v},\vec{u})$ define un ángulo θ con $\cos(\theta) = \cos(\vec{v},\vec{u})$ como el ángulo entre $\vec{v}$ y $\vec{u}$. Los términos similares tienden a aparecer en documentos similares y el ángulo entre éstos es pequeño. Por consiguiente, la similitud del coseno está cerca de 1. Por contra, en documentos que tienen poco o nada en común, no aparecen los mismos términos y, por tanto, la similitud del coseno tiende a 0. Esta medida de similitud es una de las más utilizadas en IR [BR99].

3.2.3 Encontrando la relevancia de los *opcodes*

Como primer paso en la representación de ejecutables para detectar malware, introducimos un método capaz de obtener la importancia de cada *opcode*. Debido a que algunos *opcodes* tienen una frecuencia de aparición alta tanto en malware como en software legítimo (como `mov` o `push`), pueden distorsionar los resultados de un detector de malware. Por ende, proponemos un modo de evitar este fenómeno y ofrecer a cada *opcode* la relevancia que realmente tiene.

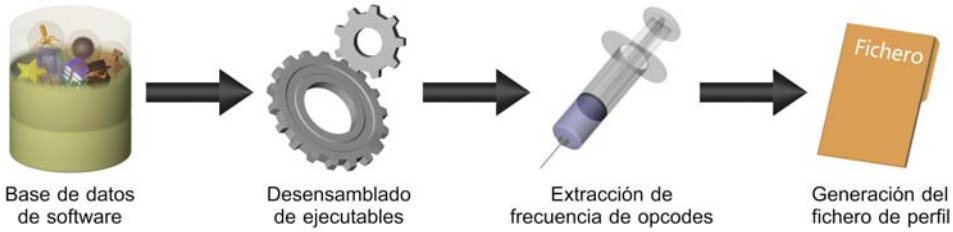

Figura 3.5: Métodología para obtener la relevancia de los *opcodes*.

Como se ha mencionado, Bilar [Bil07a, Bil07b] investigó sobre las capacidades de los *opcodes* para detectar malware. El estudio concluyó que había diferencias estadísticamente relevantes entre los *opcodes* del software malicioso y del benigno, y que los *opcodes* menos utilizados eran un mejor indicador que los más comunes.

Extendiendo este enfoque, hemos creado un método capaz de calcular la relevancia de cada *opcode* individual. Así, obtenemos un peso para cada *opcode*. Esta ponderación representa la relevancia del *opcode* concreto para clasificar entre malware y software benigno, teniendo en cuenta su ratio de aparición en el malware o en el software benigno (la Figura 3.5 muestra una visión general del método).

De este modo, hemos recogido malware del sitio web *VxHeavens* [VX10] para formar un conjunto de 13.189 ejecutables maliciosos. Se ha elegido *VxHeavens* tanto porque es un sitio conocido como por que es público. Este conjunto de datos contiene únicamente ejecutables *Portable Executable*[1] y está formado por diferentes tipos de software malicioso (p .ej., virus, caballos de Troya, gusanos y *spyware*). Para el conjunto de datos de software benigno, hemos recopilado 13.000 ejecutables de nuestros ordenadores. Este conjunto benigno incluye procesadores de texto, herramientas de dibujo

[1] El formato *Portable Executable* es un formato de fichero para ejecutables, código objeto y DLLs, utilizado en los sistemas operativos *Microsoft Windows*.

o visores de PDF. Confirmamos que los ejecutables benignos no estuviesen infectados, porque cualquier infección distorsionaría los resultados. Para ello, analizamos los ficheros benignos mediante *Eset Antivirus* [Ese10].

El método para calcular la relevancia de los *opcodes* sigue los siguientes pasos:

1. **Desensamblar los ejecutables.** Utilizamos el desensamblador *New-Basic Assembler* [New10] como la herramienta para obtener el código ensamblador.

2. **Generar el fichero de perfiles de opcodes.** Utilizando los ficheros en ensamblador, construimos los ficheros de perfiles de *opcodes*. Cada fichero contiene una lista con cada *opcode* y el número de veces que éste aparece tanto en los ejecutables maliciosos como en los benignos.

3. **Cálculo de la relevancia de los *opcodes*.** Calculamos la relevancia de cada *opcode* basándonos en la frecuencia con la que aparece en cada conjunto de datos. Utilizamos la medida *información mutua* (mostrada en la Ecuación 3.5) para medir la dependencia estadística entre dos variables [PLD05]:

$$I(X;Y) = \sum_{y \epsilon Y} \sum_{x \epsilon X} p(x,y) \log \left(\frac{p(x,y)}{p(x) \cdot p(y)} \right) \tag{3.5}$$

   donde X es la frecuencia del *opcode* e Y es la clase del fichero (v. gr., malware o software benigno), $p(x,y)$ es la función de distribución de probabilidad conjunta de X e Y; y $p(x)$ y $p(y)$ son las funciones de distribución de probabilidad marginal de X e Y. En nuestro caso, X es el *opcode* específico e Y representa si el fichero es malware o no.

Una vez hemos calculado la información mutua entre cada *opcode* y la clase de los ejecutables, hemos generado y ordenado un fichero de relevancias de *opcodes*. Hemos guardado el fichero que contiene las frecuencias de aparición para facilitar el proceso de actualización del fichero de relevancias.

La lista de relevancias de *opcodes* ayuda a la detección de malware ponderando la representación final, utilizando los coeficientes obtenidos y, de esta manera, reducir el ruido producido por los *opcodes* irrelevantes. En particular, hemos corroborado que los *opcodes* más habituales como `push`, `mov` o `add`, tienden a tener un bajo coeficiente en el resultado final.

Estos coeficientes pueden considerarse como un reemplazo de la medida IDF [Rob04] utilizada en el VSM [Sal89, MS83] para la recuperación de información. IDF pondera los términos en función de cuánto aparecen en el conjunto total de documentos. Nuestro método realiza una tarea similar utilizando información mutua en vez de sólo contar las ocurrencias de los términos. Nuestro método es, adicionalmente, más fácil de actualizar, utilizando el número total de apariciones de *opcodes* en cada conjunto de datos (software benigno y malicioso).

3.2.4 Aplicación del VSM para representar ejecutables

Para adaptar este enfoque de IR a la detección de malware, representamos los ejecutables como si se tratasen de documentos. Utilizamos el VSM para nuestra representación basada en IR de ejecutables. Como método de ponderación, utilizamos el modelo TF-IDF pero en vez de utilizar la frecuencia inversa de documento (IDF), utilizamos las relevancias de *opcodes* calculadas previamente.

Así, un programa $\mathscr{P}$ se puede definir como una secuencia de instrucciones $\mathscr{I}$ donde $\mathscr{P} = \{\mathscr{I}_1, \mathscr{I}_2, \ldots, \mathscr{I}_{n-1}, \mathscr{I}_n\}$. Una instrucción $\mathscr{I}$ es una bi-tupla compuesta de un *opcode* y de un parámetro o lista de parámetros. Como los *opcodes* resultan ser significativos por sí mismos [Bil07a, Bil07b], descartamos los parámetros y asumimos que un programa $\mathscr{P}$ está compuesto de *opcodes*.

De esta manera, redefinimos un programa $\mathscr{P}$ como una secuencia de opcodes, tal que $\mathscr{P} = \{o_1, o_2, o_3, o_4, \ldots, o_{\ell-1}, o_\ell\}$ donde ℓ es el número de instrucciones $\mathscr{I}$ del programa $\mathscr{P}$. Una secuencia de *opcodes* $\mathscr{S}$ se define como un subconjunto de *opcodes* dentro del fichero ejecutable tal que $\mathscr{S} \subseteq \mathscr{P}$, estando formada por *opcodes* o_i, tal que $\mathscr{S} = \{o_1, o_2, o_3, \ldots, o_{m-1}, o_m\}$, donde m es el tamaño de la secuencia de *opcodes* $\mathscr{S}$.

Considerando el ejemplo mostrado en la Figura 3.6, se pueden generar las siguientes secuencias de tamaño 2: $\mathscr{S}_1$ = (`mov`, `add`), $\mathscr{S}_2$ = (`add`, `push`), $\mathscr{S}_3$ = (`push`, `add`), $\mathscr{S}_4$ = (`add`, `and`), $\mathscr{S}_5$ = (`and`, `push`), $\mathscr{S}_6$ = (`push`, `push`) y $\mathscr{S}_7$ = (`push`, `and`). Como la mayoría de operaciones con intenciones maliciosas requieren más de una operación máquina [Chr07], consideramos que es más apropiado el uso de secuencias de *opcodes* que los *opcodes* por sí mismos.

Seguidamente, calculamos la frecuencia de aparición de cada secuencia dentro de cada fichero utilizando la *frecuencia de secuencia* (SF) (del inglés *Sequence Frequency*):

```
mov   ax,0000h
add   [0BA1Fh],cl
push  cs
add   [si+0CD09h],dh
and   [bx+si+4C01h],di
push  sp
push  7369h
and   [bx+si+72h],dh
```

Figura 3.6: Ejemplo de código máquina desensamblado.

$$sf_{i,j} = \frac{n_{i,j}}{\sum_k n_{k,j}} \tag{3.6}$$

donde $n_{i,j}$ es el número de veces que la secuencia de *opcodes* $\mathscr{S}$ aparece dentro de un ejecutable $\mathscr{P}$, y $\sum_k n_{k,j}$ es el número total de secuencias en el ejecutable $\mathscr{P}$.

Calculamos esta medida para toda secuencia posible de un tamaño fijo n, obteniendo un vector $\vec{v}$ de frecuencias de secuencias de *opcodes*. Ponderamos la frecuencia de aparición de cada secuencia de *opcodes*, utilizando los coeficientes de relevancia descritos anteriormente.

Definimos *frecuencia ponderada de secuencia* (*Weighted Sequence Frequency*, WSF) como el resultado de ponderar cada *opcode* en una secuencia utilizando los coeficientes previamente calculados. Calculamos WSF como el producto de la frecuencia de aparición de cada secuencia de *opcodes* y los coeficientes calculados de los *opcodes* que componen la secuencia:

$$wsf_{i,j} = sf_{i,j} \cdot \prod_{o_z \epsilon S} \frac{coeficiente(o_z)}{100} \tag{3.7}$$

donde $coeficiente(o_z)$ es la relevancia del opcode o_z y $sf_{i,j}$ es la frecuencia de aparición para una secuencia dada.

Aplicando los coeficientes, obtenemos un vector $\vec{v}$ que está compuesto por WSFs, tal que $\vec{v} = ((\mathscr{S}_1, wtf_1), (\mathscr{S}_2, wtf_2), (\mathscr{S}_3, wtf_3), ..., (\mathscr{S}_{n-1}, wtf_{n-1}), (\mathscr{S}_n, wtf_n))$ donde $\mathscr{S}$ es la secuencia de *opcodes* y wtf_i es la frecuencia ponderada de esa secuencia en particular.

Utilizando esta representación, podemos, por un lado, calcular la similitud entre representaciones de ejecutables pudiendo identificar variantes de malware y, por otro lado, entrenar algoritmos de aprendizaje automático para la detección de malware desconocido.

3.3 Evaluación

Para la evaluación del método de representación propuesto, hemos implementado un algoritmo capaz de extraer la representación de los ejecutables dentro de un directorio. Denominaremos a este algoritmo *N-Opcode-Analyser* (NOA).

Tabla 3.7: Descripción del conjunto de datos para la evaluación de tiempos de NOA.

Familia de Malware	Número de ejecutables
Agobot	98
Antilam	6
Bancodor	10
Bifrose	28
Cmjspy	8
Delf	118
Graybrid	20
Gruel	12
Hupigon	26
IRCBot	23
LittleWitch	10
Mosucker	12
PcClient	18
Rbot	23
SdBot	51
SubSeven	8
VB	263
Zokrim	17
Total	*751*

Como plataforma para nuestra evaluación de tiempos, hemos utilizado un procesador Intel Pentium M (2.13 Ghz) con 2 GB DDR de memoria RAM. La Tabla 3.7 muestra los ejecutables maliciosos que se han utilizado para la evaluación. Estos ejecutables se han descargado del conocido sitio web *VxHeavens* [VX10] y se han seleccionado únicamente las variantes que fuesen del formato *Portable Executable*. Son un total de 18 familias diferentes de malware, entre las que encontramos *bots*, caballos de Troya, gusanos, etc. Teniendo en cuenta el número total de ejecutables por familia, se han realizado un total de 751 extracciones de secuencias de *opcodes*.

Hemos evaluado NOA midiendo el tiempo necesario para extraer la representación vectorial de los ejecutables y comparado los resultados con los tiempos requeridos por funciones de resumen criptográfico. Hemos evaluado el tiempo de NOA para diferentes longitudes de secuencias: $n = 1$, $n = 2$, $n = 3$, $n = 4$ y $n = 5$ y los hemos comparado con las funciones de resumen *Message-Digest algorithm 5* (MD5) [Riv92] y SHA1 [FIP01].

- **MD5** es un algoritmo de resumen de criptografía que utiliza un valor de resumen de 128 bits. Se ha utilizado MD5 para un gran número de aplicaciones en seguridad y también se utiliza para comprobar la integridad de los ficheros (como en el caso del malware). No obstante, se ha demostrado que pueden existir colisiones (en otras palabras, varios mensajes con la misma clave) y, por consiguiente, MD5 no es apropiado para aplicaciones como certificaciones SSL o firmas digitales.

- **SHA-1** probablemente sea la más utilizada de las funciones SHA que existen y se ha utilizado en multitud de aplicaciones y protocolos. Sin embargo, también se han identificado algunos fallos en la seguridad de SHA1 que indican que existe una debilidad matemática y que se necesita alguna función de resumen más fuerte [WYY05].

Con nuestro enfoque, pretendemos proveer de una cobertura más extensa frente a una familia de malware e incluso detectar malware desconocido. Utilizamos estos métodos como una línea de base, debido a que uno de nuestros objetivos es que el método tenga un eficiencia considerable para representar ejecutables.

En concreto, queremos responder a la siguiente pregunta:

> «*¿Cuánta sobrecarga computacional introduce nuestro método de representación respecto a la sobrecarga que introducen los modelos más simples que se pueden utilizar para representar ejecutables y detectar malware?*»

Por supuesto, esperamos de antemano que nuestra representación sea más lenta que las funciones de resumen criptográfico. No obstante, las utilizamos como líneas de base porque son las representaciones más rápidas que se pueden utilizar para los ejecutables. Estas funciones, además, son en muchas ocasiones utilizadas por los antivirus comerciales para la detección

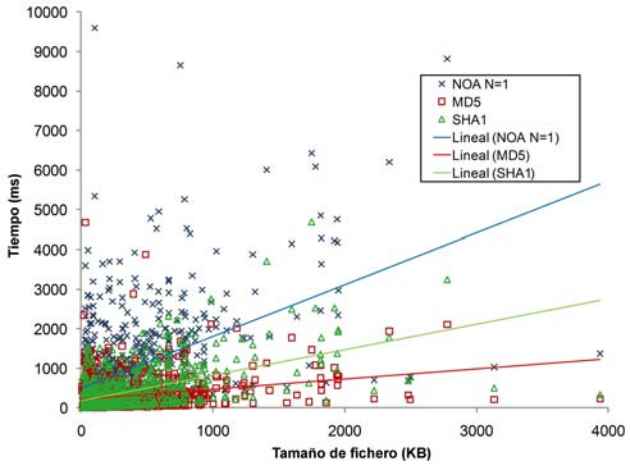

Figura 3.7: Gráfico de dispersión de tiempos para nuestro método de representación con una longitud de secuencia $n = 1$ y las funciones de resumen básicas.

de malware [Szö05]. Obviamente, éstas tan sólo pueden detectar un ejecutable concreto y cualquier tipo de variación en el mismo, que modificase el valor de la función de resumen, haría que se perdiese la detección.

La Figura 3.7 muestra un gráfico de dispersión de tiempos para nuestro método de representación con una longitud de secuencia $n = 1$ y las funcio-

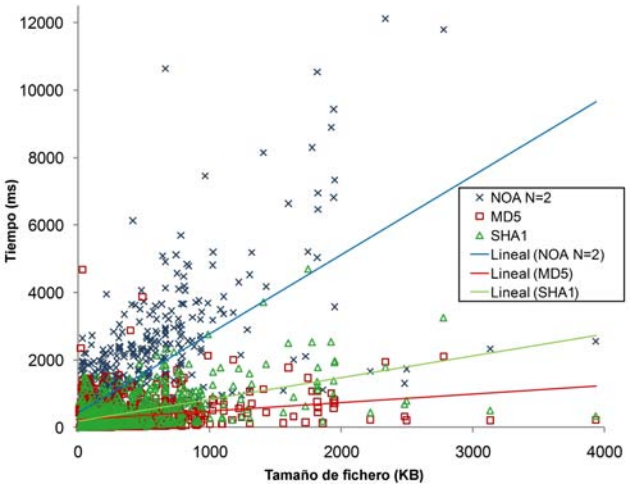

Figura 3.8: Gráfico de dispersión de tiempos para nuestro método de representación con una longitud de secuencia $n = 2$ y las funciones de resumen básicas.

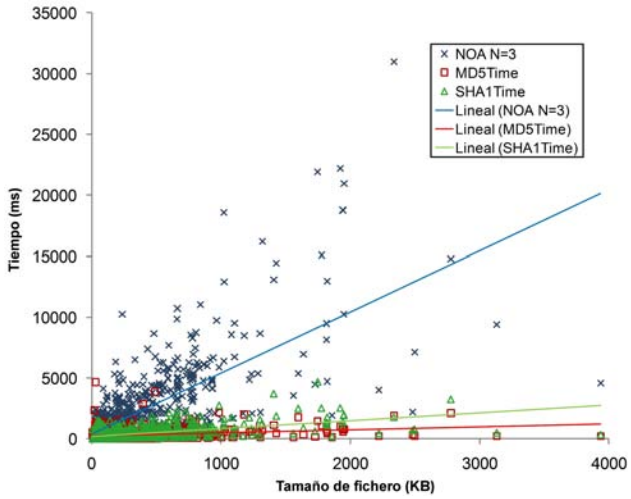

Figura 3.9: Gráfico de dispersión de tiempos para nuestro método de representación con una longitud de secuencia $n = 3$ y las funciones de resumen básicas.

nes de resumen básicas. Los resultados para las secuencias de este tamaño indican que, tal y como se pensaba, nuestro método es más lento que los métodos más sencillos. Sin embargo, la diferencia en comportamiento para secuencias de este tamaño ha sido bastante pequeña: 3 veces más lento que MD5 y 2,5 veces más lento que SHA1 de media.

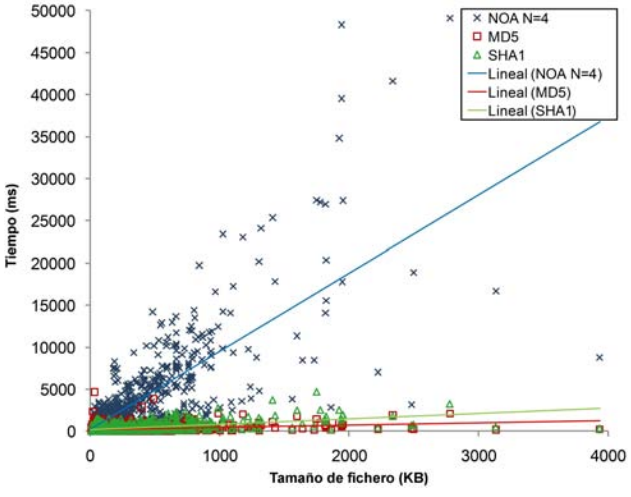

Figura 3.10: Gráfico de dispersión de tiempos para nuestro método de representación con una longitud de secuencia $n = 4$ y las funciones de resumen básicas.

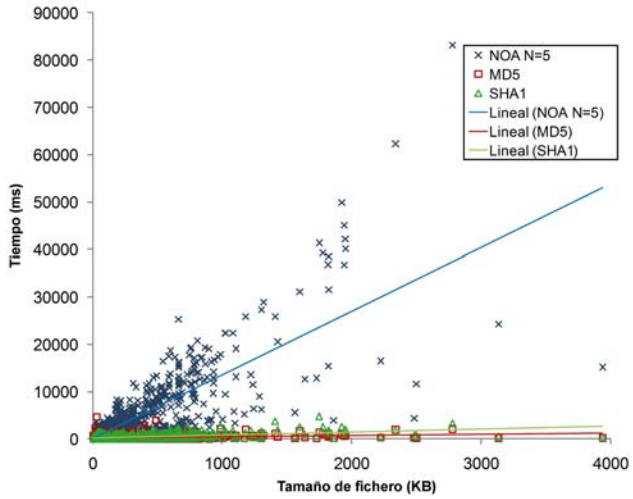

Figura 3.11: Gráfico de dispersión de tiempos para nuestro método de representación con una longitud de secuencia $n = 5$ y las funciones de resumen básicas.

La Figura 3.8 muestra el gráfico de dispersión de tiempos para nuestro método de representación con una longitud de secuencia $n = 2$ y las funciones de resumen básicas. NOA fue más lento con valor de longitud de 2 que con 1, demostrando que la eficiencia depende de la longitud de la secuencia. En la mayoría de los casos, NOA tardó entre 600 y 2200 ms: aproximadamente 4 veces más que MD5 y 3,5 que SHA1.

La Figura 3.9 muestra el gráfico de dispersión de tiempos para nuestro método de representación con una longitud de secuencia $n = 3$ y las funciones de resumen básicas. Con esta configuración la mayoría de las pruebas de NOA tardaron entre 2.000 y 4.000 ms: cerca de 6,8 veces más que MD5 y 5,3 veces que SHA1. Si comparamos estos resultados con los obtenidos para $n = 1$ y $n = 2$, la diferencia es aún más grande sugiriendo que la relación entre el tiempo y el tamaño de la secuencia puede ser polinomial.

La Figura 3.10 muestra el gráfico de dispersión de tiempos para nuestro método de representación con una longitud de secuencia $n = 3$ y las funciones de resumen básicas. Los tiempos de ejecución de NOA aumentan comparando con las configuraciones anteriores: el tiempo necesario incrementa en 3,5 veces los resultados para los tamaños más pequeños. Esta configuración fue aproximadamente 10 veces más lenta que MD5 y 8 veces que SHA1.

Para $n = 5$ (mostrado en la Figura 3.11), NOA es 5 veces más lento utilizando esta configuración que con una longitud de 1. Esta configuración es

15 veces más lenta que MD5 y 11 veces más lenta que SHA1.

A modo de resumen de los resultados, la Figura 3.12 muestra la evolución de los tiempos medios de NOA según cambia *n*. Podemos apreciar como la relación es polinomial, indicando que cuanto más grande sea el tamaño de la secuencia, más tardará NOA en obtener la representación.

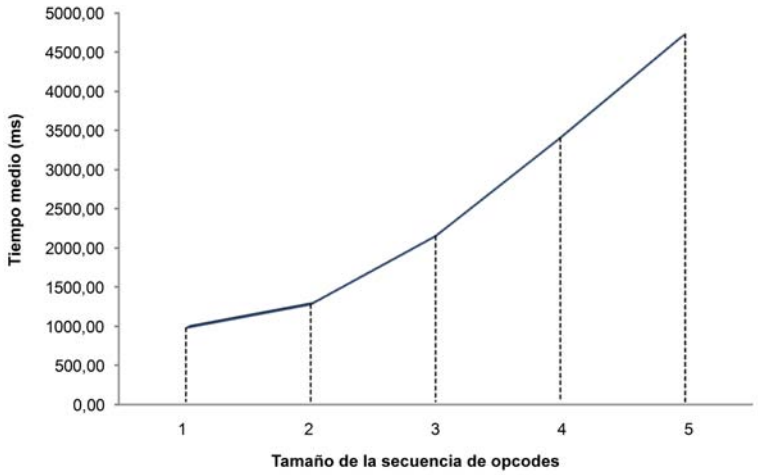

Figura 3.12: Evolución del rendimiento dependiendo del tamaño de la secuencia de *opcodes*.

Como conclusión, los resultados obtenidos demuestran que nuestro método de representación, a pesar de ser, obviamente, más lento que los métodos de resumen, es lo suficientemente rápido para su adopción como método de representación de ejecutables.

3.4 Discusión de los resultados

Nuestros resultados validan parte de nuestra hipótesis, demostrando que existen métodos de representación que son rápidos y que van más allá de meras funciones de resumen. Utilizaremos esta representación para el detector de variantes que presentaremos en la capítulo posterior. Sin embargo, existen algunos temas susceptibles de ser sometidos a debate en lo referente a la viabilidad y confianza de este método.

En primer lugar, nuestro método se basa en una fase de desensamblado estática. Debido a esta naturaleza estática, no puede manejar malware *empaquetado*. El malware empaquetado es el resultado de cifrar el código real de un ejecutable y descifrarlo cuando el ejecutable se carga en memoria. De

hecho, los métodos estáticos sólo podrían manejar código empaquetado utilizando representaciones de empaquetadores conocidos, por lo que, el análisis dinámico parece una mejor solución. Otra solución para este problema sería utilizar, en un paso previo, un desempaquetador genérico y dinámico como *PolyUnpack* [RHD+06], *Renovo* [KPY07], *OmniUnpack* [MCJ07] o *Eureka* [SYS+08]. Estos métodos ejecutan el binario en un entorno controlado y extraen su código real permitiendo un análisis posterior del mismo. Otra solución sería emplear rutinas específicas de desempaquetado, que requieren una rutina por algoritmo de empaquetado [Szö05], para recuperar el código. De igual modo, los antivirus comerciales usan técnicas de *Rayos X* que pueden manejar enfoques de compresión conocidos y cifrados débiles [PF04]. No obstante, estos métodos no pueden afrontar la creciente cantidad de nuevos algoritmos de empaquetado y, por lo tanto, los esquemas dinámicos y genéricos son la mejor opción.

En segundo lugar, aunque nuestro método ha sido capaz de extraer rápidamente la representación de los ejecutables, hay una fuerte dependencia entre el tamaño de la secuencia de *opcodes* y el tiempo requerido por NOA. Así, en la evaluación del ratio de detección de variantes de esta representación, tendremos en cuenta este hecho para buscar un equilibrio entre el tiempo necesario para la representación y la capacidad de detección. Además, hay que considerar que NOA es un prototipo experimental y, por ello, no está realmente optimizado. Si la técnica fuese a ser utilizada comercialmente, se necesitaría una fase de optimización que seguramente mejoraría los tiempos — aunque no la complejidad.

Concluyendo, nuestro método representa ejecutables como vectores formados por frecuencias de secuencias de *opcodes*. El método se ha mostrado moderadamente rápido y no se convierte en un *cuello de botella* en el proceso de generación de representaciones. Consecuentemente, si nuestro método es capaz de detectar más malware que los métodos de firmas actuales — lo cuál es muy probable que ocurra teniendo en cuenta que los métodos de firmas requieren una firma por ejecutable — debería ser considerado como una alternativa o, al menos, un complemento a los métodos actuales.

3.5 **Sumario**

Este capítulo ha introducido un método para la representación de ejecutables. Este método se basa en recuperación de información y utiliza un modelo de espacio vectorial. El ejecutable se representa con secuencias de

opcodes, mediante la frecuencia de aparición y la relevancia de los *opcodes* que componen la secuencia, formando un vector para representarlo en el espacio vectorial.

Hemos comparado el método con funciones de resumen criptográfico para medir su velocidad. Nuestro método era más lento, pero no tan lento como para no considerarlo viable para su implantación en antivirus comerciales. Por lo tanto, si el método demuestra en el capítulo posterior que es capaz de ofrecer bastante cobertura en la detección de una familia de malware, debería ser tenido en consideración para sustituir o complementar al método clásico de firmas.

«Un zorro puede cambiar de piel, pero no de comportamiento.»

Suetonius (70–130)

Capítulo

4

Detección de variantes de malware

Así como el malware evoluciona, también lo hace la creatividad y originalidad de sus autores para ocultar sus creaciones. De este modo, los creadores de malware utilizan técnicas de ofuscación de código para disfrazar el comportamiento real de su código [BMM06, ZR07, CL06b, KWLP05a]. Ejemplos de ofuscaciones son la *inserción de basura*, consistente en añadir instrucciones que no modifican el comportamiento del programa; la *re-ordenación del código*, que consiste en modificar el orden de las instrucciones; o *renombrar variables*, reemplazando un identificador con otro [CJ04]. Estos hechos han llevado a una situación en la que los desarrolladores de malware crean nuevos ejecutables y mecanismos para ocultarlos, mientras que los investigadores y analistas diseñan nuevos métodos para detectarlos [CJ03]. Debido a que el método clásico utiliza una base de datos de firmas [Mor01], no es capaz de detectar las variantes ofuscadas de malware.

Dado este entorno, presentamos un detector de variantes basado en la representación que introdujimos en el capítulo 3. Este detector, utilizando este modelo, hace las siguientes contribuciones al estado del arte:

- Detección de diferentes variantes de una familia de malware mediante algunas representaciones de la familia y no una única representación de cada ejecutable.

- Capacidad de distinguir entre miembros de una familia de malware y software benigno.
- Adaptación del conocimiento de la base de datos de malware creando modelos filogenéticos[1].

El resto del capítulo está organizado como sigue. La sección 4.1 describe los requerimientos necesarios para el método de detección de malware. La sección 4.2 introduce el método propuesto y detalla sus componentes. La sección 4.3 evalúa el método propuesto y presenta los resultados. La sección 4.4 debate las principales limitaciones de este método y propone posibles soluciones. Finalmente, la sección 4.5 resume las contribuciones de este capítulo.

4.1 Requerimientos

Como la cantidad de malware está creciendo y los desarrolladores de malware utilizan técnicas de ofuscación de software, disponer de un método de detección que no pueda ser evadido por éstas es una necesidad real para la detección de malware [Chr07, CKJ+05, CJS+05, JDF08, SMEG09]. No obstante, el método tiene que cumplir ciertos requerimientos para tener éxito:

- **Alta cobertura de variantes.** El problema del método clásico de firmas es que no detecta variantes ofuscadas de malware. Por tanto, nuestro método tiene que ser capaz de hacerlo. Además, el enfoque debería utilizar el mínimo número de representaciones posible o, dicho de otra manera, ser capaz de detectar el mayor número de variantes posible con una única representación.
- **Etiquetado específico.** Buscamos un etiquetado concreto en base a familias en vez de basado en un comportamiento, ya que las familias de malware evolucionan y, de este modo, se puede monitorizar su evolución [KWLP05a].

  Si sólo mantenemos la representación de un ejecutable en la base de datos, habrá un momento en el que no podamos asegurar la protección contra familias de malware [ZR07]. Así, el mantener ciertas representaciones ayuda a identificar el momento en el que se necesita una representación nueva en la base de datos.

[1] Un modelo filogenético determina el desarrollo evolutivo e historia de las especies o una agrupación taxonómica de los organismos.

- **Tasa baja de falsos positivos.** El problema con muchos detectores de malware es que incurren en una alta tasa de falsos positivos [CKJ+05]. La cantidad de falsos positivos hace que un enfoque pueda no ser utilizable en un entorno real donde tendría que enfrentarse a un gran número de ejecutables benignos. Por ende, nuestro método debería ser capaz de producir un número bajo de falsos positivos o ser, al menos, configurable a fin de minimizar el número de falsos positivos.

En resumen, estos requerimientos se pueden traducir en un objetivo claro: «*crear un detector de variantes de malware que especifique la familia de la variante y que mantenga bajo el número de falsos positivos*».

4.2 Descripción del método

En esta sección describimos el método propuesto para la detección de variantes, ofreciendo de una visión general del mismo y de sus componentes.

4.2.1 Visión general

En el capítulo 3, introdujimos una representación basada en IR para modelar los ejecutables. Recordando la definición formal de un modelo de IR, el modelo se definía como una 4-tupla de la forma $[\mathscr{D}, \mathscr{Q}, \mathscr{F}, \mathscr{R}(q_i, d_j)]$ [BR99] donde $\mathscr{D}$ era el conjunto de documentos, $\mathscr{Q}$ el conjunto de búsquedas del usuario, $\mathscr{F}$ era el marco de trabajo para modelar documentos y $\mathscr{R}(q_i, e_j)$ es la función de similitud.

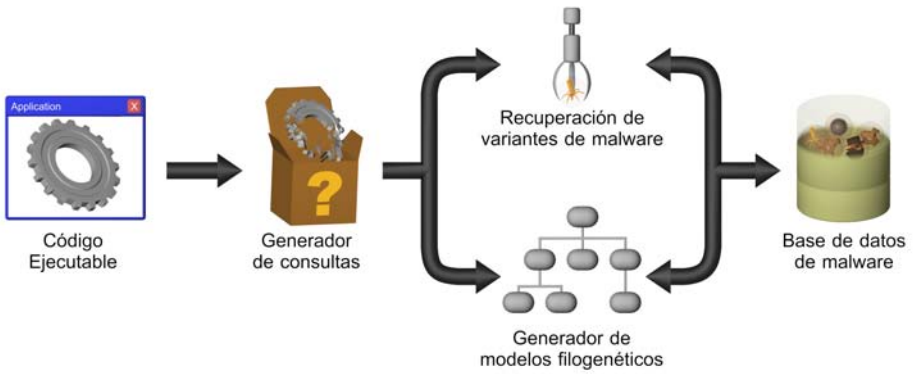

Figura 4.1: Arquitectura general de nuestro método para la detección de variantes.

Hasta ahora, hemos introducido el marco de trabajo para representar ejecutables, es decir, la $\mathscr{F}$ del modelo de IR, restándonos los otros tres componentes. Así, completamos el modelo (ilustrado en la Figura 4.1) con los componentes que nos faltaban de la definición de IR:

- **Base de datos de malware.** La base de datos de malware funciona como el *corpus* de documentos $\mathscr{D}$. Esta base de datos almacenará las representaciones para las diferentes familias de malware.
- **Generador de consultas.** Este componente es el responsable de transformar un ejecutable en una consulta para el sistema de IR. En otras palabras, este componente desensambla el ejecutable y forma la representación que hemos descrito en el capítulo anterior.
- **Recuperación de variantes de malware.** Cuando el ejecutable que está siendo analizado se transforma en una consulta, este componente recuperará las instancias más similares que haya en la base de datos de malware, basándose en la función de similitud.
- **Generador de modelos filogenéticos.** Este componente se encarga de construir modelos filogenéticos para las familias de malware. Esta representación de las familias nos permite organizar la base de datos de malware, ayuda a etiquetar las nuevas variantes y monitoriza la evolución de una familia de malware.

El funcionamiento del sistema comienza con el *generador de consultas* desensamblando el ejecutable y generando su representación vectorial que actuaría como la consulta a la base de datos. Seguidamente, en función de una medida de similitud, se recuperan los ejecutables cuyas representaciones seleccione el sistema de recuperación de variantes de malware. Es importante definir un rango de similitud a partir del cual recuperar los ejecutables. Finalmente, si no existe ninguna representación parecida en la base de datos de malware, el ejecutable se considera benigno. En caso contrario, se le considera de la familia de la que sean las variantes recuperadas.

4.2.2 Base de datos de malware

Este componente almacena las representaciones de las variantes de las diferentes familias de malware, como un *corpus* de ejecutables $\mathscr{E}$ para la recuperación de las variantes más similares, del mismo modo en que una base de datos de documentos $\mathscr{D}$ lo hace en recuperación de la información.

Nuestro enfoque almacena los ejecutables y sus secuencias de *opcodes* en una base de datos para la recuperación de los ejecutables más parecidos a una consulta. La información que se almacena se puede considerar como *firmas*, debido a que con ésta se va a detectar variantes de malware.

En esta subsección, describimos el modelo de datos que vamos a emplear para la base de datos de malware. El modelo de datos describe de una manera abstracta cómo se van a representar los datos en un sistema [Che76]. La Figura 4.2 muestra el esquema físico del modelo de datos propuesto donde PK significa clave primaria mientras que FK significa clave externa.

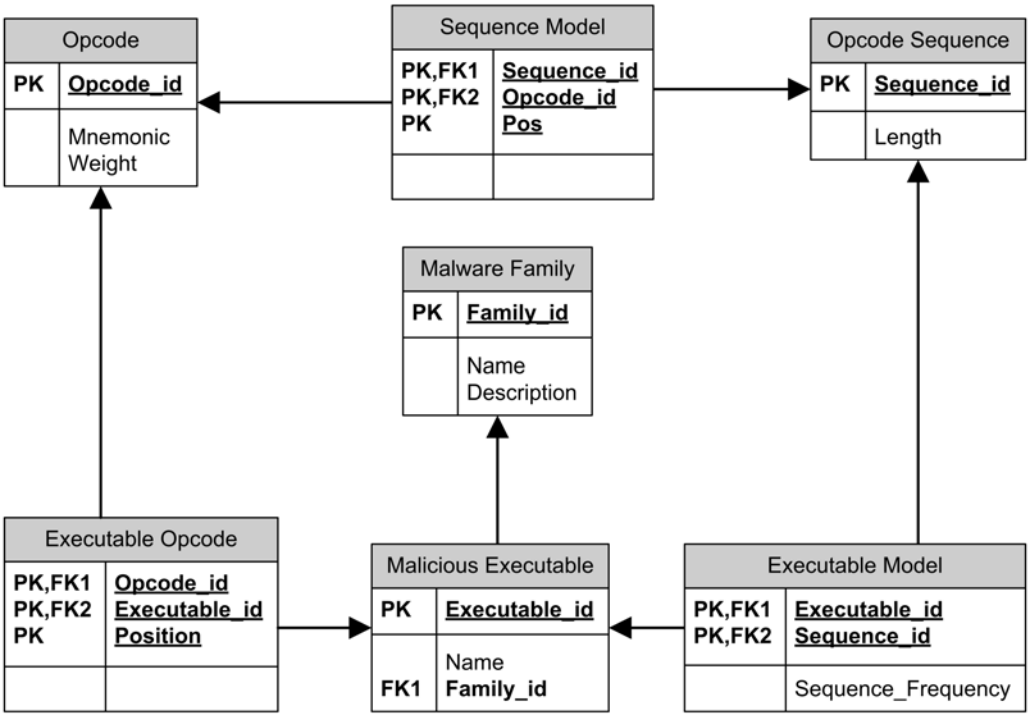

Figura 4.2: Modelo de datos propuesto.

La entidad *Opcode* representa los opcodes únicos del código máquina. *Opcode_id* es la identificación del *opcode*. El atributo *Mnemonic* es la representación en ensamblador de la operación máquina a realizar, mientras que *Weight* es la relevancia calculada para ese *opcode* (véase la sección 3.2.3 del capítulo 3).

La entidad *Sequence model* almacena las relaciones entre los diferentes *opcodes* y las secuencias. *Sequence_id* es la identificación de la secuencia, *Opcode_id* es la identificación del *opcode* y *Pos* es la posición del *opcode* en

la secuencia.

La entidad *Opcode sequence* representa las diferentes secuencias de *opcodes*. *Sequence_id* es la identificación para la secuencia única de *opcodes* y *Length* establece el tamaño de la misma.

Executable model es una entidad que almacena el número de veces que una secuencia aparece en un ejecutable. *Executable_id* es la identificación del ejecutable. *Sequence_id* es la identificación de la secuencia y *Sequence frequency* es el número de veces que la secuencia aparece en el ejecutable. La entidad *Malicious executable* guarda la información de los ejecutables que son malware. *Name* establece el etiquetado de ese ejecutable y *Family_id* es la identificación de la familia de malware a la que el ejecutable pertenece.

Malware family almacena las diferentes familias de malware. *Name* es el nombre de la familia y *Description* un pequeño resumen de la funcionalidad de ésta.

Finalmente, *Executable opcode* representa a los *opcodes* que un ejecutable dado contiene. *Executable_id* es la identificación del ejecutable, *Opcode_id* es la identificación del *opcode* y *Pos* almacena la posición del *opcode* en el ejecutable.

4.2.3 **Generador de consultas**

Este componente genera una consulta en formato de secuencias de *opcodes* a partir del binario ejecutable. La consulta resultante es el elemento clave para la recuperación de variantes similares.

Ya que el código fuente no está disponible para los ejecutables, el primer paso es desensamblarlos. Utilizamos *New Basic Assembler* [New10] un desensamblador/ensamblador *open-source*. Sin embargo, algunos de los ejecutables pueden estar empaquetados y no seríamos capaces de extraer su código ensamblador. Por consiguiente, se necesita un paso previo en el que los ejecutables empaquetados sean desempaquetados (p. ej., con PEiD [PeI10]).

Una vez que el código está disponible, extraemos la secuencia de *opcodes* para representar el ejecutable. Se requiere configurar la longitud de la secuencia para esta fase o podemos utilizar diferentes longitudes de secuencias de *opcodes* y combinarlas para formar una representación única del ejecutable.

Finalmente, en función de las secuencias extraídas, generamos la representación vectorial de frecuencias ponderadas de secuencias de *opcodes*. Consideramos a este vector la consulta para la detección de variantes de malware.

4.2.4 Recuperación de variantes de malware

La recuperación de los ejecutables maliciosos que se parezcan más a la consulta generada se corresponde con la fase de detección del modelo propuesto. Una vez que generamos el vector de frecuencias ponderadas de secuencias de *opcodes*, lo comparamos con los ejecutables maliciosos que se encuentren en la base de datos de malware.

Queremos medir cuánto de similar es la consulta, que representa al ejecutable analizable, a los ejecutables dentro de la base de datos. Representando los ficheros como dos vectores de entrada $\vec{v}$ y $\vec{u}$, podemos medir su similitud mediante la similitud del coseno [TP07].

Esta medida está en el rango entre 0 y 1, donde 0 significa que los dos ejecutables son completamente diferentes y 1 significa que tienen la misma representación. Para recuperar los ejecutables más similares, tenemos que establecer qué significa *similar*, es decir, tenemos que establecer un rango de similitud a partir del cual consideramos que un ejecutable es similar a otro. No obstante, no se puede determinar el rango de antemano y, por tanto, habrá que establecerlo a partir de la experiencia.

4.2.5 Generador de modelos filogenéticos

Se denomina filogenética al estudio de las relaciones de evolución entre diferentes grupos de organismos [Fel85]. En bioinformática, los modelos filogenéticos se representan en función de la información de los nucleídos, las proteínas o las secuencias de genes [RBR+07]. El objetivo principal de estos sistemas es «inferir el proceso que está detrás del patrón» [Inf04].

Debido a que los autores de malware a menudo utilizan generadores y comparten librerías de código, se podría decir que existe una red de intercambio de software robusta [ACKS04a, ACKS04b, KWLP05a, KWLP05b]. Por ende, el malware y sus familias evolucionan generando un gran número de bifurcaciones de su forma original [KWLP05a].

Existen varios estudios [GT01, SCF06, AVMD02] que han examinado el software utilizando sistemas filogenéticos para reconstruir la historia de su evolución [KWLP05a, KWLP05b]. El enfoque que se suele emplear analiza los módulos o sub-sistemas de una aplicación con el fin de evaluar las partes que son iguales o las partes que son diferentes. Siguiendo este modelo, estos enfoques pueden inferir relaciones de derivación de software. Por tanto, generar modelos filogenéticos de malware utilizando técnicas similares a las que se usan en bioinformática, puede ayudar sustancialmente a con-

seguir un mejor análisis del malware. Este tipo de modelos da pistas a los analistas de malware sobre cómo los especímenes de malware se relacionan con versiones más antiguas de sí mismos [KWLP05a].

Karim et ál. [KWLP05a] se dieron cuenta de que, a pesar de no poder reflejar el modelo filogenético real del malware, generar un modelo sigue resultando útil para el análisis del malware. La razón es que pueden existir similitudes además de las que están relacionadas con la derivación de código.

Utilizando la representación vectorial detallada en el capítulo 3, construimos modelos filogenéticos utilizando la similitud del coseno [TP07] y técnicas comunes de *clustering* (agrupamiento, en castellano).

Nuestro objetivo es generar un modelo filogenético por cada familia de malware y, así, conseguir dos contribuciones diferentes al análisis de malware:

- **Encontrar la tendencia evolutiva de una familia de malware**. El modelo filogenético explica exactamente cómo una familia de malware ha evolucionado a lo largo del tiempo.

  Por ello, éste puede ser interesante para responder a la pregunta acerca de cuáles son las tendencias de los desarrolladores de malware.

- **Generar un etiquetado para el malware.** Cada uno de los diferentes antivirus comerciales suele generar una etiqueta diferente [Bon04] para los ejecutables maliciosos.

  Los modelos filogenéticos pueden servir para estandarizar este etiquetado, a través de, simplemente, recalcular el modelo cada vez que un nuevo espécimen de malware se encuentra y añadirlo a la base de datos.

Formalmente, una familia de malware $\mathscr{F}$ es un conjunto de ejecutables e_i donde para cada e_i podemos calcular su representación vectorial v_j. Definimos Γ como el conjunto que contiene todos los v_j de una familia de malware tal que $\Gamma = \{v_1, v_2, ..., v_{n-1}, v_n\}$.

Una vez hemos calculado Γ, se puede obtener una matriz Θ (véase la ecuación 4.1) utilizando las diferentes características $wsf_{i,j}$ de las diferentes secuencias $\mathscr{S}_i$ del vector v_j resultante de procesar el ejecutable e_j:

$$\Theta = \begin{pmatrix} wsf_{\mathscr{S}_1,v_1} & wsf_{\mathscr{S}_2,v_1} & \dots & wsf_{\mathscr{S}_{m-1},v_1} & wsf_{\mathscr{S}_m,v_1} \\ wsf_{\mathscr{S}_1,v_2} & wsf_{\mathscr{S}_2,v_2} & \dots & wsf_{\mathscr{S}_{m-1},v_2} & wsf_{\mathscr{S}_m,v_2} \\ & & \dots & & \\ wsf_{\mathscr{S}_1,v_{n-1}} & wsf_{\mathscr{S}_2,v_{n-1}} & \dots & wsf_{\mathscr{S}_{m-1},v_{n-1}} & wsf_{\mathscr{S}_m,v_{n-1}} \\ wsf_{\mathscr{S}_1,v_n} & wsf_{\mathscr{S}_2,v_n} & \dots & wsf_{\mathscr{S}_{m-1},v_n} & wsf_{\mathscr{S}_m,v_n} \end{pmatrix} \tag{4.1}$$

donde n es el número de ejecutables de la familia de malware, m es el número de secuencias de *opcodes* $\mathscr{S}_i$ y $wsf_{\mathscr{S}_i,v_j}$ es la frecuencia ponderada de la secuencia $\mathscr{S}_i$ dentro del vector v_j. Esta matriz es la entrada para calcular y construir el modelo filogenético mediante algoritmos de *clustering*.

Utilizamos *CLUTO* [Kar03], una herramienta para construir *dendrogramas*[1]. Como criterio de agrupamiento utilizamos el método *Unweighted Pair Group Method with Arithmetic mean* (UPGMA) [SS73] (véase la ecuación 4.2) que examina la estructura de la matriz para construir el dendrograma:

$$\frac{1}{|A| \cdot |B|} \sum_{x \in A} \sum_{y \in B} d(x,y) \tag{4.2}$$

donde A y B son grupos o *clusters*, x e y son especímenes dentro de los *clusters* y $d(x,y)$ es la diferencia entre x e y (v. gr., similitud del coseno). De esta forma, los *clusters* más cercanos se combinan en un *cluster* de un nivel más alto. La distancia entre dos *clusters* cualesquiera A y B es la media de todas las distancias entre cada par de objetos $x \in A$ e $y \in B$: en otras palabras, la distancia media entre los elementos que pertenecen a cada *cluster*.

Resumiendo, el generador de modelos filogenéticos es un componente importante del sistema que, basado en los vectores de secuencias de *opcodes* de los especímenes pertenecientes a una familia de malware, es capaz de proveernos de un esquema de etiquetado para el malware y, asimismo, de un método para analizar la evolución de una familia. Por tanto, este sistema asiste a los analistas de malware a mejorar la detección actual de malware e incluso les ayuda a saber cuáles son las nuevas tendencias en la creación y evolución del malware.

[1] Un dendrograma es un diagrama en forma de árbol que se utiliza frecuentemente para ilustrar el posicionamiento de los grupos o *clusters* producidos por un *clustering* jerárquico. Los dendrogramas se emplean en biología para ilustrar el agrupamiento de los genes.

4.3 Evaluación empírica

Esta sección describe los experimentos realizados para la validación del método de detección de variantes de malware, así como los modelos filogenéticos de las familias de malware.

4.3.1 Metodología general

Hemos evaluado nuestro método contra variantes de malware del mundo real, recogidas de *VxHeavens* [VX10]. Este experimento busca responder a las siguientes preguntas:

> «*¿Cómo de similares son los miembros de una familia de malware entre ellos, comparados con la representación vectorial propuesta?*»
>
> «*¿Cómo de similares son los miembros de una familia de malware respecto al software benigno utilizando la representación vectorial propuesta?*»
>
> «*¿Qué precisión tiene un sistema de detección de variantes basado en el modelo propuesto que minimice la cantidad de software benigno clasificado incorrectamente?*»
>
> «*¿Cuánto difieren los modelos filogenéticos de las etiquetas habituales, para familias de malware basadas en la representación vectorial propuesta?*»

Para responder a la primera pregunta, hemos realizado un experimento en el que utilizamos varias familias de malware comparando cada ejecutable malicioso con el resto de los miembros de su familia. Para la segunda cuestión, hemos recogido varios ejecutables benignos de nuestros equipos y hemos realizado un análisis utilizando *Eset Antivirus* [Ese10] para corroborar si realmente eran benignos. Hemos comparado cada ejecutable de este conjunto de datos con todos los ejecutables maliciosos de todas las familias. Para responder a la tercera pregunta hemos establecido ciertos umbrales de similitud para, así, evaluar las capacidades reales de detección del método propuesto. Finalmente, hemos extraído el modelo filogenético de varias familias de malware utilizando *CLUTO* [Kar03], creando una matriz de cada familia como entrada para *CLUTO*. Hemos comparado los modelos con la nomenclatura de *VxHeavens* [VX10].

4.3.2 Similitud de variantes

Hemos empleado 17 familias diferentes de malware (véase la Tabla 4.1) y 1.000 ejecutables benignos de nuestros propios equipos. Para evaluar el método propuesto, hemos comparado cada miembro de cada familia respecto al resto de miembros y al conjunto de software benigno.

Tabla 4.1: Descripción del conjunto de datos de familias de malware para la comparación de similitud.

Familia de malware	Número de variantes
Agobot	98
Antilam	6
Bancodor	10
Bifrose	28
Cmjspy	8
Delf	118
Graybrid	20
Gruel	12
Hupigon	26
IRCBot	23
LittleWitch	10
Mosucker	12
PcClient	18
Rbot	23
SdBot	51
SubSeven	8
Zokrim	17

Las longitudes de las secuencias de *opcodes* que hemos utilizado van desde 1 hasta 5. Además, hemos realizado comparaciones con representaciones combinadas de una longitud n. Esto significa que cada una de las secuencias de longitud 1 hasta n se han unido en un vector combinado. Por ejemplo, una representación combinada de $n = 2$ significa que el vector resultante es la combinación de frecuencias de secuencias de *opcodes* de tamaño 1 y 2. En total hemos extraído 9 representaciones vectoriales diferentes tanto para el malware como para el software benigno.

La Figura 4.3 Histograma de similitudes para un tamaño de secuencia de 1 para variantes de malware donde más del 45 % de las comparaciones entre las variantes de malware se situaron en el intervalo de similitud 95-

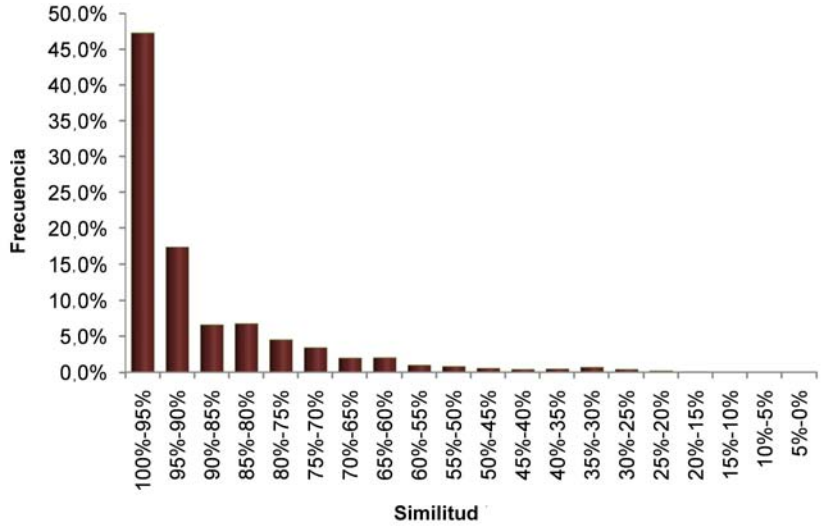

Figura 4.3: Histograma de similitudes para un tamaño de secuencia de 1 para variantes de malware.

100 %.

La Figura 4.4 muestra las similitudes para un tamaño de secuencia de 1 cuando se compara con software benigno. En este último caso, sólo el 3,4 % de los ejecutables legítimos produjeron una similitud mayor del 95 %. To-

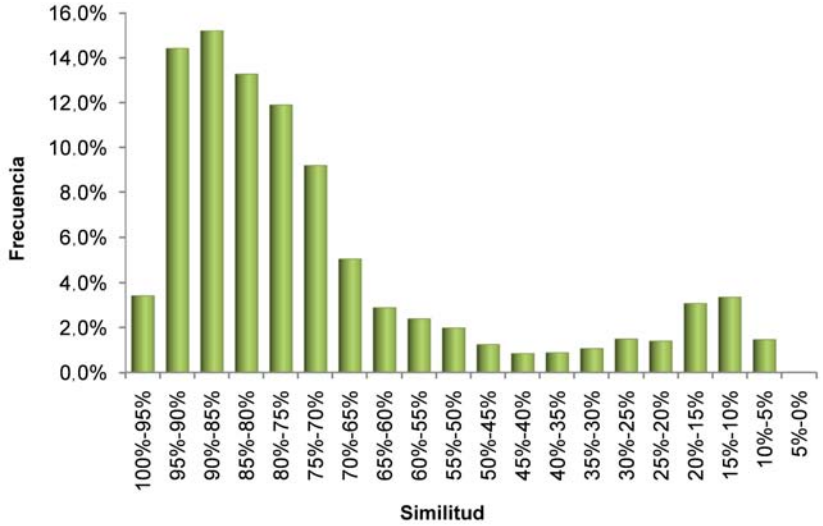

Figura 4.4: Histograma de similitudes para un tamaño de secuencia de 1 cuando se compara con software benigno.

Tabla 4.2: Las 5 familias de malware con los ratios de similitud más altos para una longitud de secuencias de 1.

Familia de malware	Especímenes en el rango 95-100 %
Gruel	100,0 %
Antilam	94,4 %
Bancodor	82,0 %
Zokrim	71,6 %
Hupigon	71,0 %

das las familias se comportaron de una manera similar, con ratios de similitud desde el 0,5 % (de la familia *Zokrim*) al 12,5 % (de la familia PcClient) de las comparaciones produciendo similitudes mayores que el 95 %.

La Tabla 4.2 muestra las 5 familias de malware con los ratios de similitud más altos para una longitud de secuencias de 1. Los ratios de similitud más altos se obtuvieron cuando los ejecutables maliciosos comparados pertenecían a familias con un pequeño número de miembros.Los resultados para una longitud de secuencia de 1 indican que, con esa longitud, las similitudes entre miembros de la misma familia de malware son más altas que el 95 % de similitud en un 40 % de los casos. Sin embargo, el número de ejecutables en ese rango es demasiado alto, haciendo más difícil establecer un rango apropiado para obtener una tasa de cero falsos positivos.

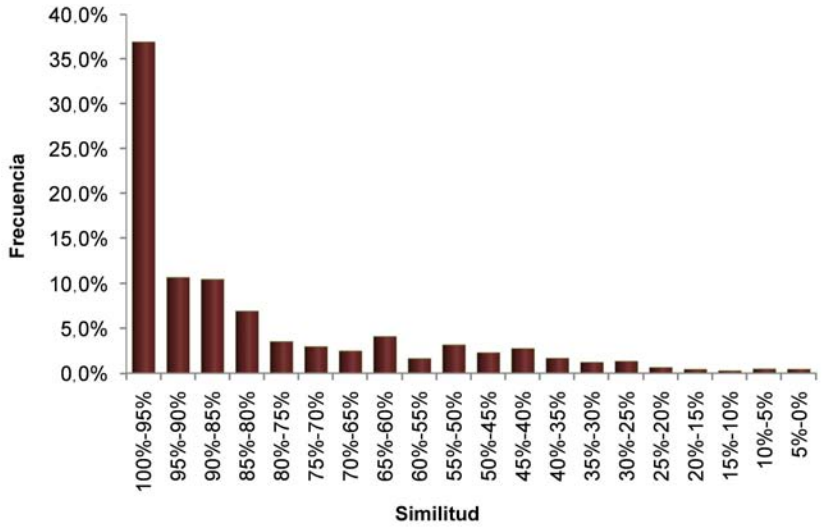

Figura 4.5: Histograma de similitudes para un tamaño de secuencia de 2 para variantes de malware.

La Figura 4.5 muestra las similitudes para un tamaño de secuencia de 2 para variantes de malware. En esta configuración, los ratios de similitud fueron menores que los obtenidos con un tamaño de secuencia de 1: específicamente, el 37 % de las variantes de malware eran similares en más de un 95 % al resto de su familia. Estos resultados exhiben una pérdida del 10 % ó, en otras palabras, esta representación, cubre un 10 % menos de las variantes de malware.

Cuando se compara con software benigno (véase la Figura 4.6), el 1,2 % de los ejecutable benignos produjeron similitudes más altas del 95 %, ratio bastante más pequeño que el obtenido con un tamaño de 1. Esta secuencia produjo similitudes más bajas para el software benigno pero también para las variantes de malware.

La Tabla 4.3 muestra las 5 familias de malware con los ratios de similitud más altos para una longitud de secuencias de 2. Como ocurrió con una longitud de 1, obtuvimos las tasas más altas de similitud en las familias pequeñas. La familia *Antilam* ha reducido su tasa de miembros en el rango de similitud más alto de un 94,4 % a un 50 %. En contraste, la familia *Zokrim* ha incrementado de un 71,6 % a un 79 %. La familia *Gruel* con un 100 % de las comparaciones ofrece tasas de similitud mayores del 95 %.

Los resultados para una longitud de secuencia de 2 nos muestran tasas de similitud menores, especialmente notables en el caso del software legítitmo. El cambio es bastante menos pronunciado en las variantes de malware.

La Figura 4.7 muestra las similitudes para una representación combinada de tamaños 1 y 2 para variantes de malware. Utilizando esta configuración, el 44,9 % de las comparaciones produjeron similitudes mayores del 95 %, resultados más altos que los obtenidos con una longitud de 2 (un 7,9 % más) pero más bajos que los obtenidos con un tamaño de secuencia de 1 (sólo un 2,5 % menos). Cuando se comparan con software benigno (véase la Figura 4.8), el 2,7 % de los ejecutables benignos obtuvieron unas tasas de similitud mayores que el 95 %. Estos resultados son más bajos que los obtenidos con un tamaño de secuencia de 1 pero mayores que para un tamaño de secuencia de 2. En cuanto a las 5 familias de malware con los ratios de similitud más altos para longitudes de secuencia de 1 y 2 (mostradas en la Tabla 4.4), las tasas de similitud también fueron más altas para las familias con un número pequeño de variantes. En general, con la excepción de la familia *Zokrim*, se consiguieron resultados similares a los obtenidos con un tamaño de secuencia de 1.

Los resultados para la representación combinada de secuencias de ta-

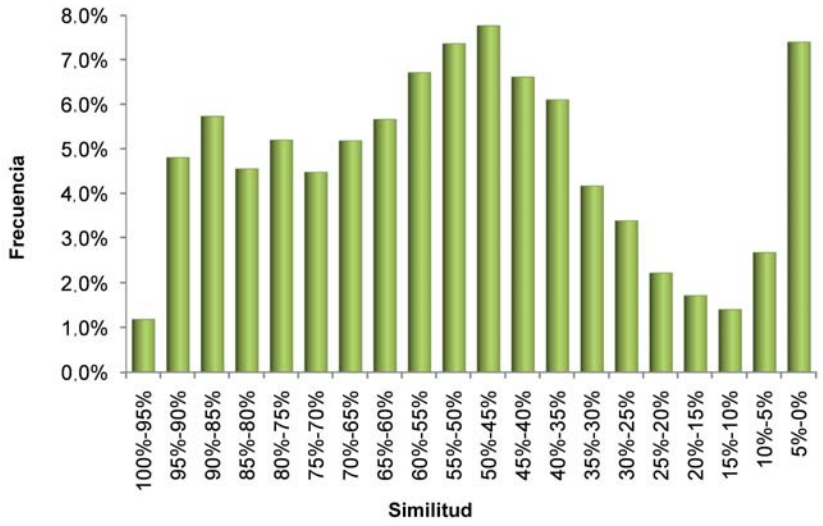

Figura 4.6: Histograma de similitudes para un tamaño de secuencia de 2 cuando se compara con software benigno.

maño 1 y 2 consiguió unos resultados interesantes: la similitud media entre variantes era tan alta como para $n = 1$, mientras que la similitud para ejecutables benignos fue similar que con $n = 2$.

La Figura 4.9 muestra las similitudes para un tamaño de secuencia de 3 cuando se comparan variantes de malware. En este caso, el 33,4% de las variantes consiguieron un ratio de similitud por encima del 95%. Esta porción es menor que con las configuraciones anteriores, sugiriendo que cuanto más grande sea el tamaño de la secuencia de *opcodes*, menor será el ratio de similitud. Cuando se compara software benigno utilizando esta longitud de secuencia, el 1,2% de las comparaciones con software legítimos estuvieron en el rango más alto de similitud. Estos resultados son los

Tabla 4.3: Las 5 familias de malware con los ratios de similitud más altos para una longitud de secuencias de 2.

Familia de malware	Especímenes en el rango 95-100%
Gruel	100,0%
Bancodor	82,0%
Zokrim	79,2%
Hupigon	55,3%
Antilam	50,0%

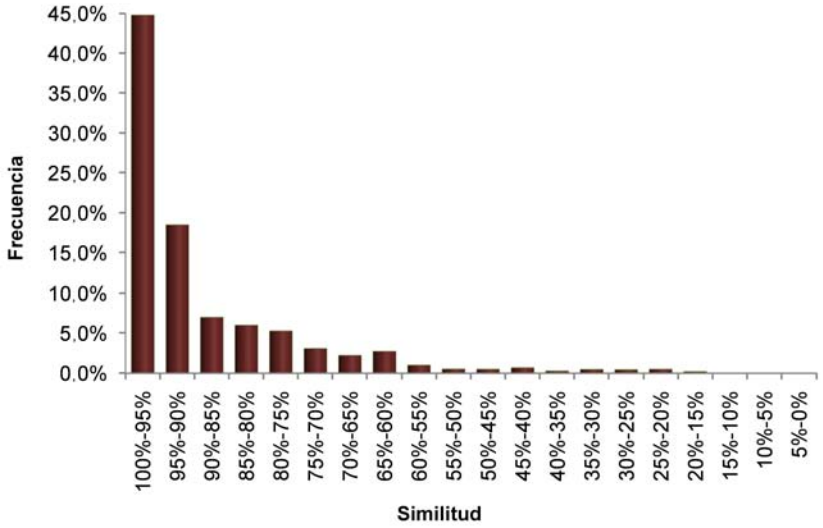

Figura 4.7: Histograma de similitudes para una representación combinada de tamaños 1 y 2 para variantes de malware.

mismos que los obtenidos para $n = 2$, aunque los resultados para variantes fueron más bajos. En cuanto a las familias de malware que tuvieron los resultados más altos (véase la Tabla 4.5) no son las mismas que en el resto de configuraciones. Específicamente, la familia *Antilam* ya no aparece en

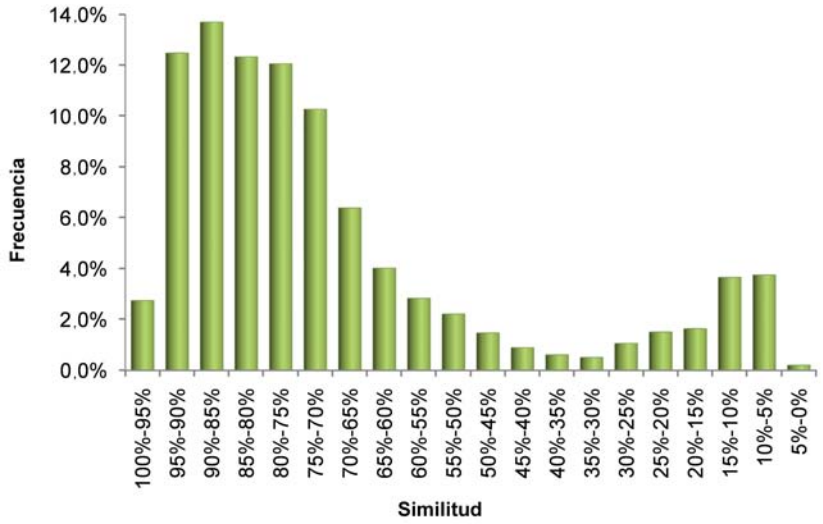

Figura 4.8: Histograma de similitudes para una representación combinada de tamaños 1 y 2 cuando se compara con software benigno.

Tabla 4.4: Las 5 familias de malware con los ratios de similitud más altos para longitudes de secuencia de 1 y 2.

Familia de malware	Especímenes en el rango 95-100 %
Gruel	100,0 %
Antilam	94,0 %
Bancodor	82,0 %
Hupigon	69,8 %
Zokrim	68,9 %

la lista. Sin embargo, la familia *Zokrim* obtiene mejores resultados con esta longitud que con las previas. La familia *LittleWitch* aparece por primera vez

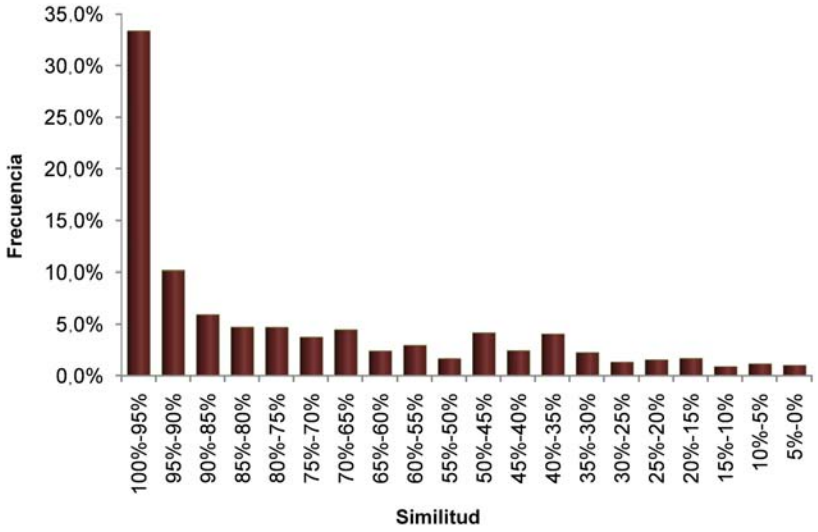

Figura 4.9: Histograma de similitudes para un tamaño de secuencia de 3 para variantes de malware.

Tabla 4.5: Las 5 familias de malware con los ratios de similitud más altos para un tamaño de secuencia de 3.

Familia de malware	Especímenes en el rango 95-100 %
Gruel	100,0 %
Bancodor	74,0 %
Zokrim	88,2 %
Littlewitch	58,0 %
Hupigon	47,3 %

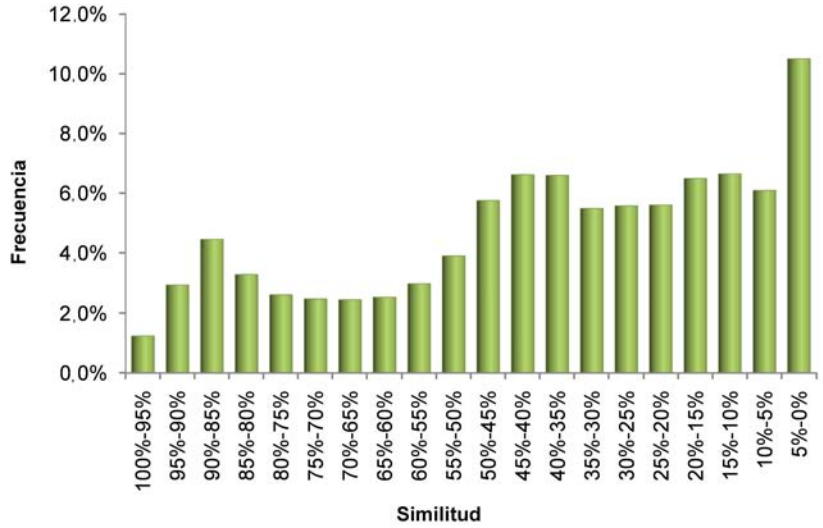

Figura 4.10: Histograma de similitudes para un tamaño de secuencia de 3 cuando se compara con software benigno.

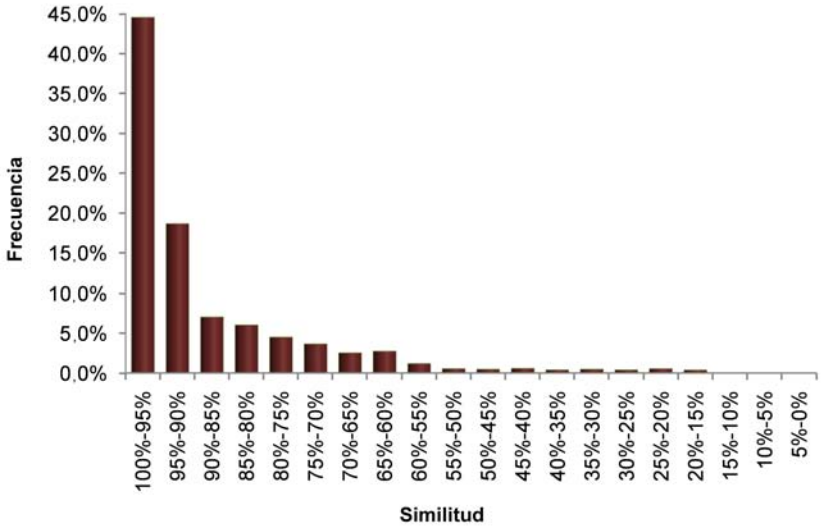

Figura 4.11: Histograma de similitudes para una representación combinada de tamaños 1, 2 y 3 para variantes de malware.

con un 58 % de sus miembros en el rango más alto de similitud. La familia *Gruel* fue otra vez la mejor con todas sus variantes en el rango más alto.

En general, la longitud de secuencia de 3 produjo tasas de similitud más bajas para las variantes de malware. Los resultados para el software benigno son los mismos que con $n = 2$.

Tabla 4.6: Las 5 familias de malware con los ratios de similitud más altos para longitudes de secuencia de 1, 2 y 3.

Familia de malware	Especímenes en el rango 95-100 %
Gruel	100,0 %
Antilam	94,4 %
Bancodor	82,0 %
Zokrim	68,9 %
Hupigon	68,6 %

Para la representación combinada de tamaños 1, 2 y 3 (véase la Figura 4.11), el 44,6 % de las variantes tiene similitud por encima del 95 %. Estos resultados son más altos que para $n = 3$, pero algo más bajos que para $n = 1$ y $n = 2$. Las familias que obtuvieron una tasa de similitud mayor (mostradas en la Tabla 4.6) fueron las mismas que para $n = 1$, $n = 2$ y para la combinación de ambas.

En cuanto al software benigno (véase la Figura 4.12), el 2,7 % de las comparaciones producen similitudes mayores del 95 %, como en la configuración combinada de 1 y 2, aunque en este caso el resto de ejecutables en los otros rangos altos también es menor.

Los resultados para la configuración de las longitudes de 1, 2 y 3 com-

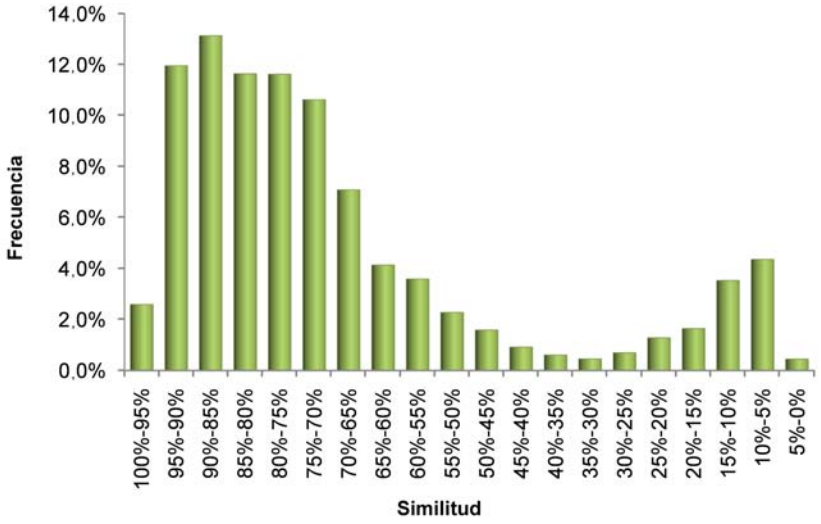

Figura 4.12: Histograma de similitudes para una representación combinada de tamaños 1, 2 y 3 cuando se compara con software benigno.

binadas, muestran que el malware obtiene unas tasas de similitud ligeramente menores que la combinación anterior: 1 y 2. El software benigno, también obtiene unas tasas de similitud menores para esta configuración.

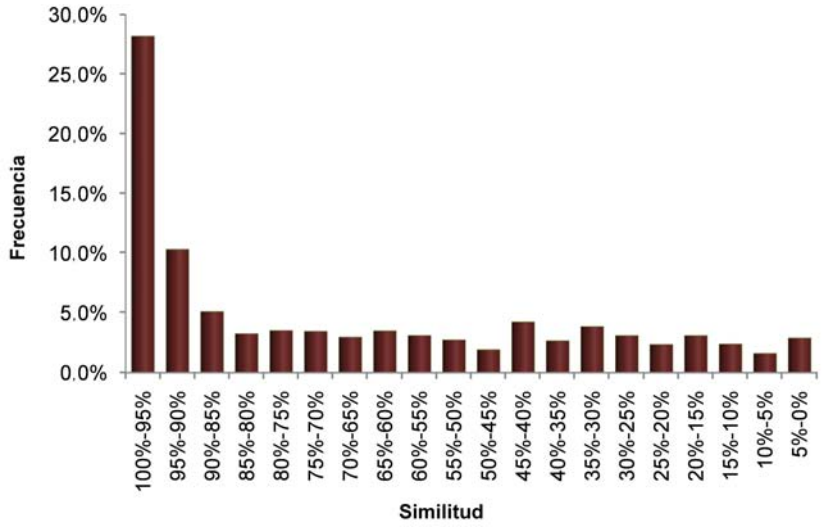

Figura 4.13: Histograma de similitudes para un tamaño de secuencia de 4 para variantes de malware.

La Figura 4.13 muestra el histograma de similitudes para un tamaño de

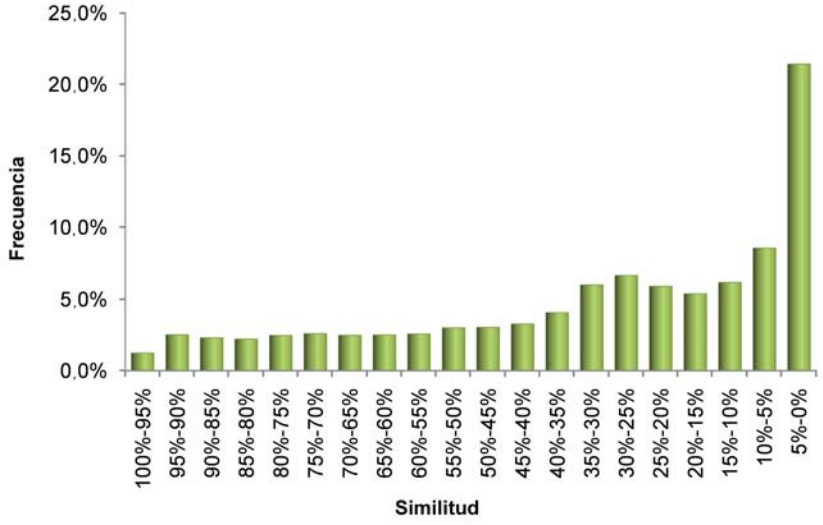

Figura 4.14: Histograma de similitudes para un tamaño de secuencia de 4 cuando se compara con software benigno.

Tabla 4.7: Las 5 familias de malware con los ratios de similitud más altos para un tamaño de secuencia de 4.

Familia de malware	Especímenes en el rango 95-100 %
Gruel	100,0 %
Zokrim	97,9 %
Littlewitch	54,0 %
Cmjspy	43,8 %
Hupigon	34,0 %

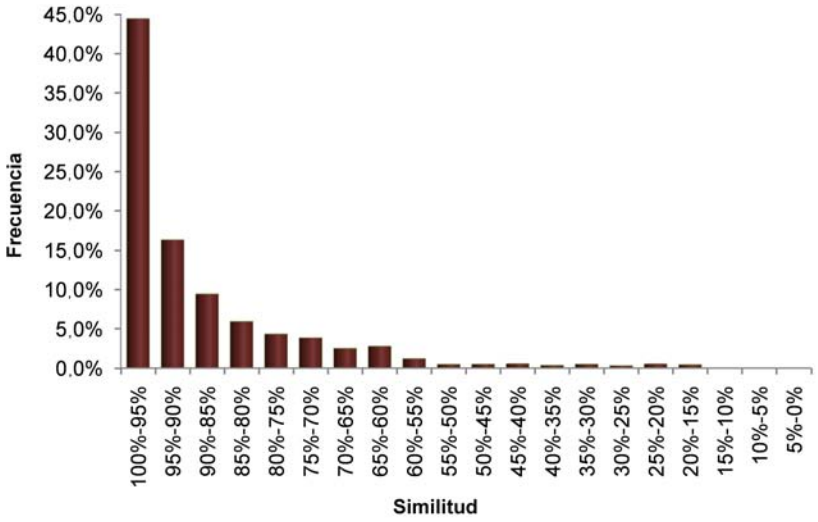

Figura 4.15: Histograma de similitudes para una representación combinada de tamaños 1, 2, 3 y 4 para variantes de malware.

Tabla 4.8: Las 5 familias de malware con los ratios de similitud más altos para longitudes de secuencia de 1, 2, 3 y 4. Las familias con similitudes más altas.

Familia de malware	Especímenes en el rango 95-100 %
Gruel	100,0 %
Antilam	94,4 %
Bancodor	82,0 %
Zokrim	68,9 %
Hupigon	67,5 %

secuencia de 4 para variantes de malware. Sólo el 28,2 % de las variantes de malware estaban en el rango más alto de similitud. Estos resultados son más bajos que los obtenidos anteriormente. Estos resultados parecen con-

Tabla 4.9: Las 5 familias de malware con los ratios de similitud más altos para un tamaño de secuencia de 5.

Familia de malware	Especímenes en el rango 95-100 %
Gruel	100,0 %
Zokrim	100,0 %
Littlewitch	54,0 %
Cmjspy	43,8 %
Bancodor	34,0 %

firmar que cuanto más grande sea la secuencia, más baja será la similitud entre las variantes de malware.

En cuanto al software legítimo, la Figura 4.14 muestra que el 1,2 % de los ejecutables benignos obtuvieron tasas de similitud en el intervalo 95-100 %. Como en el caso de $n = 3$, los resultados son iguales a $n = 2$. En este caso, sin embargo, el resto de rango altos está menos poblado, dispersando las similitudes a rangos más bajos.

En el caso de una longitud de secuencia de 4, la Tabla 4.7 muestra las 5 familias de malware con los ratios de similitud más altos para un tamaño de secuencia de 4. *Bancodor* y *Antilam* no aparecen la lista, decreciendo su población en el rango más alto del 70 % al 20 %. Esto significa que las

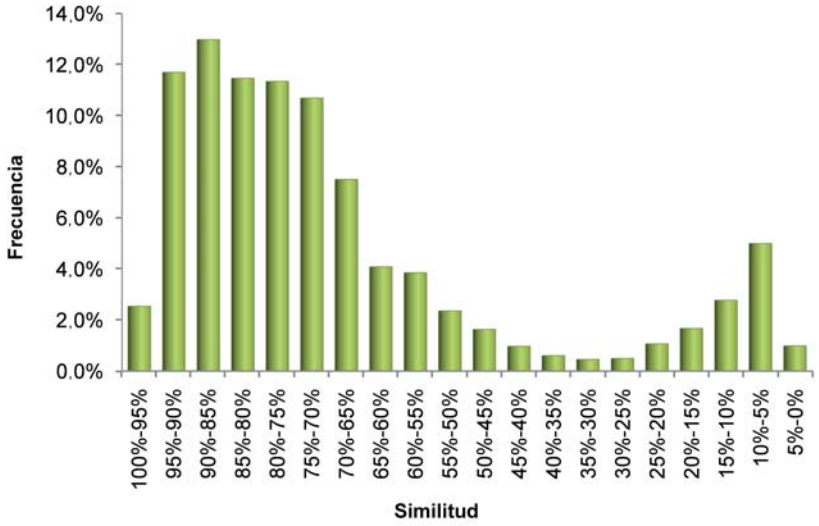

Figura 4.16: Histograma de similitudes para una representación combinada de tamaños 1, 2, 3 y 4 cuando se compara con software benigno.

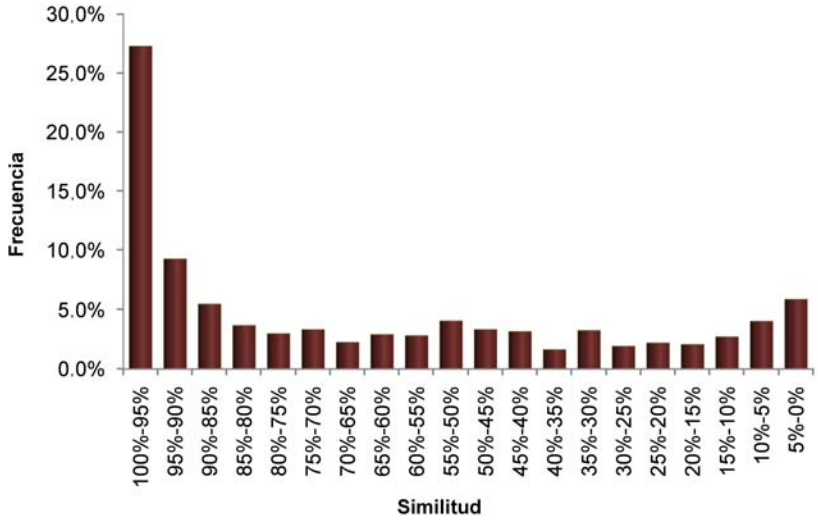

Figura 4.17: Histograma de similitudes para un tamaño de secuencia de 5 para variantes de malware.

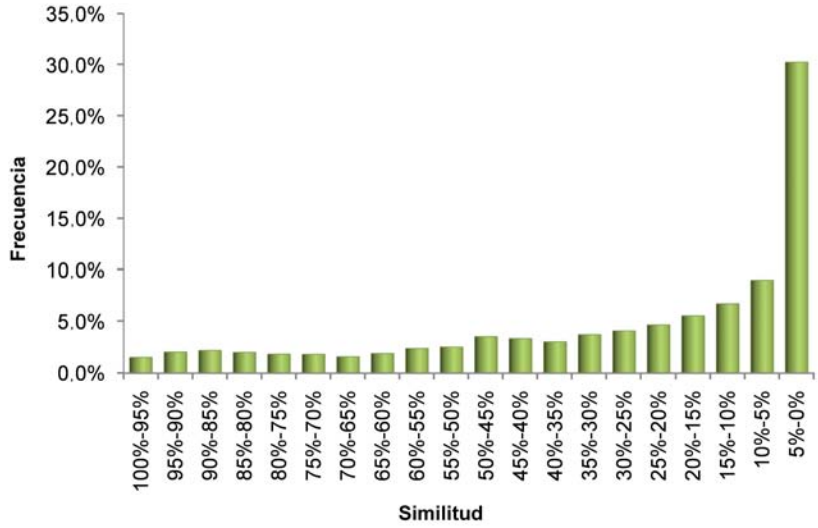

Figura 4.18: Histograma de similitudes para un tamaño de secuencia de 5 cuando se compara con software benigno.

variantes de estas familias estaban probablemente ofuscadas con reordenamiento de código y, por ende, no se pueden dar similitudes con secuencias grandes. Por contra, la población de *Zokrim* incrementa a casi un 100 %.

Los resultados para un tamaño de secuencia de 4, obtienen similitudes menores que los tamaños más pequeños tanto para software malicioso

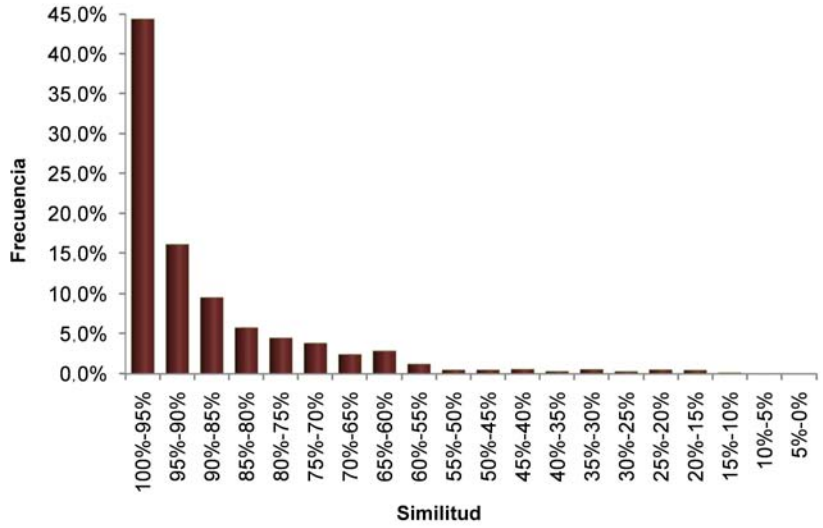

Figura 4.19: Histograma de similitudes para una representación combinada de tamaños 1, 2, 3, 4 y 5 para variantes de malware.

como para software benigno. Muchas de las familias que antes eran fácilmente detectabas dejan de serlo, indicando que las secuencias largas son más fáciles de evadir para las técnicas de ofuscación.

La Figura 4.15 muestra el histograma de similitudes para una represen-

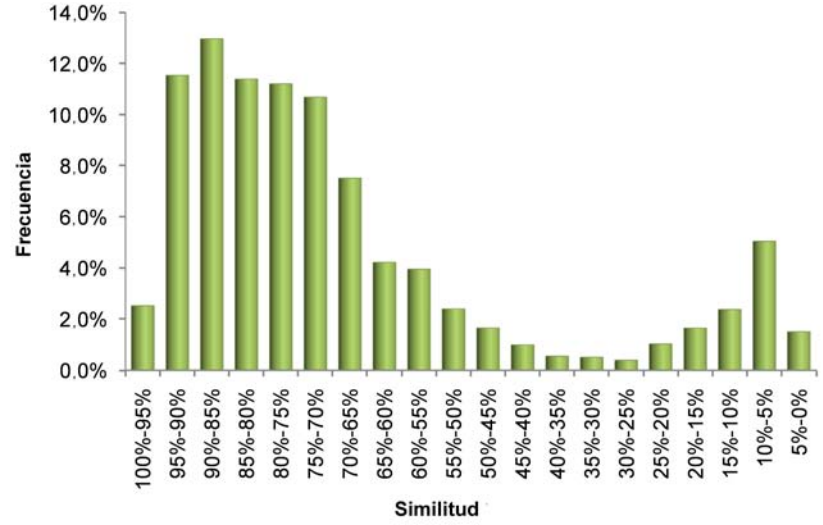

Figura 4.20: Histograma de similitudes para una representación combinada de tamaños 1, 2, 3, 4 y 5 cuando se compara con software benigno.

Tabla 4.10: Las 5 familias de malware con los ratios de similitud más altos para longitudes de secuencia de 1, 2, 3, 4 y 5.

Familia de malware	Especímenes en el rango 95-100 %
Gruel	100,0 %
Antilam	94,4 %
Bancodor	82,0 %
Zokrim	68,9 %
Hupigon	67,5 %

tación combinada de tamaños 1, 2, 3 y 4 para la comparación de variantes de malware. La tendencia en las similitudes utilizando representaciones combinadas se mantiene. Los ratios de similitud para las variantes de malware son prácticamente tan altos como para $n = 1$ y $n = 2$. Específicamente, el 44,5 % de las pruebas de variantes de malware produjeron una similitud mayor del 95 %. En cuanto al software benigno (véase la Figura 4.16), El 2,5 % de las comparaciones resultaron más altas del 95 %. A pesar de que los resultados en el rango más alto son similares a los obtenidos previamente, se siguen dispersando las similitudes a rangos más bajos de similitud. Las familias más similares para esta configuración (véase la Tabla 4.8) son las mismas que en el resto de configuraciones combinadas. La única familia que ha tenido un decremento en la tasa de similitud ha sido *Hupigon* cuyo ratio ha bajado un 1 %.

Las comparaciones entre el malware con una representación combinada con longitudes de secuencia 1, 2, 3 y 4 obtienen unos resultados similares en los rangos más altos de similitud. Sin embargo, las similitudes de rango alto para el código benigno se dispersan a rangos más bajos.

La Figura 4.17 muestra las similitudes entre variantes para un tamaño de secuencia de 5. Sólo el 27,3 % de las variantes de malware se situaron en el intervalo de similitud más alto. Este resultado es el más bajo de las configuraciones probadas, confirmando definitivamente que una longitud de secuencia alta no es apropiada para la detección de variantes de malware. Respecto al software benigno (véase la Figura 4.18), El 1,2 % de los ejecutables benignos estaban en el rango de similitud 95-100 %. Como en el caso de $n = 4$, la fracción de software legítimo es prácticamente la misma que previamente. No obstante, el resto de rangos se mueven a intervalos más bajos. En esta configuración, las familias con mayor ratio de similitud (mostradas en la Tabla 4.9) son las mismas que para $n = 4$, con la excepción de *Hupigon*. Sorprendentemente, el número de variantes de la familia *Zo-*

krim en el intervalo 95-100 % incrementa, significando que las secuencias largas funcionan bien con esta familia. En otras palabras, las ofuscaciones de la familia *Zokrim* son débiles. Sin embargo, surge una interesante cuestión: «*¿Hay que fijar el umbral de similitud a partir del cual se clasifica a un ejecutable como malware, por cada familia o es ese valor común para todas las familia de malware?*».

La configuración $n = 5$ resulta en similitudes más bajas cuando se comparan tanto las variantes de malware como los ejecutables legítimos.

En lo referente a la representación combinada de tamaños de secuencia de 1, 2, 3, 4 y 5, la Figura 4.19 muestra los resultados de las comparaciones entre las variantes de malware. El 44,4 % de las comparaciones de variantes estaban en el rango más alto de similitud. Por otro lado, la Figura 4.20 muestra el resultado de las comparaciones con el software legítimo. Las comparaciones se desplazan a rangos más bajos de similitud que para la combinación de secuencias de 1, 2, 3 y 4. No obstante, la diferencia entre esta configuración y la anterior no es muy significativa y, teniendo en cuenta los tiempos de $n = 5$, la configuración anterior parece más apropiada. La Tabla 4.10 muestra las familias que obtuvieron las similitudes más altas que fueron las mismas que en las otras representaciones combinadas, siendo *Gruel*, *Antilam*, *Bancodor* y *Hupigon* las familias con mayor similitud entre sus miembros.

Queremos remarcar que el tamaño de la familia influye a la hora de determinar la cobertura (en otras palabras, cuántas variantes se detectan con una única representación) de la familia. Obviamente, cuanto más grande sea la familia, también son más grandes las diferencias entre sus variantes [ZR07, Zha08]. No obstante, las diferencias son más pequeñas en las familias con un número pequeño de variantes y, por tanto, podemos concluir que utilizando nuestro método, podríamos detectar nuevas variantes de malware solamente si las variantes más similares han sido previamente registradas en la base de datos de malware. Por tanto, es obligatorio que el sistema registre y almacene toda variante que se desvíe, en términos de similitud, de las previamente documentadas.

Además, en relación a la similitud del software benigno con el malware, los resultados muestran que las tasas de similitud son más bajas que las obtenidas con las variantes de malware. Sin embargo, estos resultados dependen fuertemente de las longitudes de las secuencias de la representación: cuanto más altas sean las secuencias, más baja será la similitud. Por consiguiente, se vuelve más fácil establecer el límite para considerar a un ejecutable malware si hay secuencias de longitud superior a 3.

4.3.3 **Detección de variantes**

En esta sección, nos centramos en describir un sistema de detección de variantes (llamado *N-Opcode Analyzer*: NOA) utilizando los resultados previos. Este experimento muestra que NOA puede detectar variantes de malware dentro de un entorno común de computación.

Para realizar el experimento, hemos seleccionado un conjunto de malware compuesto por variantes de 5 familias de malware conocidas: *Antilam*, *Bancodor*, *Gruel*, *Hupigon* y *PcClient* (para más detalles respecto al conjunto de datos, véase el Apéndice C). Estas muestras se han descargado de VxHeavens [VX10]. Como conjunto de software benigno hemos utilizado un conjunto de 1.000 ejecutables recogidos de una instalación reciente de *Microsoft Windows XP*.

Después, hemos seleccionado un subconjunto de las variantes de malware, que consiste en el 80 % de cada familia de malware para poblar la base de datos de variantes de malware. Hemos utilizado el 20 % restante para analizarlo y, de este modo, evaluar NOA. Además, hemos analizado los 1.000 ejecutables benignos para comprobar el número de falsos positivos (el número de ejecutables benignos incorrectamente clasificados como malware) en los que NOA incurre.

En nuestra validación, hemos empleado representaciones de un tamaño de secuencia desde 1 a 3, combinando los enfoques *recursivos* (R) y *no recursivo* (NR).

Tabla 4.11: Los parámetros utilizados para el primer experimento de detección.

Parámetros	Valores posibles
Tamaño de secuencia	1, 2, 3
Recursividad	Recursivo (R), No Recursivo (NR)
Umbral	0,99, 0,98, 0,97, 0,96, 0,95

Hemos implementado una rutina simple que llama al proceso de análisis de un ejecutable utilizando el enfoque propuesto. De este modo, repetimos un proceso de análisis para cada parámetro de configuración. Para el primer experimento hemos seleccionado los rangos de parámetros mostrado en la Tabla 4.11, en el que todos los posibles valores se permutaron en 30 procesos de análisis. En este caso, los resultados obtenidos mostraban que

los valores de umbral menores de 0,99 no eran suficientemente precisos para la comparación de ejecutables, consiguiendo valores altos de detección de variantes pero, también, un gran número de falsos positivos. Por tanto, continuamos con un segundo experimento que utilizaba otros valores de umbral (véase la Tabla 4.12), que se permutaron en 54 procesos de análisis.

Tabla 4.12: Los parámetros utilizados para el segundo experimento de detección.

Parámetros	Valores posibles
Tamaño de secuencia	1, 2, 3
Recursividad	Recursivo (R), No Recursivo (NR)
Umbral	0.999, 0.998, 0.997, 0.996, 0.995, 0.994, 0.993, 0.992, 0.991

Como plataforma de experimentación, hemos utilizado dos máquinas diferentes: la máquina donde se encontraba la base de datos y la máquina donde NOA iba a ejecutarse. Como servidor de base de datos, hemos utilizado un *Intel Core i7 940* a 2.93 GHz con 8 GB de memoria RAM. El servidor de base de datos era un *SQL Server 2008 SP1*. Como plataforma de análisis, hemos utilizado una máquina virtual alojada en un *Intel Core i7 940* a 3.07 GHz con 12 GB de RAM. La especificación de la máquina virtual es la siguiente: 1 procesador, 1 GB de memoria RAM y Windows XP SP3 como sistema operativo.

Para evaluar el rendimiento y la precisión de NOA, hemos medido los siguiente aspectos:

- **Resultados de rendimiento.** Hemos utilizado las siguientes 4 métricas:
  - *Tiempo total de análisis.* El tiempo total requerido para analizar un ejecutable. Incluye desensamblado, representación y comparación.
  - *Tiempo de desensamblado.* El porcentaje del tiempo total que se ha empleado para desensamblar al ejecutable en análisis. Cabe remarcar que este tiempo no depende de nuestra solución ya que utilizamos el desensamblar *NewBasic* [New10] para este fin.

  - *Tiempo de representación.* El porcentaje del tiempo total utilizado para representar al ejecutable, transformándolo a vector.
  - *Tiempo de comparación.* El porcentaje del tiempo total usado para comparar el ejecutable con las representaciones de variantes de la base de datos.

- **Resultados de precisión.** Hemos evaluado los resultados de precisión midiendo la tasa de falsos negativos (*False Negative Ratio*, FNR) y la tasa de falsos positivos (*False Positive Ratio*, FPR). En particular, FNR se define como:

$$FNR = \frac{FN}{FN + TP} \tag{4.3}$$

  donde TP es el número de ejecutables maliciosos correctamente detectados y FN es el número de ejecutables maliciosos no detectados.

  FPR se define como:

$$FPR = \frac{FP}{FP + TN} \tag{4.4}$$

  donde FP es el número de ejecutables benignos incorrectamente detectados como malware mientras que TN es es el número de ejecutables benignos correctamente clasificados.

La Figura 4.21 muestra el ratio entre el tiempo total y el tamaño de la secuencia para representar así el coste del proceso de representación de ejecutables donde NR significa representación *no recursiva* mientras que R significa *recursiva*. Los resultados de tiempo para cada valor de tamaño de secuencia, expresados en milisegundos, requeridos para analizar un *megabyte* de datos. Se puede observar que el tiempo de proceso se incrementa de forma exponencial con el tamaño de secuencia.

La Figura 4.22 muestra la proporción de las tres fases de análisis (desensamblado, representación y comparación) donde NR significa representación *no recursiva* mientras que R significa *recursiva*. El tiempo requerido para la fase de comparación se incrementa de modo exponencial según incrementa el tamaño de la secuencia, mientras que las proporciones de tiempo dedicadas al desensamblado y representación disminuyen.

La Figura 4.23 representa la relación entre el tamaño del fichero y el tiempo de proceso donde NR significa representación *no recursiva* mientras

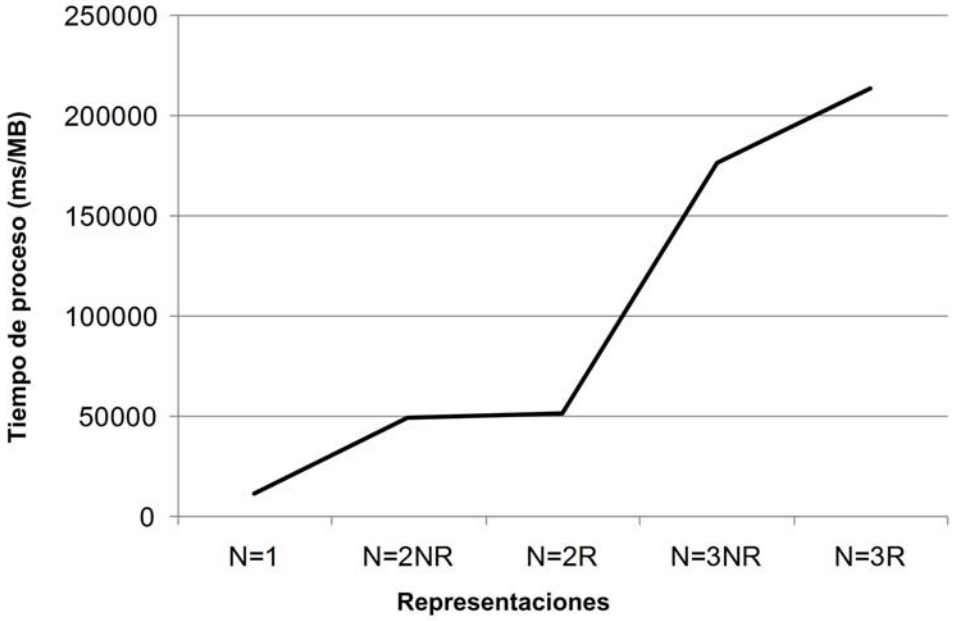

Figura 4.21: Los resultados de tiempo para cada valor de tamaño de secuencia, expresados en milisegundos, requeridos para analizar un *megabyte* de datos.

que R significa *recursiva*. Finalmente, la Figura 4.24 muestra tanto el FNR como el FPR para conjunto de parámetros utilizado en el experimento.

El tiempo de análisis depende de tres parámetros: (i) tiempo de desensamblado; (ii) tiempo de representación, que depende en el tamaño de la secuencia (cuanto más grande sea la secuencia, más posibles secuencias hay en la representación); y (iii) tiempo de comparación, que depende en el tamaño del vector generado en la fase de representación y, por tanto, en el tamaño de secuencia. El tiempo de comparación también depende del número de variantes almacenadas en la base de datos.

Por otro lado, podemos observar como el tiempo necesario para el análisis aumenta según aumenta el tamaño del ejecutable (tal y como se muestra en la Figura 4.23). Además, el tiempo de comparación es el que más proporción del total requiere (véase la Figura 4.22). Del mismo modo, el tiempo total incrementa de forma exponencial con el tamaño de la secuencia, tal y como se puede apreciar en la Figura 4.21.

Aunque la complejidad de nuestro modelo depende fuertemente de estos tres parámetros, el análisis no resulta excesivamente costoso. En concreto, el tiempo medio de análisis para cada fichero, sin tener en cuenta su

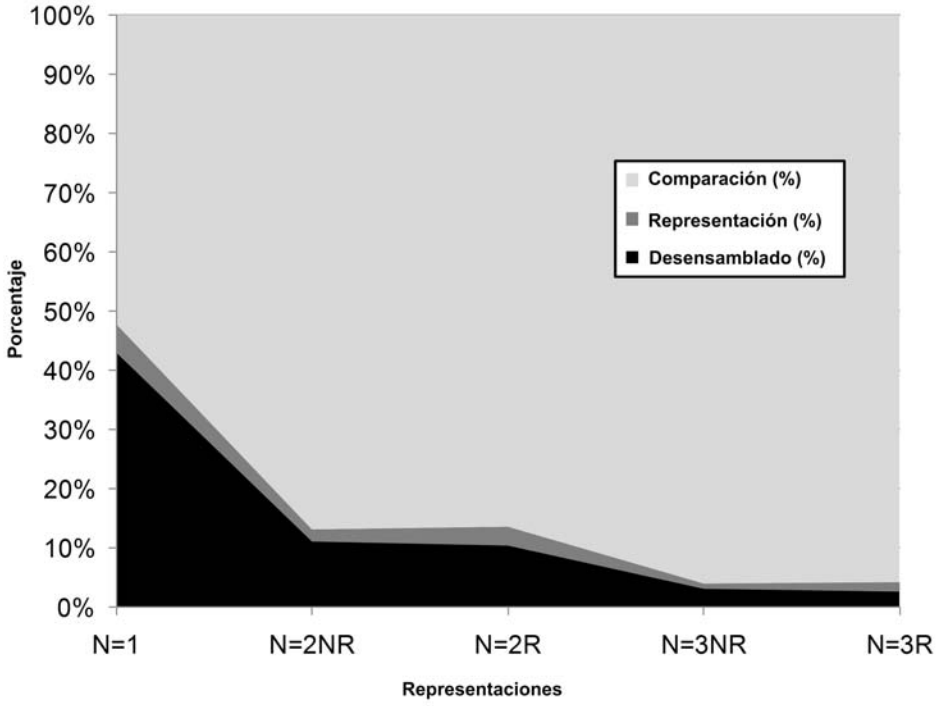

Figura 4.22: El porcentaje del análisis dedicado a cada fase.

tamaño, fue: 563,37 ms para $n = 1$; 2106,94 ms para $n = 2$; 2207,44 ms para $n = 2$ recursivo; 7554,83 ms para $n = 3$ y 8930,7 ms para $n = 3$ recursivo.

En lo referente a los resultados de precisión (véase la Figura 4.24), parece obvio pensar que cuanto mayor sea el valor del umbral, menor es la tasa FPR y mayor la tasa FNR (los ejecutables tienen que ser más similares para ser detectados), mientras que valores más bajos del umbral obtienen el efecto contrario.

Consecuentemente, es importante seleccionar un valor de umbral que equilibre estas dos tasas. Además, tenemos que seleccionar el valor de tamaño de secuencia y recursividad apropiados.

Las tasas de detección no varían notoriamente para tamaños de secuencia mayores que 2 mientras que el tiempo necesario para el análisis incrementa substancialmente. Específicamente, para garantizar una tasa de FPR de 0,0 %, tenemos que generar la representación en modo no recursivo, un tamaño de secuencia de 2 o 3 y un valor de umbral de 0,999.

Sin embargo, estas representaciones obtienen un 76,47 % de FNR (un valor bastante alto) para un tamaño de secuencia de 3. Por otro lado, el

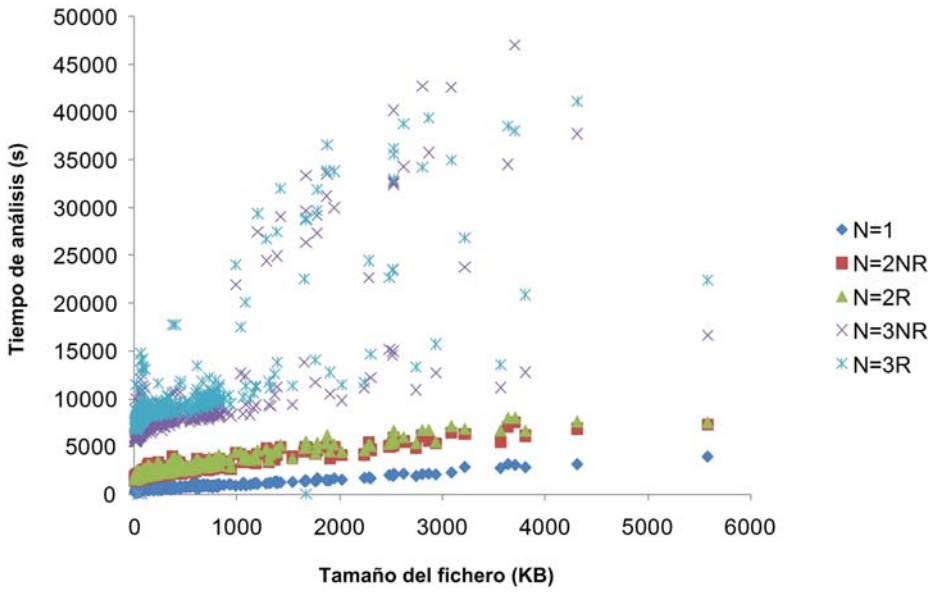

Figura 4.23: Relación entre el tamaño del fichero y el tiempo necesario para el análisis.

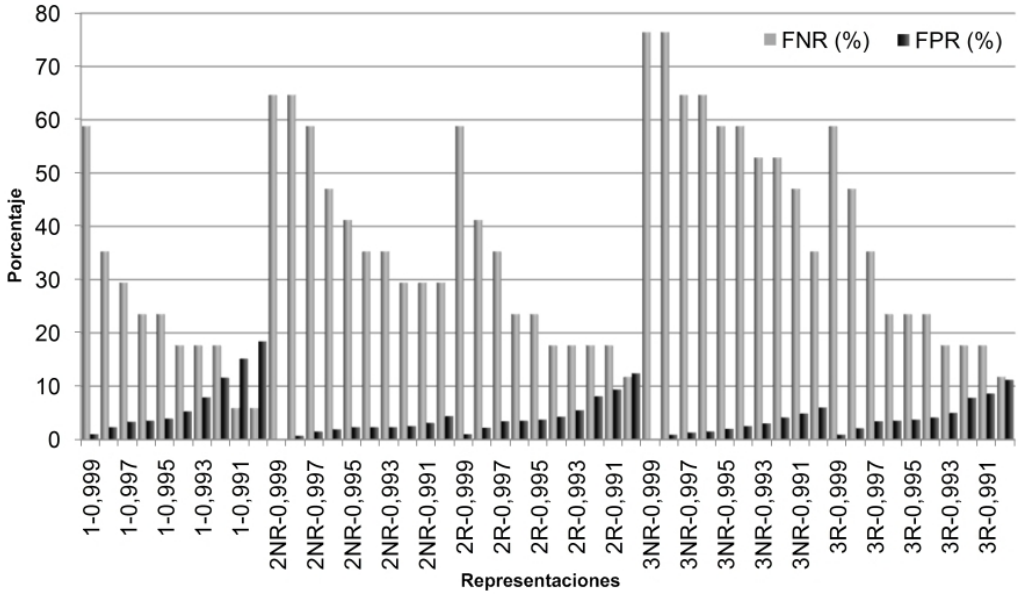

Figura 4.24: Resultados de FPR y FNR.

valor más bajo de FNR lo obtuvimos con la configuración $n = 1$ no recursiva y 0,991 como umbral con un 5,88 %.

Además, esta configuración no generó un gran número de falsos positivos: sólo el 15,2 % de los ejecutables benignos se detectaron como malware. Otra representación interesante es $n = 2$ recursivo, con un valor de umbral de 0.994: 17,64 % de FRN y 4,3 % de FPR.

4.3.4 Modelos filogenéticos

Hemos generado diferentes modelos filogenéticos de varias familias de malware. El objetivo del experimento es mostrar que es posible seguir la evolución de las familias de malware. Hemos comparado los modelos filogenéticos con el etiquetado de *VxHeavens* [VX10]. De este modo, la pregunta a responder es:

> *¿Cómo de diferente es el etiquetado de las casas de antivirus (que son similares al etiquetado de VxHeavens) respecto a nuestros modelos filogenéticos?*

Hemos utilizado diferentes familias de malware para extraer su modelo filogenético: *Antilam*, *Bancodor*, *Cmjspy*, *Gruel* y *Zokrim*. La Tabla 4.13 muestra estas familias y el número de variantes que las componen. Hemos elegido estas familias y no otras porque tienen un número bajo de variantes y, de esta forma, es más fácil representar gráficamente el dendrograma resultante.

Tabla 4.13: Descripción del conjunto de datos para el experimento de generación de modelos filogenéticos.

Familia de malware	Número de ejecutables
Antilam	6
Bancodor	10
Cmjspy	8
Gruel	12
Zokrim	17

Como configuración para la representación de los ejecutables de estas familias, hemos elegido la combinación de las secuencias de tamaño 1, 2, 3 y 4, que era la que mejores resultados globales había obtenido. Para valorar los modelos filogenéticos hemos seguido los siguientes pasos:

- **Construir la matriz de representaciones.** Para construir la matriz de las representaciones de los ejecutables, primeramente hemos desensamblado los ejecutables utilizando *NewBasic Assembler* [New10] y, seguidamente, hemos extraído la representación para la configuración seleccionada.

- **Calcular el dendrograma de la familia de malware.** Hemos utilizado *CLUTO* [Kar03] como la herramienta para generar los modelos filogenéticos de las familias. Hemos empleado la siguiente configuración:

  - *Función de similitud.* Como función de similitud, hemos utilizado la similitud del coseno [TP07].
  - *Método de clustering o agrupamiento.* Empleamos el método *aglomerativo* [Orl67], un método clásico para el *clustering* jerárquico.
  - *Función de criterio*. Hemos usado UPGMA [SS73].

- **Comparar el etiquetado previo con el modelo generado.** El etiquetado en los programas antivirus (y en *VxHeavens*) se da con la última letra del nombre de la variante y, en el caso de que no se pueda aseverar la variante pero si la familia, se añade el sufijo *.gen* (viene de genérico). Podemos comparar los modelos filogenéticos identificando los *clusters* o grupos del modelo que tengan variantes con letras cercanas como sufijo del nombre en el etiquetado de *VxHeavens* [VX10].

La Figura 4.25 muestra el modelo para la familia *Antilam*. Tiene una similitud media (habiendo hecho comparaciones de cada una de las variantes con el resto) de 0,98935. Se divide en dos grupos: uno que contiene Antilam.20.q, Antilam.20.p y Antilam.20.n; y otra con Antilam.14.i, Antilam. 20.gen (la versión genérica de Antilam) y Antilam.20.s. Finalmente, las versiones Antilam.20.p y Antilam.20.n forman un *cluster* con una similitud de 0,99843. En el mismo nivel de la jerarquía, la versión genérica de *Antilam* (Antilam.20.gen) y Antilam.14.n forman un grupo con una similitud media de 0,99635. Antilam.20.q y Antilam.20.s no forman ningún grupo a este nivel. Si comparamos el modelo con el etiquetado de *VxHeavens*, existe un grupo en el que el modelo filogenético y el etiquetado son parecidos: el grupo formado por las variantes Antilam.20.q, Antilam.20.p,y Antilam.20.n. Sin embargo, el otro grupo, formado por las variantes Antilam. 14.i y Antilam.20.gen, no se parece al etiquetado de *VxHeavens*.

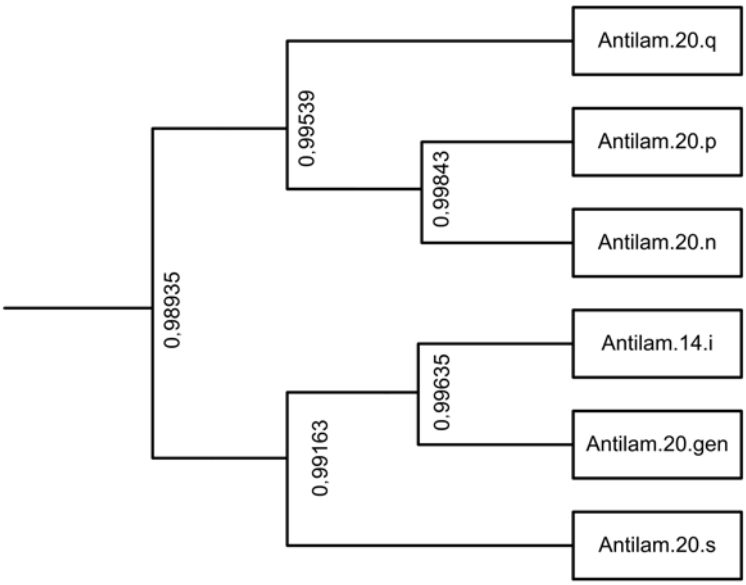

Figura 4.25: El modelo filogenético de la familia *Antilam*.

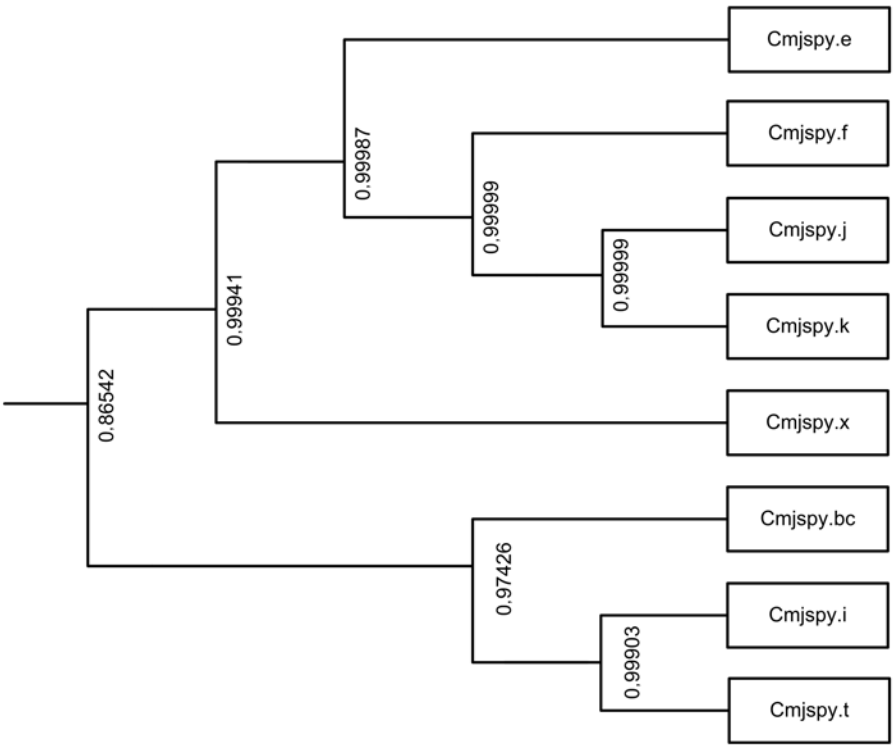

Figura 4.26: El modelo filogenético de la familia *Cmjspy*.

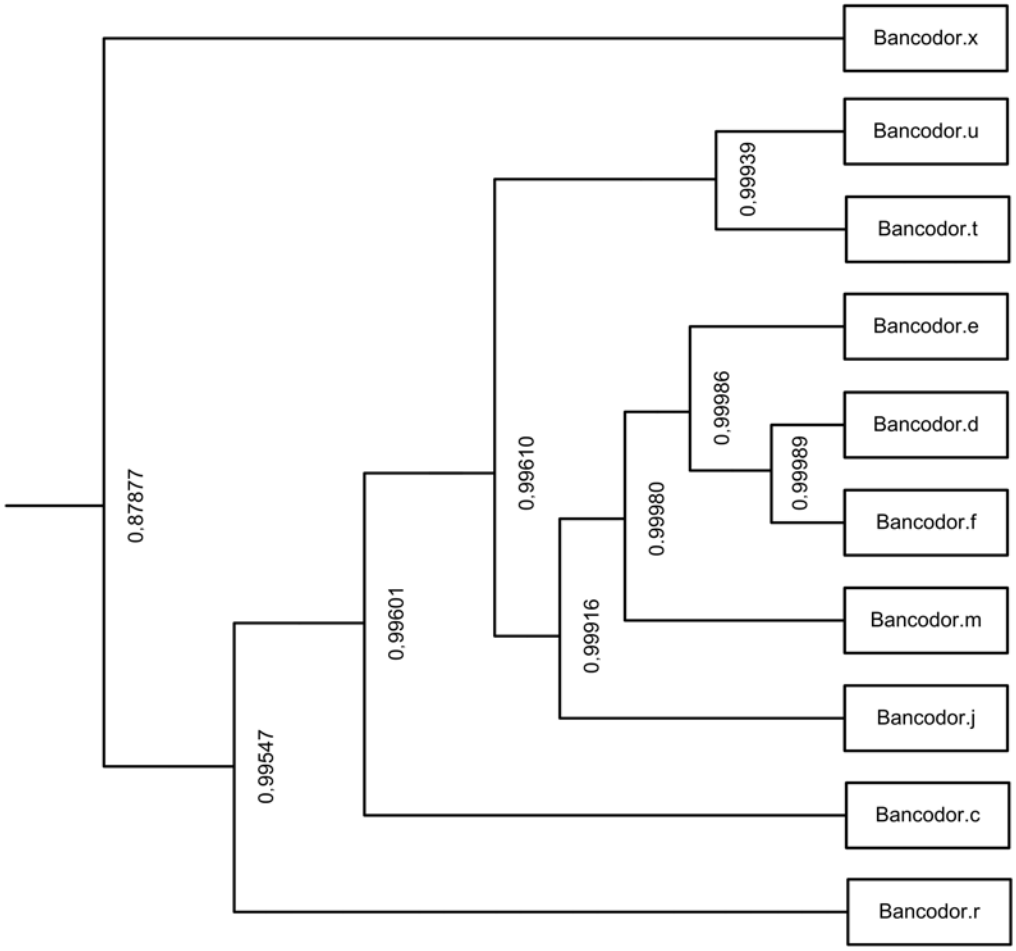

Figura 4.27: El modelo filogenético de la familia *Bancodor*.

El modelo filogenético para *Cmjspy* (véase la Figura 4.26) muestra que los miembros de la familia son similares en un 0,86542. La familia se divide en un *cluster* formado por Cmjspy.e, Cmjspy.f, Cmjspy.j, Cmjspy.k y Cmjspy.x; y en un segundo *cluster* formado por las variantes Cmjspy.bc, Cmjspy.i, y Cmjspy.t. Este último se divide en dos grupos: el primero formado únicamente por Cmjspy.x y el segundo agrupando las variantes Cmjspy.j y Cmjspy.k forman un *cluster* en el nivel más bajo de la jerarquía. Luego forman un grupo con la variante Cmjspy.f y luego otro con Cmjspy.e. Este modelo tiene varias similitudes con el etiquetado de VxHeavens: en el primer grupo, exceptuando Cmjspy.x, el resto de variantes concuerdan.

La Figura 4.27 muestra el modelo filogenético para la familia *Bancodor*. La familia tiene una similitud media de 0,87877. Existe un gran *cluster*

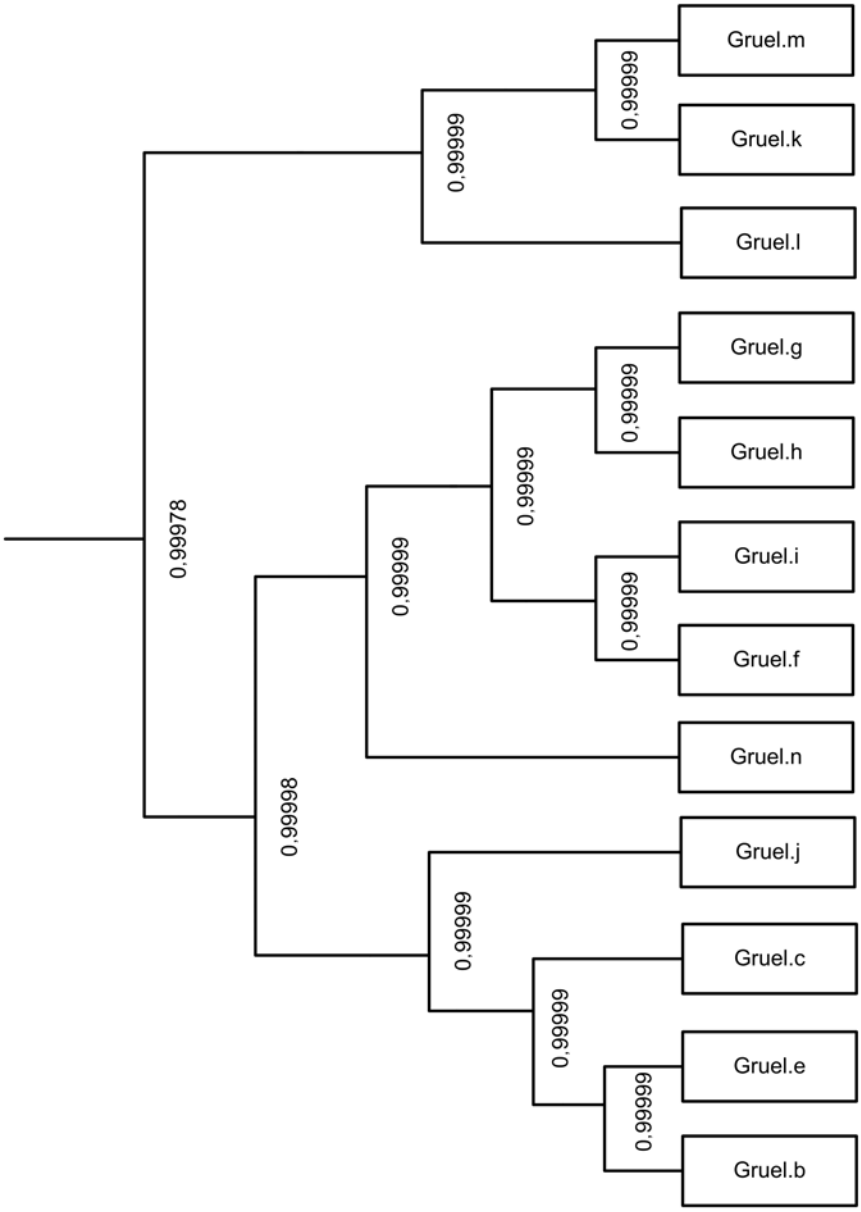

Figura 4.28: El modelo filogenético de la familia *Gruel*.

que agrupa todas las variantes con la excepción de la variante Bancodor.x. Las únicas coincidencias apreciables entre el modelo filogenético generado por nuestro método y el etiquetado realizado por *VxHeavens*, las podemos encontrar en el *cluster* formado por las variantes Bancodor.e, Bancodor.f y Bancodor.d; y en el grupo que está compuesto por los ejecutables Bancodor.t y Bancodor.u.

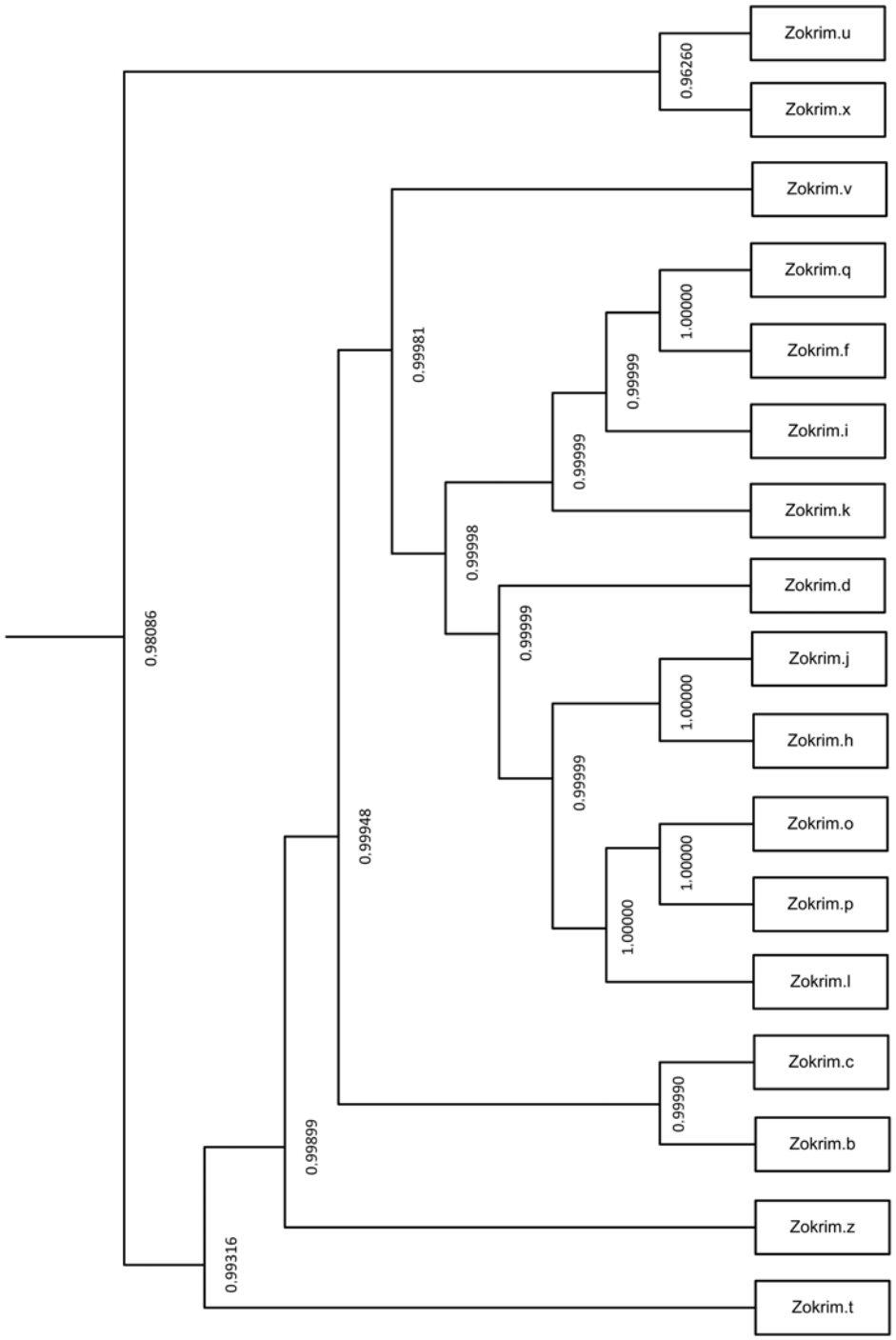

Figura 4.29: El modelo filogenético de la familia *Zokrim*.

La familia *Gruel* (véase la Figura 4.28) forma dos grupos: un grupo formado por Gruel.m, Gruel.k y Gruel.l; y otro con el resto de variantes. Este último se divide a su vez en un grupo formado por Gruel.n y otro compuesto por el resto de variantes del grupo anterior. En esta familia hay similitudes entre el etiquetado de *VxHeavens* y los grupos del modelo filogenético formados por: las variantes Gruel.c, Gruel.b y Gruel.e; las variantes Gruel.g, Gruel.h, Gruel.i, Gruel.f y Gruel.n; y las variantes Gruel.m, Gruel.k y Gruel.l.

La familia *Zokrim* (véase la Figura 4.29) está compuesta por variantes muy similares. En el modelo, existen dos grupos bien diferenciados: uno con las variantes Zokrim.x y Zokrim.u; y otro con el resto de variantes. Zokrim.t forma un *cluster* él solo en el segundo nivel, mientras que las variantes restantes se dividen en otros dos *clusters*.

El primero está formado por Zokrim.b y Zokrim.c y el segundo se divide en otros dos grupos, formando un subárbol donde uno de los arcos es Zokrim.v. El resto de variantes se divide en dos grupos con un 0,9999 de similitud. El modelo es muy parecido al etiquetado de *VxHeavens*. Hemos identificado los siguientes grupos que coinciden con el etiquetado: Zokrim.u y Zokrim.x; Zokrim.c y Zokrim.b; y Zokrim.j, Zokrim.l, Zokrim.o, Zokrim.h y Zokrim.p.

A pesar de que algunos grupos coincidían con los modelos filogenéticos, generalmente, los modelos filogenéticos no se parecen al etiquetado. De todos modos, hemos de tomar en consideración que no existe una convención sobre la nomenclatura de variantes de malware y que hay fuertes discrepancias entre un antivirus y otro [Bon04]. Para estandarizar el etiquetado, se pueden utilizar los modelos filogenéticos en conjunción con el método de Karim et ál. [KWLP05a] (véase la Figura 4.30).

Mediante el estudio de estos modelos, un analista puede identificar diferentes grupos cuyos miembros son similares y, así, seleccionar qué variantes se tienen que almacenar en la base de datos de malware.

4.4 Discusión de los resultados

Los resultados muestran que con la representación basada en secuencias de *opcodes* podemos detectar variantes de malware. Sin embargo, existen ciertas consideraciones respecto al método.

Puede parecer que nuestro enfoque está detectando el compilador utilizado para crear los ejecutables. De hecho, utilizar un compilador hace a un

Encontrar la mejor coincidencia para la nueva variante en el modelo filogenético;
if *la similitud es mayor que un límite k en una hoja l del modelo* **then**
    Dividir l en dos nuevas hojas l_1 y l_2;
    Almacenar l en l_1 y añadir la nueva variante a l_2;
end
else
    Añadir una nueva raíz r_n con dos hijos: la raíz anterior y la nueva hoja l_n;
    Añadir la variante en l_n;
end
Asignar a la variante un nuevo nombre;

Figura 4.30: Método para etiquetar variantes basado en modelos filogenéticos.

ejecutable rico en ciertas secuencias de *opcodes*. No obstante, cuando hemos seleccionado los ejecutables que iban a formar parte de conjunto de datos, los hemos analizado con PEiD, una herramienta que detecta la mayoría de empaquetadores y compiladores para los ficheros ejecutables. Después de eliminar los empaquetados (nuestro sistema no habría sido capaz de detectarlos), no existía ninguna diferencia significativa entre los compiladores utilizados para el software benigno y el malicioso. Los compiladores más usados eran *Microsoft Visual C++*, *Borland C++* y *Borland Delphi*. Sin embargo, podríamos utilizar la habilidad de detectar compiladores para nuestro propio beneficio. Podemos aplicar nuestro método para detectar si un ejecutable está empaquetado y, después, utilizando un desempaquetador, extraer el código original. Después, lo podemos analizar para determinar si es malware o no.

Además, para evaluar los modelos filogenéticos, los hemos comparado con el etiquetado de *VxHeavens* y hemos encontrado que únicamente había un pequeño número de coincidencias. No obstante, no es realmente un error en los modelos, sino una falta de consenso en la nomenclatura de una variante, debido a que ni siquiera diferentes antivirus proporcionan la misma denominación para la mima variante [KWLP05a, KWLP05b].

En conclusión, nuestro método es una nueva forma de calcular similitudes entre ficheros ejecutables y, por tanto, detectar variantes de malware. En este capítulo, hemos demostrado que este método también es capaz de detectar variantes de malware. Por consiguiente, puede ser utilizado como un reemplazo o un complemento de los motores antivirus de firmas.

4.5 Sumario

Este capítulo ha introducido un nuevo método para detectar variantes de malware. El método utiliza la representación descrita en el capítulo 3 y está compuesto de una base de datos de malware, un generador de consultas y un método de recuperación de variantes de malware.

La base de datos de malware almacena las representaciones de las variantes conocidas de malware. El generador de consultas extrae la representación de un ejecutable para formar la consulta a realizar sobre la base de datos. El sistema de recuperación de variantes utiliza la consulta para saber si el ejecutable en inspección es malware o no, recuperando las variantes más similares que existan en la base de datos. Finalmente, el generador de modelos filogenéticos construye modelos de evolución de las familias de malware que ayudan al análisis del malware, mejorando el etiquetado del malware y facilitando la selección de las variantes a incluir en la base de datos de malware.

En resumen, la validación empírica ha demostrado que nuestro modelo puede dar cobertura a un gran número de variantes de malware de una familia concreta. Por consiguiente, es un método que puede sustituir o complementar a las típicas bases de datos de firmas.

«Cuéntame y olvidaré. Enséñame y recordaré. Involúcrame y aprenderé.»
Benjamin Franklin (1706–1709)

Capítulo 5

Detección de malware desconocido

Recientes estudios han intentado afrontar el problema de detectar malware desconocido que los métodos clásicos no pueden. En concreto, los métodos basados en el aprendizaje automático son una apuesta de la comunidad científica para la detección de malware desconocido. Sin embargo, la mayoría de estos métodos se basan en distribuciones de bytes, que los creadores de malware pueden evadir fácilmente [Chr07].

Dado el estado actual de la investigación en este área, proponemos un nuevo método para la detección de malware desconocido utilizando un enfoque de aprendizaje automático supervisado. Utilizamos la representación basada en frecuencias de secuencias de *opcodes* para entrenar varios clasificadores de aprendizaje supervisado.

No obstante, los métodos completamente supervisados requieren una gran cantidad de datos etiquetados de ambas clases (malware y software benigno), que resultan en un importante esfuerzo adicional. Por tanto, proponemos también la adopción del aprendizaje de clase única (*single-class learning* en inglés) para la detección de malware desconocido, utilizando asimismo la representación vectorial basada en secuencias de opcodes. El aprendizaje de clase única sólo necesita que una de las clases esté parcialmente etiquetada, reduciendo, de este modo, los esfuerzos necesarios para el análisis previo al entrenamiento de los algoritmos. En particular, hemos

utilizado el método Roc-SVM [LL03], que se ha empleado en problemas de categorización de texto [LL03].

En resumen, las contribuciones que hemos realizado en este contexto, son las siguientes:

- Mostramos cómo utilizar una representación de frecuencias de secuencias de *opcodes* para detectar y clasificar malware desconocido utilizando aprendizaje supervisado.
- Proveemos de una validación empírica de nuestro método supervisado con un estudio extenso y exhaustivo de modelos supervisados de aprendizaje automático para detectar malware desconocido.
- Demostramos que los métodos supervisados consiguen tasas de detección altas. También debatimos las principales limitaciones del enfoque supervisado.
- Describimos cómo adoptar Roc-SVM para la detección de malware desconocido.
- Evaluamos si es mejor etiquetar software malicioso o benigno en el aprendizaje de clase única.
- Estudiamos cuál es el número de ejecutables más adecuado etiquetados para la detección en base al aprendizaje de clase única.
- Mostramos cómo los enfoques de aprendizaje de clase única pueden reducir los esfuerzos de etiquetado en la industria anti-malware, manteniendo un alto grado de precisión en la clasificación.

Lo que resta de capítulo está organizado como sigue. La sección 5.1 describe los enfoques supervisados que se han utilizado. La sección 5.2 introduce el método de aprendizaje de clase única que hemos utilizado. La sección 5.3 detalla la evaluación de ambos métodos para la detección de malware desconocido. La sección 5.4 debate ambos enfoques. Finalmente, la sección 5.5 resume las contribuciones de este capítulo.

5.1 Aprendizaje supervisado

El aprendizaje supervisado — o algoritmos de clasificación — utiliza un conjunto de datos previamente etiquetado [Kot07] (en nuestro caso con las

etiquetas: *malware* y *benigno*). En esta sección revisaremos los algoritmos de aprendizaje supervisado que hemos utilizado para la detección de malware desconocido.

5.1.1 Árboles de decisión

Los árboles de decisión son un tipo de clasificadores de aprendizaje automático que pueden ser representados gráficamente como árboles (véase la Figura 5.1). Los nodos interiores representan las condiciones o estados posibles de las variables del problema y los nodos finales u *hojas* constituyen la decisión final del clasificador [Qui86].

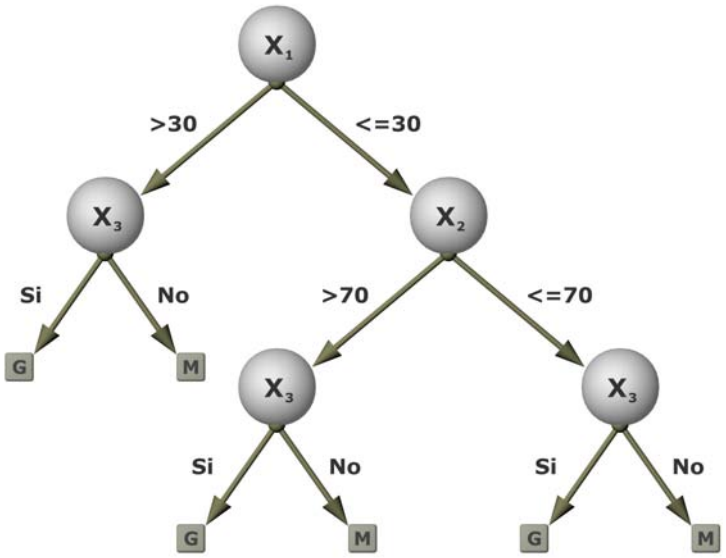

Figura 5.1: Un ejemplo de árbol de decisión. Los nodos interiores representan los diferentes estados que una variable puede tomar, mientras que los nodos finales u hojas son la decisión final que el clasificador puede tomar.

5.1.1.1 Formalización

Un árbol de decisión es, formalmente, un grafo $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ que consiste en un conjunto no vacío de nodos finitos $\mathcal{V}$ y un conjunto de aristas $\mathcal{E}$. Si el conjunto de las aristas $\mathcal{E}$ está compuesto a su vez por bi-tuplas ordenadas de nodos de la forma (v, w), entonces se puede decir que el grafo $\mathcal{G}$ es dirigido.

Un *camino* en el grafo $\mathcal{G}$ se define como una secuencia de aristas de la forma $(v_1, v_2), (v_2, v_3), ..., (v_{n-1}, v_n)$. Los caminos se pueden expresar por el origen, su final y la distancia recorrida (el mínimo número de aristas desde el origen hasta el final). Además, si (v, w) es una arista en el árbol, entonces se considera a v *nodo padre* de w, mientras que w sería el *nodo hijo* de v. El único nodo en el árbol sin padres se denomina *nodo raíz*. Cualquier otro nodo en el árbol, es un *nodo interno*. Para construir la representación gráfica de un árbol, se contesta a un conjunto de preguntas binarias (de sí o no).

Hay varios algoritmos de aprendizaje supervisado, que se utilizan para aprender la estructura de un árbol mediante un conjunto de datos etiquetados. En este trabajo, hemos utilizado *random forest* (en castellano, bosque aleatorio), que es una combinación o *ensemble* (combinación de varios clasificadores en uno más fuerte) de árboles de decisión construidos al azar. Además, hemos utilizado J48, que es la implementación de WEKA[1] [Gar95] del algoritmo C4.5 [Qui93].

5.1.1.2 **Algoritmo C4.5**

C4.5 [Qui93] es un algoritmo que genera un árbol de decisión y es una extensión del algoritmo ID3 [Qui86]. Este algoritmo construye árboles de decisión utilizando un conjunto de datos etiquetados y mediante el concepto de *entropía de la información*, que mide la incertidumbre asociada con una variable aleatoria. El término se suele referir a la *entropía de Shannon* [Sha51, SW49], que mide el valor esperado de información (*Information Gain*, IG) dentro de un mensaje.

Para generar un árbol de decisión mediante el algoritmo C4.5 (véase la Figura 5.2), se requiere un conjunto de datos de entrenamiento $\mathcal{S} = \{s1, s2, ..., s_{n-1}, s_n\}$ previamente etiquetado. Cada espécimen del conjunto de datos $\mathcal{S}$, $s_i = \{x_1, x_2, ..., x_{m-1}, x_m\}$ es un vector donde cada x_i representa los atributos o características del mismo. El conjunto de datos de entrenamiento se complementa con un vector $\mathcal{C} = \{c_1, c_2, ..., c_{n-1}, c_n\}$ donde cada c_i indica a qué clase pertenece el espécimen i.

El algoritmo utiliza ciertos casos base o configuraciones para comenzar:

- $\forall s_i \in S : s_i \in c_j$, ó, en otras palabras, todo espécimen en el conjunto de datos de entrenamiento pertenece a la misma clase. En este caso, el algoritmo tan sólo crea un nodo *hoja* en el árbol, donde se selecciona esa clase concreta.

[1] WEKA es una herramienta de aprendizaje automático desarrollada por investigadores de la Universidad de Waikato que es muy utilizada en la comunidad científica.

```
entrada: El conjunto de variables del árbol τ
Comprobar los casos base;
// Poner la variable que almacena el máximo IG a cero
mejor ← 0;
// Para cada variable (nodo) en el árbol
foreach τ_i ∈ τ do
    // Encontrar un valor de IG para dividir por él
    if IG(τ_i, 𝒞) > mejor then
        mejor ← IG(τ_i, 𝒞);
    end
    s ← Los sub-árboles que se dividen por mejor;
    // Para cada sub-árbol en el árbol
    foreach s_i ∈ s do
        if s_i = then
            // Se añade s_i al grupo de hojas
        end
        else
            C4,5(s_i);
        end
    end
end
```

Figura 5.2: Pseudo-código de C4.5.

- $\forall x_j : IG(j) = 0$ ó, en otras palabras, ninguna de las características ha conseguido un valor de IG. En este caso, C4.5 crea un nodo de decisión en lo alto del árbol utilizando el valor esperado de la clase.
- $\exists s_i \in S : s_i \in c_j \wedge c_j \notin \mathscr{C}$ ó, en otras palabras, el algoritmo encuentra una instancia de una clase previamente desconocida. De nuevo, el algoritmo crea un nodo de decisión en lo alto del árbol utilizando el valor esperado.

Para cada nodo del árbol, el algoritmo C4.5 elige el atributo dentro de los datos que, de forma más eficiente, divide el conjunto restante de especímenes en subconjuntos de una clase o de la otra. El criterio para dividir el conjunto es la *ganancia de información* o *Information Gain* (IG), la cual calcula la dependencia estadística entre dos variables aleatorias. De este modo, el atributo con el mayor IG se selecciona para ser el nodo que divida

el conjunto de datos. Este proceso se repite recursivamente hasta que no se puedan realizar más divisiones del conjunto de datos.

C4.5 se diferencia de ID3 en que es capaz de manejar tanto atributos continuos como discretos, datos sin completar, atributos con diferentes costes y, además, puede podar[1] los árboles después de generarlos.

5.1.1.3 El método *random forest*

Random forest (o bosque aleatorio) [Bre01] es un clasificador del tipo de árbol de decisión que está compuesto, a su vez, de varios sub-árboles de decisión. Para la clasificación de una instancia, genera un vector de entrada para cada uno de los árboles de la combinación. Cada árbol genera una clasificación y el clasificador global elige aquélla que ha obtenido la mayor cantidad de votos.

entrada: El conjunto de datos τ, el tamaño del conjunto de datos ℓ, el número de árboles aleatorios a generar N y el número de variables del conjunto de datos M

// El tamaño del conjunto de datos de entrenamiento para el árbol es el mismo que el tamaño del conjunto de datos original

$n \leftarrow N$;

// El conjunto de datos para el árbol es el conjunto de las N muestras aleatorias

$t \leftarrow N$ *muestras aleatorias*;

foreach *nodo* $\in$ *Tree* **do**

  n es el nodo en el árbol que maximiza IG;

  // El árbol se divide por el nodo n

  Dividir(Árbol,n) ;

end

// Se genera cada árbol y no se poda

Generar árbol;

Figura 5.3: Generación de los árboles de decisión con el algoritmo *random forest*.

Para generar los árboles de decisión aleatorios que componen a un *random forest* (véase la Figura 5.3 para el pseudo-código del algoritmo), se elige

[1] La poda de árboles de decisión es un proceso de optimización de los mismos.

un subconjunto aleatorio del conjunto total de datos. Para ello, el método utiliza una discriminación estocástica [Kle96].

Seguidamente, se calcula la mejor división posible para ese subconjunto de características. Se repite el proceso para cada nodo del árbol de decisión y, finalmente, cuando el árbol está creado, no se hace una poda del mismo (al contrario que en la mayoría de métodos para la construcción de árboles de decisión).

El algoritmo *random forest* produce un clasificador preciso para muchos conjuntos de datos, manteniendo su precisión incluso cuando existe un gran número de variables de entrada. Además, estima la importancia de las mismas a la hora de determinar la clasificación final y es capaz de enfrentarse a datos incompletos.

Sin embargo, suelen ser bastante susceptibles a sobreajustarse[1] al conjunto de datos, especialmente en tareas con datos ruidosos[2] [SLTW04].

5.1.2 El método *Support Vector Machines* (SVM)

Los clasificadores SVM se basan en un hiperplano que divide una representación n-dimensional de los datos, en dos regiones. Utilizando un conjunto de datos de entrenamiento, un SVM genera un modelo lineal capaz de predecir si un nuevo espécimen (cuya categoría nos es desconocida) pertenece a una u otra categoría.

Se define el hiperplano como aquél que maximiza el margen (m) entre las dos regiones o clases en el espacio. Este margen se define a su vez como la distancia más larga entre los ejemplos de las clases, calculada a partir de la distancia de los especímenes más cercanos al hiperplano (véase la Figura 5.4).

5.1.2.1 Formalización

Para su representación más básica, SVM requiere un conjunto de datos de entrenamiento $\mathscr{D}$ de la forma:

$$\mathscr{D} = \{(x_i, c_i) x_i \in \mathbb{R}^p, c_i \in \{-1, 1\}\}_{i=1}^{n} \tag{5.1}$$

[1] El problema de sobreajuste ocurre cuando un clasificador ofrece unos resultados óptimos para el conjunto de entrenamiento pero no es capaz de adaptarse a otros.

[2] Se consideran datos ruidosos a los datos que no están debidamente recogidos, por lo que puede considerarse errores que hacen más difícil la clasificación.

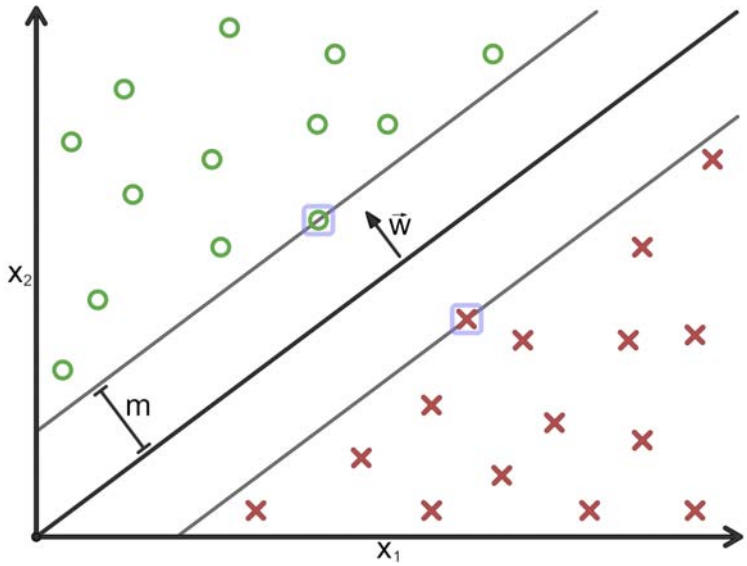

Figura 5.4: Ejemplo de un clasificador SVM en un espacio bi-dimensional.

donde c_i indica la clase del punto x_i y puede ser 1 ó -1. Cada x_i es un vector de números reales en un espacio p-dimensional. El objetivo del algoritmo es encontrar el hiperplano de máximo margen, que divide los puntos $c_i = 1$ de los $c_i = -1$ en el espacio formado por $\mathscr{D}$. Se puede representar el hiperplano como el conjunto de puntos x_i que satisfagan la siguiente condición:

$$w \cdot x - m = 0 \tag{5.2}$$

donde la operación «·» denota el producto escalar entre los dos vectores: w y x. El vector w es un vector normal y es ortogonal respecto al hiperplano mientras que el parámetro $\frac{m}{\|w\|}$ (que se puede obtener a partir de la ecuación 5.2) determina el desplazamiento del hiperplano desde el origen a través del vector normal w. Los dos hiperplanos (uno para cada clase c_i) se pueden describir por las ecuaciones:

$$w \cdot x - m = 1 \tag{5.3}$$

y

$$w \cdot x - m = -1 \tag{5.4}$$

Cuando el conjunto de datos se puede separar de forma lineal, el método selecciona los hiperplanos del margen de tal modo que no haya puntos entre ellos y, después, intenta maximizar la distancia entre estos. Utilizando geometría, la distancia entre estos dos hiperplanos es $\frac{2}{\|w\|}$, siendo el objetivo

minimizar $\|w\|$. Como el algoritmo tiene también que prevenir que existan puntos del espacio dentro del margen, se añade la siguiente restricción que depende de la clase del punto: para cada x_i de la primera clase, i está sujeto a:

$$w, x_i - m \geq +1 \tag{5.5}$$

mientras que para cada x_i de la segunda clase, i está sujeto a:

$$w, x_i - m \leq -1 \tag{5.6}$$

Estas dos ecuaciones se pueden reformular como:

$$c_i(w \cdot x_i - m) \geq 1, \quad \text{para todo } 1 \leq i \leq n \tag{5.7}$$

El problema de SVM es minimizar $\|w\|$ (en w, m) sujeto a $c_i(w \cdot X_i - m) \geq 1$. El problema se resuelve en términos de programación cuadrática[1] ya que resolverlo en otros términos resulta muy difícil porque depende en $\|w\|$, la normal de w, que supone una raíz cuadrada. Para resolver por programación cuadrática, se expresa, en primer lugar, como:

$$\min_{w,m} \max_{\alpha} \left(\frac{1}{2}\|w\|^2 - \sum_{i=1}^{n} \alpha_i [c_i(w \cdot x_i - m) - 1] \right) \tag{5.8}$$

en otras palabras, el problema ahora es buscar un punto de referencia. Para su transformación, se ha sustituido $\|w\|$ con $\frac{1}{2}\|w\|^2$ (el factor de 1/2 se utiliza matemáticamente según la conveniencia de cada caso) y se han añadido las restricciones α_i de la forma primaria correspondiente del problema cuadrático. El problema trata de minimizar $\frac{1}{2}\|w\|^2$ (para cada $i = 1, \ldots,$ n) en (w, b) sujeto a:

$$c_i(w \cdot x_i - b) \geq 1 \tag{5.9}$$

Por tanto, todos los puntos se pueden separar como $c_i(w \cdot x_i - m) - 1 > 0$ ajustando el α_i correspondiente a cero. La solución se puede expresar en términos de combinación lineal de los vectores de entrenamiento, tal que:

$$w = \sum_{i=1}^{n} \alpha_i c_i x_i \tag{5.10}$$

[1] La programación cuadrática es un caso particular de programación matemática no lineal.

Unos pocos α_i serán mayores que cero. Los x_i correspondientes son los vectores de soporte, donde cae el margen, y cumplen $c_i(w \cdot x_i - m)$ además:

$$w \cdot x_i - m = 1/c_i = c_i \iff n = w \cdot x_i - c_i \tag{5.11}$$

lo que nos permite definir el desplazamiento m. En la práctica, suele ser más robusto calcular la media sobre los N_{SV} vectores de soporte:

$$m = \frac{1}{N_{SV}} \sum_{i=1}^{N_{SV}} (w \cdot x_i - c_i) \tag{5.12}$$

El problema de optimización para encontrar w y m se puede formular en terminos de productos escalares x_i, x_j entre los puntos del conjunto de datos, lo que lleva a una clasificación lineal.

Generalmente, el espacio no puede ser divido por un hiperplano lineal y, por tanto, en vez de emplear productos escalares, se utilizan las llamadas *funciones de kernel*. Estas funciones llevan a clasificaciones no lineales como polinomiales, radiales o sigmoidales [AW99].

5.1.2.2 Funciones de *kernel*

Si se utiliza la *forma dual*[1] de la representación cuadrática para representar al hiperplano, se revela que la tarea de clasificación es únicamente una función de los vectores de soporte [Bis06].

Debido a que $\|w\|^2 = w \cdot w$ y sustituyendo $w = \sum_{i=0}^{n} \alpha_i c_i x_i$, se muestra que la forma dual del SVM se condensa para maximizar en α_i:

$$\begin{aligned} \tilde{L}(\alpha) &= \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i,j} \alpha_i \alpha_j c_i c_j x_i^T x_j \\ &= \sum_{i=1}^{n} \alpha_i - \frac{1}{2} \sum_{i,j} \alpha_i \alpha_j c_i c_j k(x_i, x_j) \end{aligned} \tag{5.13}$$

sujeto a

$$\alpha_i \geq 0 \tag{5.14}$$

y a la condición de minimizar en el margen m

[1] En programación lineal, el problema primario y la forma dual con complementarios. Una solución a cualquiera de ellos determina la solución a ambos.

$$\sum_{i=1}^{n} \alpha_i c_i = 0 \tag{5.15}$$

Por tanto, un *kernel* se define por $k(x_i, x_j) = x_i \cdot x_j$ que, en este caso, es el *kernel* lineal. En esta tesis, utilizamos las siguientes funciones de *kernel*:

- ***Kernel* polinomial** [AW99] sustituye el producto escalar con

$$k(x_i, x_j) = (x_i \cdot x_j)^d \tag{5.16}$$

  donde d representa el grado del polinomio.

- ***Kernel* polinomial normalizado** [AW99] sustituye el producto escalar con

$$K(x_i, x_j) = \frac{(x_i \cdot x_j)^d}{\sqrt{(x_i \cdot x_j)^d}} \tag{5.17}$$

  donde d representa el grado del polinomio.

- ***Kernel Radial Basis Function* (RBF)** [AW99] sustituye el producto escalar con

$$k(x_i, x_j) = \exp(-\gamma \cdot \|x_i - x_j\|^2) \text{ para } \gamma > 0 \tag{5.18}$$

  donde el producto escalar se sustituye por una función de base radial que es una función cuyo valor depende sólo de la distancia respecto al origen γ.

- ***Kernel Pearson VII*** [UMB06] sustituye el producto escalar por la función

$$k(x_i, x_j) = \frac{1}{\left[1 + \left(\frac{\sqrt{\|x_i - x_j\|^2} \cdot \sqrt{2^{\frac{1}{\omega}} - 1}}{\delta}\right)^2\right]^{\omega}} \tag{5.19}$$

5.1.3 El algoritmo *K-Nearest Neighbours* (KNN)

El algoritmo KNN [FH52] es uno de los clasificadores más simples del aprendizaje automático supervisado. Este método clasifica una instancia en función de las instancias más cercanas a ésta en el espacio de entrenamiento.

En la fase de entrenamiento de este algoritmo, se utiliza un conjunto de instancias de entrenamiento $\mathcal{S} = \{s_1, s_2, ..., s_{\ell-1}, s_\ell\}$ donde ℓ es el número de variables de cada instancia (p. ej., la frecuencia de aparición de cada secuencia de *opcodes*). Estos datos se representan en un espacio de entrenamiento (véase la Figura 5.5).

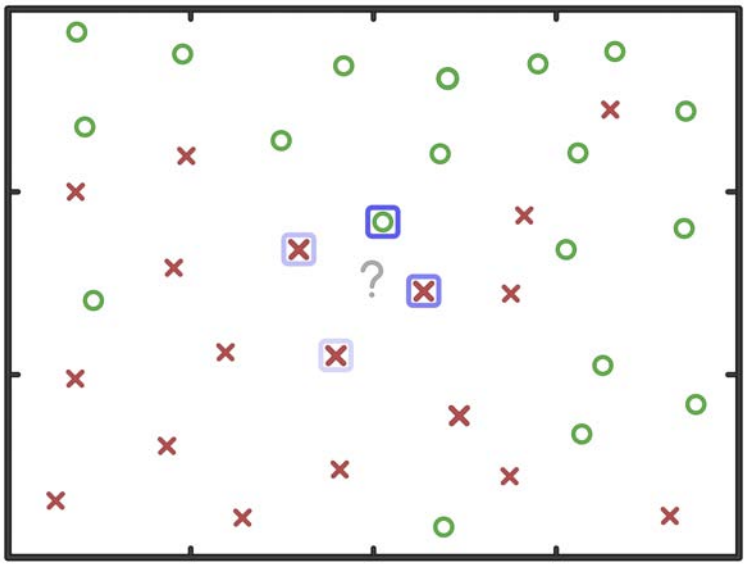

Figura 5.5: Ejemplo de un KNN en un espacio bi-dimensional.

En la fase de clasificación se mide la distancia entre los especímenes del conjunto de datos de entrenamiento y la instancia a clasificar. Esta distancia se puede medir utilizando cualquier métrica de distancia. En nuestro caso, empleamos la distancia euclidiana:

$$\sqrt{\sum_{i=0}^{n}(X_i - Y_i)^2} \tag{5.20}$$

Existen varios métodos para determinar la clase de una instancia. La técnica más utilizada es simplemente elegir la clase de la instancia como la clase más común entre los k puntos del espacio de entrenamiento más cercanos al espécimen a clasificar.

5.1.4 Redes bayesianas

El teorema de Bayes [Bay63] es la base para la inferencia bayesiana, un método estadístico de razonamiento que determina, basándose en un número de observaciones, la probabilidad de que una cierta hipótesis sea cierta. El teorema de Bayes ajusta la probabilidades tan pronto como existan nuevas observaciones.

$$P(A|B) = \frac{P(B|A) \cdot P(A)}{P(B)} \tag{5.21}$$

De acuerdo con su formulación clásica (véase la ecuación 5.21), dados dos eventos A y B, la probabilidad condicionada $P(A|B)$ de que ocurra A si ha ocurrido B, se puede obtener si sabemos la probabilidad de que ocurra A, $P(A)$, la probabilidad de que ocurra B, $P(B)$, y la probabilidad condicionada de B dado A, $P(B|A)$.

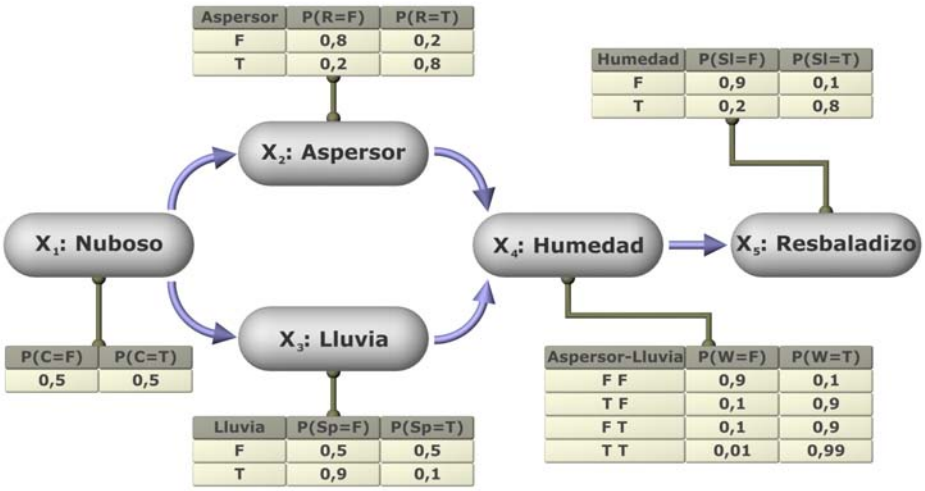

Figura 5.6: Ejemplo de red bayesiana.

Extendiendo este enfoque, las redes bayesianas (véase la Figura 5.6) son modelos probabilísticos para el análisis multivariable, basados en el teorema de Bayes. Formalmente, son grafos dirigidos acíclicos con una función de distribución probabilística asociada [CGH96]. Los nodos en el grafo representan variables (que pueden ser tanto una premisa como una conclusión) mientras que las aristas representan dependencias condicionales entre las variables. La función de probabilidad ilustra la fuerza de las relaciones [CGH96].

Para nuestras necesidades, la capacidad más importante de las redes bayesianas es su habilidad para inferir la probabilidad de que cierta hipótesis sea cierta (p. ej., la probabilidad de un ejecutable de ser malware).

5.1.4.1 **Formalización**

Supongamos que una red bayesiana $\mathscr{B}$ es una bi-tupla tal que $\mathscr{B} = (\mathscr{D}, \mathscr{P})$, donde $\mathscr{D}$ es un grafo acíclico dirigido y $\mathscr{P} = \{p(x_1|\Psi_2), ..., p(x_n|\Psi_n)\}$ es el conjunto compuesto por n funciones de probabilidad condicional, una para cada variable, donde Ψ_i es el conjunto de los nodos padre del nodo $x_i \in \mathscr{D}$. El conjunto $\mathscr{P}$ se puede definir como la función de distribución conjunta de probabilidad (véase la ecuación 5.22):

$$P(x) = \prod_{i=1}^{n} p(x_i|\Psi_i) \tag{5.22}$$

Existen varios algoritmos para el aprendizaje de la estructura de una red bayesiana. En esta tesis, utilizamos K2 [CH91], *hill climbing* o *escalar colinas* [RN03] y *Tree Augmented Naïve* (TAN) [GGP+97], además del clasificador *Naïve Bayes* (clasificador bayesiano ingenuo) [Lew98].

5.1.4.2 **Método de aprendizaje *hill climbing***

Hill climbing [RN03] es un método de búsqueda local (véase la Figura 5.7 para el pseudo-código del algoritmo *hill climbing*). El algoritmo intenta maximizar una función heurística $f(x)$. Con este fin, comienza con una solución aleatoria e iterativamente realiza pequeños cambios a ésta, consiguiendo mejoras en cada iteración.

Cuando el algoritmo no puede conseguir más mejoras en la función heurística, éste finaliza. A pesar de que el objetivo es encontrar una solución cercana a la óptima, no se puede garantizar que el algoritmo *hill climbing* encuentre la mejor solución y, en muchas ocasiones, se obtiene un máximo local.

Como algoritmo de aprendizaje bayesiano, añade, borra, y da la vuelta a las aristas basándose en una función heurística. La búsqueda no se restringe por un orden en las variables (al contrario que K2).

La heurística que se suele emplear para buscar las relaciones entre los nodos, es la función de puntuación bayesiana (denominada también *Bayesian scoring function* [DGP08].

```
nodoActual = nodoDeComienzo;
while verdadero do
    L ← NodosCercanos(nodoActual);
    siguienteEvaluacion ← −∞;
    nodoSiguiente ← NULL;
    foreach x ∈ L do
        if Eval(x) > siguienteEvaluacion then
            nodoActual = null;
            siguienteEvaluacion = Eval(x);
        end
    end
    if siguienteEvaluacion <= Eval(nodoActual) then
        // Devolver el nodo actual ya que no existen nodos
           mejores cerca
    end
    nodoActual ← nodoSiguiente;
end
```

Figura 5.7: Algoritmo *hill climbing*.

5.1.4.3 Algoritmo de aprendizaje K2

El algoritmo K2 (para una descripción algorítmica del algoritmo refiérase a la Figura 5.8) busca la estructura de red más probable dado un conjunto de datos. Para esto, el algoritmo adapta un algoritmo *hill climbing* restringiendo el orden en las variables [GGP+97].

La diferencia de K2 con el *hill climbing* sencillo es que este algoritmo no considera las aristas de la primera red ingenua para borrarlas posteriormente.

El algoritmo K2 comienza asumiendo que un nodo no tiene padre. Seguidamente, añade de un modo incremental los padres cuya adición incremente la probabilidad de la estructura resultante (véase la ecuación 5.23).

$$f(i,\pi_i) = \prod_{j=1}^{q_i} \frac{(r-1)!}{(N_{i,j}+r_i-1)!} \cdot \prod_{k=1}^{r_i} \alpha_{i,j,k}! \tag{5.23}$$

donde π_i es el conjunto de padres del nodo x_i. q_i es el cardinal del conjunto phi_i, que contiene todas las posibles instancias de los nodos padre de x_i en el conjunto de datos $\mathscr{D}$. Por otro lado, r_i es el cardinal del conjunto V_i que

entrada: Un conjunto de n nodos, un orden en los nodos, un límite en el número de nodos padre que otro nodo pueda tener u y un conjunto de datos $\mathscr{D}$ con m casos
for $i \leftarrow 0$ **to** n **do**
  // π_i es el conjunto de nodos padre del nodo x_i
  $\pi_i \leftarrow 0$;
  // La f se calcula mediante la ecuación 5.23
  $P_{viejo} \leftarrow f(i, \pi_i)$;
  $Continuar \leftarrow verdadero$;
  while $Continuar \wedge |\pi_i| < u$ **do**
    z es el nodo en $Pred(x_i) - \pi_i$ que maximiza $f(i, \pi_i)$;
    $P_{nuevo} \leftarrow f(i, \pi_i)$;
    if $P_{nuevo} > P_{viejo}$ **then**
      $P_{nuevo} \leftarrow P_{viejo}$;
      $\pi_i \leftarrow \pi_i \cup \{z\}$;
    else
      $Continuar \leftarrow falso$;
    end
  end
  escribir('Nodo: ', x_i,' Padre de x_i: ',π_i);
end

Figura 5.8: Algoritmo K2.

contiene todos los posibles valores para el nodo x_i. $\alpha_{i,j,k}$ es el número de especímenes dentro de $\mathscr{D}$ donde un atributo x_i se instancia con su valor k-ésimo, y los nodos padre de x_i en π_i se instancian con la j-ésima instancia en ϕ_i.

Ocasionalmente, el algoritmo termina de añadir padres a los nodos. Esto sucede cuando al añadir un nodo padre no se incrementa la probabilidad de la estructura de la red [LRS04] y, por tanto, el algoritmo finaliza su ejecución, dando como resultado la estructura de la red bayesiana final.

5.1.4.4 Clasificador *Naïve Bayes*

Un clasificador bayesiano ingenuo (*Naïve Bayes*) asume que no existe ningún tipo de dependencia entre los nodos de la red bayesiana [CGH96]. Por tanto, cada nodo en la red está únicamente vinculado con el nodo conclusión de la misma.

El clasificador combina el modelo bayesiano con una regla de decisión (véase la ecuación 5.24):

$$\underset{c \in C}{\text{argmáx}}\ P(c) = \prod_{i=1}^{n} P(F_i = f_i | C = c) \tag{5.24}$$

donde cada f_i corresponde a cada una de las características conocidas del espécimen a clasificar.

5.1.4.5 El método *Tree Augmented Naïve* (TAN)

En el método TAN, cada nodo tiene a lo sumo dos nodos padre (uno de ellos es el nodo conclusión). Concretamente, el algoritmo TAN consta de los siguientes pasos:

1. Calcular la información mutua (véase la ecuación 5.25) $I_{\hat{P_D}}(A_i; A_j|C)$ entre cada par de atributos, $i \neq j$.
2. Construir un grafo completo y no dirigido en el que los nodos son los atributos de la forma $A_1, \ldots, A_n$.
3. Anotar el peso de cada arista que conecte un atributo A_i a otro atributo A_j por $I_{\hat{P_D}}(A_i; A_j|C)$.
4. Construir un sub-árbol que sea máximo ponderado (*weighted maximum spanning tree*) [GGP+97].
5. Transformar el árbol no dirigido resultante en uno dirigido eligiendo un nodo raíz y configurando la dirección saliente del mismo.
6. Construir un modelo TAN añadiendo un nodo con la etiqueta C y añadiendo una arista desde C hasta A_i.

TAN utiliza la información mutua condicional (véase la ecuación 5.25) entre los atributos dada la clase que mide la información que Y da acerca de X cuando el valor de Z es desconocido.

$$I_P(X; Y|Z) = \sum_{x,y,z} \sum P(x, y, z) \cdot \log\left(\frac{P(x, y|z)}{P(x|z) \cdot P(y|z)}\right) \tag{5.25}$$

TAN es mejor que *Naïve Bayes* en términos de precisión y, sin embargo, mantiene una estructura considerablemente sencilla [GGP+97].

5.2 Aprendizaje de clase única

Los algoritmos supervisados de aprendizaje automático requieren una gran cantidad de ejecutables etiquetados de las dos clases (en nuestro caso, de malware y software benigno). No obstante, en un entorno real, como es el análisis de código malicioso, el obtener esa gran cantidad de datos etiquetados resulta difícil. Para generar estos datos, se necesita un proceso costoso de análisis y, durante éste, algunos ejecutables maliciosos pueden no ser detectados. Dentro del aprendizaje automático, se han propuesto diferentes enfoques capaces de afrontar este problema.

El aprendizaje semi-supervisado es un tipo de técnica de aprendizaje automático especialmente útil cuando existe una cantidad limitada de datos para cada clase. Estas técnicas crean un clasificador supervisado basándose en los datos etiquetados y predicen la etiqueta para todas las instancias no etiquetadas. Los especímenes cuyas clases se hayan predicho superando cierto nivel de certidumbre, se añaden al conjunto de datos etiquetados [CSZ06]. El proceso se repite hasta que ciertas condiciones de fin se satisfacen (un criterio muy utilizado es el de *máxima verosimilitud* encontrado por la técnica *Expectation-Maximisation* (EM) [Moo96]).

Desde el punto de vista práctico, estos enfoques mejoran la precisión final de los métodos completamente no supervisados (en los que no se requiere etiqueta ninguna y buscan agrupaciones en los datos [KP04]). Sin embargo, los métodos semi-supervisados requieren un mínimo de datos etiquetados de cada clase y, por consiguiente, no son aplicables en dominios donde sólo una de las clases está siendo debidamente etiquetada. Por ello, aunque pueden ser considerados como una primera opción para reducir el esfuerzo del etiquetado de software, existen otras alternativas que reducirían aún más el esfuerzo y no requieren etiquetas en ambas clases.

De este modo, los conjuntos de datos que sólo contienen instancias etiquetadas de una clase y que se utilizan para el aprendizaje automático se denominan *conjuntos de datos parcialmente etiquetados* [LL03]. La clase que contiene las instancias etiquetadas se denomina *clase positiva* [LDL+03]. A la generación de clasificadores utilizando este tipo de conjunto de datos se le conoce como *aprendizaje de clase única* (*single-class learning*) o *aprendizaje a partir de datos positivos y no etiquetados* (*learning from positive and unlabelled data*) [LL03].

Dado este contexto, proponemos la adaptación del aprendizaje de clase única para la detección de malware desconocido. Debido a que la cantidad de malware está creciendo cada año, la tarea de etiquetar malware es

// Asignar al conjunto no etiquetado $\mathscr{U}$ la clase negativa, y al conjunto etiquetado $\mathscr{P}$ la clase positiva
foreach $e \in \mathscr{U}$ **do**
  $e \leftarrow \vec{c^-}$;
end
foreach $e \in \mathscr{P}$ **do**
  $e \leftarrow \vec{c^+}$;
end
$\vec{c^+} \leftarrow \alpha \frac{1}{|\mathscr{P}|} \sum_{e \in \mathscr{P}} \frac{\vec{e}}{\|\vec{e}\|} - \beta \frac{1}{|\mathscr{U}|} \sum_{e \in \mathscr{U}} \frac{\vec{e}}{\|\vec{e}\|}$;
$\vec{c^-} \leftarrow \alpha \frac{1}{|\mathscr{U}|} \sum_{e \in \mathscr{U}} \frac{\vec{e}}{\|\vec{e}\|} - \beta \frac{1}{|\mathscr{P}|} \sum_{e \in \mathscr{P}} \frac{\vec{e}}{\|\vec{e}\|}$;
foreach $e \in \mathscr{U}$ **do**
  if $sim(\vec{c^+}, \vec{e}) \leq sim(\vec{c^-}, \vec{e})$ **then**
    $\mathscr{N} \leftarrow \mathscr{N} \cup e$;
  end
end

Figura 5.9: Selección a través de Rocchio de las instancias negativas de $\mathscr{U}$ a $\mathscr{N}$.

cada vez más complicada, y se necesitan enfoques que no requieran que todos los datos estén etiquetados. Por ende, estudiamos las habilidades de un enfoque en dos pasos para el aprendizaje de clase única, concretamente, Roc-SVM [LL03], que ha sido aplicado con éxito a la categorización de texto [LL03].

Roc-SVM [LL03] está basado en la combinación del método *Rocchio* [Roc71] y de los SVM [Vap00]. La técnica utiliza Rocchio para seleccionar algunas instancias negativas (que pertenecen al conjunto de datos sin etiquetar) relevantes y, seguidamente, aplica SVM para generar iterativamente clasificadores de los que, eventualmente, escogerá uno.

Para el primer paso (véase la Figura 5.9), el método asume que todo el conjunto no etiquetado $\mathscr{U}$ se compone de instancias negativas y, después, utiliza al conjunto positivo y etiquetado $\mathscr{P}$ y al propio $\mathscr{U}$ como los datos de entrenamiento para generar el clasificador *Rocchio*. El modelo se emplea para predecir la clase de instancias pertenecientes a $\mathscr{U}$. Para la predicción, cada instancia de prueba $e \in \mathscr{U}$ se compara con el vector prototipo $e \in \mathscr{P}$ utilizando la similitud del coseno [MS83]. Las instancias clasificadas como negativas, se consideran datos negativos denotados por $\mathscr{N}$.

Para el segundo paso (véase la Figura 5.10), Roc-SVM entrena iterativamente varios SVMs [Vap00] y, después, selecciona un clasificador entre las

```
// Asignar al conjunto 𝒩 la clase negativa, y al
   conjunto 𝒫 la clase positiva
foreach e ∈ 𝒩 do
    e ← c⃗⁻;
end
foreach e ∈ 𝒫 do
    e ← c⃗⁺;
end
repeat
    // Utilizar 𝒫 y 𝒩 para entrenar un clasificador SVM
       S_i
    S_i ← Entrenar(𝒫 ∪ 𝒩);
    // Clasificar 𝒬 utilizando S_i. El conjunto de
       instancias en 𝒬 que se clasifican como negativas se
       denotan por 𝒲
    𝒬' ← Clasificar(𝒬);
    𝒲 ← {e ∈ 𝒬' : e = c⃗⁻};
    if |𝒲| ≠ 0 then
        𝒬 ← 𝒬 − 𝒲;
        𝒩 ← 𝒩 ∪ 𝒲;
    end
until |𝒲| = 0;
// Utilizar el último clasificador SVM S_final si >8% de las
   instancias positivas se clasifican como negativas o el
   primer clasificador SVM S_1 en otro caso, para
   clasificar 𝒫
if |{e ∈ 𝒬' : e = c⃗⁺}| > 8/100 |𝒬'| then
    Seleccionar S_final como el clasificador;
end
else
    Seleccionar S_1 como el clasificador;
end
```

Figura 5.10: Generando el clasificador mediante Roc-SVM.

opciones generadas. La generación se realiza con el conjunto de datos $\mathscr{P}$ y $\mathscr{N}$. $\mathscr{Q}$ es el conjunto de las instancias que aún no tienen etiquetas, de tal

modo que se cumple $\mathscr{Q} = \mathscr{U} - \mathscr{N}$.

5.3 Validación empírica

En esta sección se recoge la experimentación que hemos realizado para validar ambos enfoques: el método de detección de malware desconocido mediante aprendizaje supervisado y mediante aprendizaje de clase única.

5.3.1 Metodología general

Para validar ambos paradigmas, hemos utilizado un conjunto de datos formado por un subconjunto de datos de malware y uno de software benigno.

Nos hemos descargado varias muestras de la web *VxHeavens* [VX10]. Esta web tiene un total de cerca de 17.000 programas maliciosos, incluyendo 585 familias de malware que representan los diferentes tipos de malware: caballos de Troya, virus, gusanos, etc. A pesar de que éstos estaban ya etiquetados con su nombre de familia, hemos utilizado *Eset Antivirus* [Ese10] para comprobar la nomenclatura. Hemos utilizado también PEiD [PeI10] para determinar si los ejecutables maliciosos estaban empaquetados. Hemos eliminado de la muestra los ejecutables empaquetados (que nuestro método de representación estático no podría modelar debidamente) y hemos seleccionado aleatoriamente un total de 1.000 ejecutables maliciosos. Para el conjunto de datos benigno, hemos recogido 1.000 ejecutables de nuestros propios equipos y los hemos analizado utilizando *Eset Antivirus* [Ese10].

Utilizando este conjunto de datos, queremos responder a las siguientes preguntas de investigación:

> «*¿Cuál es el tamaño óptimo de secuencia para la detección de malware desconocido utilizando aprendizaje supervisado?*»
>
> «*¿Cuáles son los mejores clasificadores supervisados para la detección de malware desconocido?*»
>
> «*¿Qué clase es mejor etiquetar para la detección de malware desconocido mediante aprendizaje de clase única?*»
>
> «*¿Cuál es el tamaño óptimo de instancias positivas (etiquetadas) para el aprendizaje de clase única?*»

Las dos primeras preguntas se refieren al enfoque completamente supervisado, en el que todos los ejecutables están etiquetados con su categoría (benigno o malware) y se describen en la sub-sección 5.3.2. Las otras dos preguntas se resuelven mediante los experimentos relativos al enfoque de clase única, descrito en la sub-sección 5.3.3.

Para medir la eficacia de los métodos obtenemos la tasa de positivos verdaderos (*True Positive Ratio*, TPR), que es el número de ejecutables maliciosos correctamente detectados, divididos por el número total de ficheros malware (véase la ecuación 5.26):

$$TPR = \frac{TP}{TP + FN} \tag{5.26}$$

donde TP es el número de casos de malware correctamente clasificados (positivos verdaderos o *True Positives*) y FN es el número de casos de malware incorrectamente clasificados como ficheros benignos. Medimos, además, la tasa de falsos positivos (*False Positive Ratio*, FPR), en otras palabras, el número de ejecutables benignos incorrectamente clasificados como malware (véase la ecuación 5.27):

$$FPR = \frac{FP}{FP + TN} \tag{5.27}$$

donde FP es el número de casos de software benigno incorrectamente clasificados como malware y TN es el número de ejecutables benignos correctamente clasificados. Además, medimos la precisión (Prec.), que es el número total de aciertos del clasificador dividido entre el total de ejecutables en el conjunto de datos (véase la ecuación 5.28):

$$Prec.(\%) = \frac{TP + TN}{TP + FP + TN + FN} \cdot 100 \tag{5.28}$$

Además, medimos *f-measure* una métrica típica de IR, que se calcula como sigue:

$$F\text{-}measure(\%) = \frac{2 \cdot TP}{2 \cdot TP + FN + FP} \tag{5.29}$$

Con estas métricas validaremos los paradigmas de detección de malware completamente supervisado, así como el enfoque de aprendizaje de clase única.

5.3.2 Aprendizaje supervisado para la detección de malware desconocido

Hemos extraído la representación de frecuencias de secuencias de *opcodes* para todo ejecutable en el conjunto de datos para diferentes tamaños de secuencia n. En concreto, hemos extraído las representaciones para $n = 1$ y $n = 2$. No hemos extraído representaciones con un tamaño superior debido a la complejidad subyacente de los métodos de aprendizaje automático supervisado y la gran cantidad de características obtenidas que haría el proceso de aprendizaje muy lento. Además, ya hemos comprobado en el capítulo 4 que no hay una mejora por utilizar longitudes de secuencia mayores de 2. No obstante, sí hemos combinado las dos representaciones de $n = 1$ y $n = 2$.

El total de características era muy alto para $n = 2$: 144.598 características. Para manejar esta circunstancia, hemos aplicado un paso de selección de características utilizando la ganancia de información (*Information Gain*, IG) [Ken83] para seleccionar las 1.000 con mayor relevancia. De este modo, hemos obtenido, finalmente, tres conjuntos de datos diferentes: el conjunto de datos con la representación de ejecutables con $n = 1$, de la que no hemos seleccionado características; el conjunto de datos con la representación de ejecutables con $n = 2$, al que se le ha aplicado una reducción de características; y el conjunto de datos con la representación combinada a partir de las representaciones con $n = 1$ y $n = 2$, del que también hemos seleccionado 1.000 características. Para conjunto de datos, hemos seguido los siguientes pasos metodológicos para su validación:

- ***Cross validation.*** Para evaluar el comportamiento de los clasificadores de aprendizaje automático, se suele emplear *k-fold cross validation* en la experimentación [Bis06]. De esta forma, con $k = 10$, cada conjunto de datos se divide 10 veces en 10 conjuntos diferentes de entrenamiento (90 % del total) y de prueba (10 % del total).

- **Aprendizaje del modelo.** Para cada paso de la validación, llevamos a cabo la fase de entrenamiento de cada algoritmo, aplicando diferentes parámetros o algoritmos de aprendizaje dependiendo del modelo concreto. Específicamente, hemos utilizado los cuatro modelos siguientes:

  - *Árboles de decisión (AD).* Hemos usado *random forest* [Bre01] y J48 (La implementación del algoritmo C4.5 [Qui93] realizada

por equipo de WEKA[Gar95]).

- *K-Nearest Neighbour (KNN)*. Hemos realizado experimentos en el rango desde $k = 1$ hasta $k = 10$ para entrenar a los KNNs.
- *Redes bayesianas (RB)*. Hemos usado diferentes algoritmos de aprendizaje estructural: K2 [CH91], *Hill Climbing* [RN03] y *Tree Augmented Naïve* (TAN) [GGP+97]. Además, hemos realizado experimentos con un clasificador *Naïve Bayes* [Lew98].
- *Support Vector Machines (SVM)*. Hemos empleado el algoritmo *Sequential Minimal Optimisation* (SMO) [Pla99] y hemos realizado experimentos con un *kernel polinomial* [AW99], un *kernel polinomial normalizado* [AW99], un *kernel Pearson VII* [UMB06] y un *kernel radial basis function* (RBF) [AW99].

- **Evaluar el modelo:** Para evaluar las capacidades de cada clasificador, hemos utilizado las métricas que hemos descrito con anterioridad: TPR, FPR, precisión y *f-measure*.

La Tabla 5.1 muestra los resultados para un tamaño de secuencia de 1. Utilizando este tamaño de secuencia, representamos únicamente la distribución de los *opcodes* en los ejecutables.

Los mejores resultados los obtuvo el SVM entrenado con el *kernel Pearson VII*, con una precisión del 92.02%. Casi todos los clasificadores que hemos probado, nos han dado resultados de precisión por encima del 90%, sugiriendo que las secuencias de opcodes son una representación acertada para la detección de malware desconocido. Las redes bayesianas entrenadas con K2, *hill climbing* y *Naïve Bayes* obtuvieron peores resultados que el resto de los clasificadores.

KNN K=2 tuvó los mejores resultados en TPR con un valor de 0,95. El resto de clasificadores obtuvieron resultados mayores del 85%, excepto, de nuevo, las redes bayesianas entrenadas con K2, *hill climbing* y *Naïve Bayes*; y el clasificador SVM con *kernel* RBF. En cuanto al FPR, la tasa más baja se obtuvo utilizando un SVM entrenado con el *kernel* Pearson VII: un 3%. Las redes bayesianas entrenadas con K2 y *hill climbing*, junto con el clasificador bayesiano *Naïve Bayes* y KNN K=2 obtuvieron un FPR mayor o igual a 0,10. Respecto al *f-measure*, SVM entrenado con el *kernel* Pearson VII y KNN k=1 fueron los clasificadores con mejores resultados. Por contra, las redes bayesianas entrenadas con K2 y *hill climbing*, y *Naïve Bayes* fueron los únicos clasificadores que obtuvieron valores menores que el 85%.

Tabla 5.1: Resultados de detección de malware desconocido para un tamaño de secuencia de 1.

Modelo	**Prec. (%)**	**TPR**	**FPR**	***F-measure***
KNN K=1	92,83	0,93	0,07	0,93
KNN K=2	90,75	0,95	0,14	0,91
KNN K=3	91,71	0,91	0,08	0,91
KNN K=4	91,40	0,93	0,10	0,92
KNN K=5	90,95	0,90	0,08	0,91
KNN K=6	90,87	0,91	0,10	0,91
KNN K=7	90,80	0,89	0,07	0,91
KNN K=8	90,68	0,9	0,09	0,90
KNN K=9	90,40	0,88	0,08	0,90
KNN K=10	90,36	0,9	0,09	0,90
AD: J48	91,25	0,92	0,09	0,92
AD: Random Forest N=10	91,43	0,92	0,09	0,92
SVM: RBF	81,67	0,67	0,03	0,79
SVM: Polinomial	89,65	0,88	0,09	0,89
SVM: Polinomial normalizado	91,67	0,88	0,05	0,91
SVM: Pearson VII	92,92	0,89	0,03	0,93
Naïve Bayes	79,64	0,90	0,31	0,81
RB: K2	84,24	0,79	0,10	0,84
RB: Hill Climbing	84,24	0,79	0,10	0,84
RB: TAN	90,65	0,87	0,05	0,91

Para este tamaño de secuencia de *opcodes*, los clasificadores se comportaron correctamente con precisiones mayores del 90% y tasas de falsos positivos menores del 10%, en su mayoría. En particular, el clasificador SVM utilizando el *kernel* Pearson VII sobresalió en esta configuración experimental, con los mejores resultados medios y superando al resto de clasificadores. Los clasificadores bayesianos, por el contrario, han sido los que peor resultados globales han obtenido para este tamaño de secuencia.

Cuando se utiliza un tamaño de secuencia $n = 2$ (véase la Tabla 5.2), los resultados mejoran sustancialmente.

Esta vez, los resultados de precisión del SVM con el *kernel polinomial normalizado* fueron los mejores, consiguiendo un 95,90% de precisión. Las redes bayesianas entrenadas con K2 y *hill climbing* fueron las peores — una precisión menor del 90%. En cuanto al TPR, *random forest*, SVM con *kernel polinomial* y KNN K=2 fueron los clasificadores que obtuvieron los mayo-

Tabla 5.2: Resultados de detección de malware desconocido para un tamaño de secuencia de 2.

Modelo	Prec. (%)	TPR	FPR	*F-measure*
KNN K=1	94,83	0,95	0,05	0,95
KNN K=2	93,15	0,96	0,10	0,93
KNN K=3	94,16	0,94	0,05	0,94
KNN K=4	93,89	0,95	0,07	0,94
KNN K=5	93,50	0,92	0,05	0,93
KNN K=6	93,38	0,93	0,06	0,93
KNN K=7	92,87	0,90	0,04	0,93
KNN K=8	92,89	0,91	0,05	0,93
KNN K=9	92,10	0,88	0,04	0,92
KNN K=10	92,24	0,90	0,05	0,92
AD: J48	92,61	0,93	0,08	0,93
AD: Random Forest N=10	95,26	0,96	0,06	0,95
SVM: RBF	91,93	0,89	0,05	0,92
SVM: Polinomial	95,50	0,96	0,05	0,96
SVM: Polinomial normalizado	95,90	0,94	0,02	0,96
SVM: Pearson VII	94,35	0,95	0,06	0,95
Naïve Bayes	90,02	0,90	0,10	0,90
RB:: K2	86,73	0,83	0,09	0,86
RB: Hill Climbing	86,73	0,83	0,09	0,86
RB: TAN	93,40	0,91	0,04	0,93

res resultados: 96% de detección. Con la excepción de las redes bayesianas entrenadas con K2 y *hill climbing*, los clasificadores consiguieron un TPR mayor del 85%. Para FPR, el SVM con el *kernel polinomial normalizado* superó al resto de clasificadores con un TPR de 0,2. Finalmente, todos los clasificadores, excepto las redes bayesianas entrenadas con K2 y *hill climbing*, obtuvieron un valor de *f-measure* mayor de 0,9.

En esta configuración del tamaño de secuencia de *opcodes*, los clasificadores se comportan mejor que para un tamaño de secuencia menor. Los clasificadores obtuvieron, por normal general, valores mayores del 90% de precisión, mayores del 85% de TPR, menores del 10% de FPR y mayores del 90% de *f-measure*. En particular, el clasificador SVM con *kernel polinomial normalizado* ha destacado sobre el resto de los clasificadores con una precisión del 95,9%, TPR del 94%, FPR de tan sólo 2% y un *f-measure* del 93%. Por contra, los modelos bayesianos entrenados con K2 y *hill climbing*

han sido los peores para esta configuración, con valores, algo mejores que para $n = 1$, pero que distan mucho del resto de clasificadores.

Tabla 5.3: Resultados de detección de malware desconocido para una combinación de secuencias de *opcodes* de tamaño 1 y 2. Los mejores resultados globales los da el SVM entrenado con el *kernel polinomial normalizado*.

Modelo	**Prec. (%)**	**TPR**	**FPR**	***F-measure***
KNN K=1	94,73	0,95	0,05	0,95
KNN K=2	93,06	0,96	0,10	0,93
KNN K=3	94,30	0,94	0,05	0,94
KNN K=4	94,13	0,95	0,07	0,94
KNN K=5	93,98	0,93	0,05	0,94
KNN K=6	94,09	0,94	0,06	0,94
KNN K=7	93,80	0,92	0,04	0,94
KNN K=8	93,80	0,93	0,06	0,93
KNN K=9	93,16	0,91	0,04	0,93
KNN K=10	93,20	0,92	0,05	0,93
AD: J48	92,34	0,93	0,09	0,92
AD: Random Forest N=10	94,98	0,96	0,06	0,95
SVM: RBF	91,07	0,89	0,05	0,92
SVM: Polinomial	95,50	0,96	0,05	0,96
SVM: Polinomial normalizado	95,80	0,94	0,02	0,96
SVM: Pearson VII Kernel	94,29	0,95	0,06	0,95
Naïve Bayes	89,81	0,90	0,11	0,90
RB: K2	87,29	0,83	0,08	0,87
RB: Hill Climbing	87,29	0,83	0,08	0,87
RB: TAN	93,40	0,91	0,04	0,93

Sorprendentemente, los resultados de la representación combinada con longitudes de secuencia de 1 y 2 (véase la Tabla 5.3), no consigue mejores resultados que para $n = 2$.

Específicamente, los resultados para esta configuración son ligeramente inferiores que para un tamaño de secuencia de 2, con la excepción de los modelos bayesianos. En términos de precisión el SVM con el *kernel polinomial normalizado* consiguió un 98 %, un 0,10 % menos que para el resultado previo utilizando secuencias de tamaño 2. Los resultados para TPR fueron los mismos que para $n = 2$, donde *random forest*, SVM con *kernel polinomial* y KNN con $K = 2$ obtuvieron 0,96 de TPR. Lo mismo ocurrió para el FPR, donde el SVM entrenado con el *kernel polinomial normalizado* fue el mejor

con un resultado del 2 %. Finalmente, los resultados para *f-measure* fueron prácticamente los mismos que en la configuración $n = 2$.

Los resultados arrojan una conclusión: la representación introducida en el capítulo 3 es válida para la detección de malware desconocido utilizando algoritmos de aprendizaje automático supervisado. En particular, los modelos bayesianos no parecen aptos para su adopción debido principalmente al gran número de falsos positivos en el que incurren. No obstante, el resto de modelos son adecuados para detectar malware desconocido, especialmente el SVM con *kernel polinomial normalizado*, que ha obtenido los mejores resultados del experimento.

A pesar de los buenos resultados de los modelos supervisados, éstos requieren un proceso de etiquetado que muchas veces puede resultar laborioso. Además, hemos de considerar que el conjunto de datos que hemos utilizado es muy pequeño en comparación con las ingentes cantidades de malware que procesan en una casa comercial anti-malware.

En el siguiente apartado estudiamos el modelo Roc-SVM [LL03] que no requiere tanta supervisión, con la intención de obtener, no tan buenos resultados como en los modelos supervisados, pero sí lo suficientemente adecuados como para ofrecer una alternativa para el periodo en que los que no se tienen todos los datos etiquetados.

5.3.3 Aprendizaje de clase única para la detección de malware desconocido

Para este experimento, hemos extraído la representación de secuencias de *opcodes* para todo fichero del conjunto de datos. Como configuración de la representación, hemos elegido un tamaño de secuencia de 2, debido a que esta configuración es la que mejores resultados consiguió en el anterior experimento. En este experimento, también hemos reducido el número de características de un total de 144.598 a las 1.000 con mayor ganancia de información [Ken83].

Hemos realizado dos experimentos diferentes, en el primero, hemos seleccionado la clase malware como la clase a etiquetar (la clase positiva) mientras que en el segundo hemos elegido el software benigno como la clase positiva. Para ambos experimentos, hemos dividido el conjunto de datos en 10 partes de entrenamiento y prueba, respectivamente, utilizando *cross-validation* [Bis06].

Seguidamente, hemos modificado el número de instancias etiquetadas

en la clase positiva a 100, 200, 300, 400, 500, 600, 700, 800 y 900; para, así, medir los efectos que produce el total de instancias etiquetadas sobre los resultados del clasificador. En total, hemos realizado nueve pruebas (de diferentes tamaños del conjunto de datos etiquetado) de Roc-SVM para cada clase positiva posible (malware y software benigno), para cada uno de los 10 conjuntos de datos resultantes de la división por *cross-validation*. Del mismo modo que en la validación anterior, hemos medido el comportamiento de Roc-SVM en términos de precisión, TPR, FPR y *f-measure*.

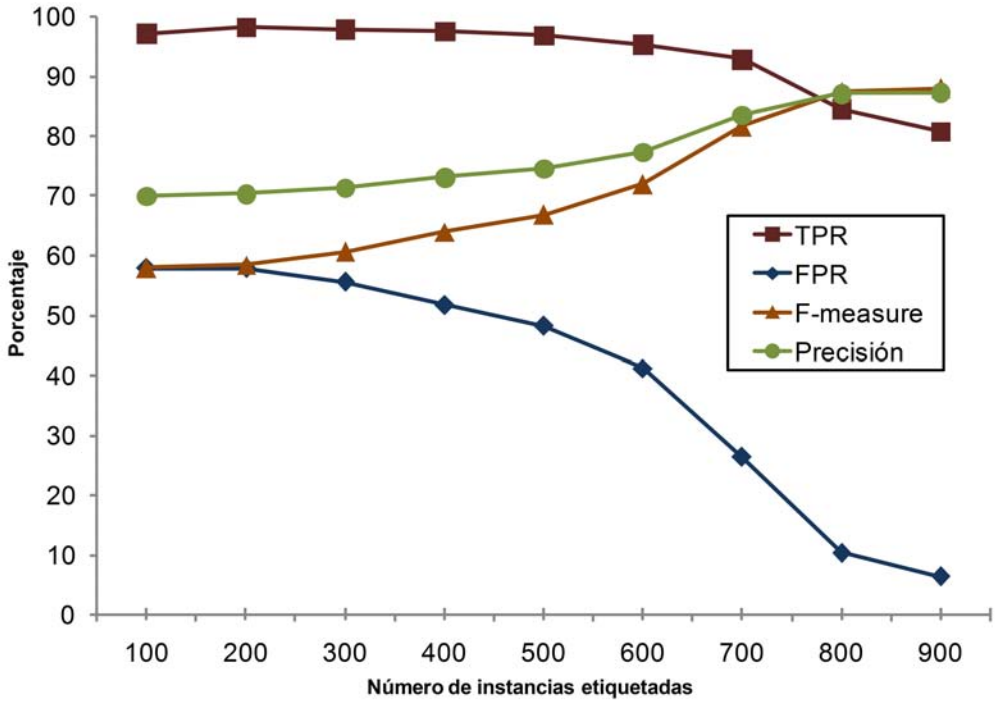

Figura 5.11: Los resultados de detección de malware desconocido cuando etiquetamos la clase compuesta de ejecutables maliciosos.

La Figura 5.11 muestra los resultados cuando el malware es la clase seleccionada para el etiquetado. De este modo, podemos apreciar como los resultados generales mejoran cuantos más ejecutables maliciosos se etiquetan. En concreto, en lo que respecta al TPR, la relación es indirecta: cuanto más malware se etiqueta, menos TPR se obtiene, es decir, menos malware se detecta. Por contra, al aumentar el tamaño del conjunto de malware etiquetado, también bajan los falsos positivos, por tanto, aumenta la confianza de Roc-SVM sobre lo que es verdaderamente malware. Tanto la *f-measure* como la precisión aumentan a la vez que el tamaño de conjunto de datos

etiquetado. Roc-SVM puede garantizar una precisión del 83,43% cuando se etiquetan 600 ejecutables maliciosos, un 60% del conjunto de datos de malware y un 30% del total.

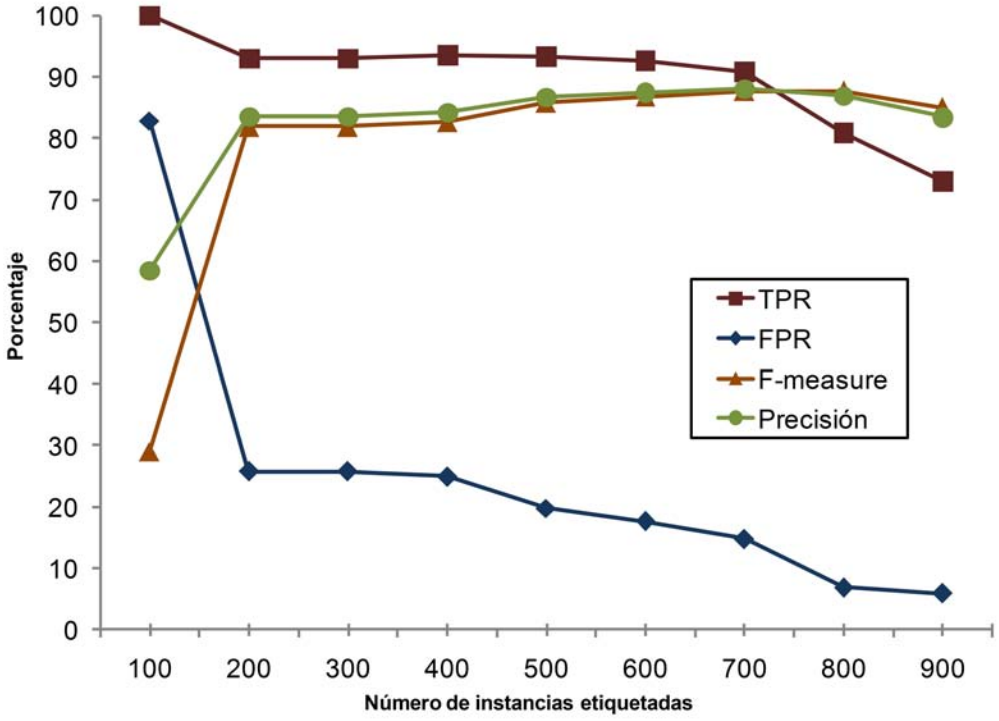

Figura 5.12: Los resultados de detección de malware desconocido cuando etiquetamos la clase compuesta por ejecutables benignos.

La Figura 5.12 muestra los resultados cuando el software benigno se etiqueta. En este caso, también mejoran los resultados generales de Roc-SVM al aumentar el número de ejecutables etiquetados. Sin embargo, esta vez sólo mejoran hasta un tamaño de conjunto de software benigno etiquetado de 600. Una vez superado este umbral, el clasificador comienza a comportarse peor. Concluimos de este resultado, que para Roc-SVM demasiado software benigno es redundante. Las tendencias generales de Roc-SVM son muy similares a las obtenidas cuando era el malware la clase a etiquetar: el TPR disminuye, según incrementa el tamaño de software benigno etiquetado, los falsos positivos son menores a más ejecutables benignos y, tanto precisión como *f-measure* incrementan según aumenta el tamaño de ejecutables etiquetados. En concreto, Roc-SVM consigue una precisión del 84,22% con tan sólo 400 ejecutables benignos etiquetados, es decir, etiquetar el 40% del software benigno y el 20% del total de ejecutables. Para 600 ejecuta-

bles benignos etiquetados, los resultados son los mejores del experimento: 87,46 %.

Los resultados muestran que es mejor enfocar los esfuerzos de etiquetado en la clase del software benigno que en la de malware si sólo vamos a etiquetar un número pequeño de ejecutables. Si es posible etiquetar una gran cantidad de malware, entonces el clasificador puede ser mejorado. Como conclusión, este enfoque de aprendizaje de clase única tiene unos resultados, que si bien no son tan altos como los obtenidos con un enfoque supervisado, requieren una labor de etiquetado mucho menor. En concreto, hemos demostrado que teniendo conocimiento de que el 30 % del conjunto de datos para el entrenamiento de un sistema de detección de malware desconocido es software benigno, podemos garantizar que va a clasificar correctamente un 86 % del software. Para garantizar una tasa de falsos positivos menor del 10 % tendríamos que incrementar el conjunto de datos etiquetados al 90 % de una clase, es decir, un 45 % del software total.

5.4 Discusión de los resultados

Los resultados validan nuestra hipótesis inicial de que es posible construir un detector de malware desconocido basado en una representación de frecuencias de aparición de *opcodes*.

Nuestro trabajo con los enfoques supervisados es original en cuanto a la representación utilizada, pero los mismos métodos que hemos utilizado para el aprendizaje, han sido empleados anteriormente con otro tipo de características. En particular, el trabajo de Schultz et ál. [SEZS01] fue el primero en utilizar aprendizaje automático para la detección de malware desconocido en el 2001. En ese trabajo utilizaba frecuencias de secuencias no solapadas de bytes para su representación. Kolter et ál. [KM04a] mejoraron esa representación utilizando secuencias que sí se solapaban (n-gramas). Nuestro enfoque es diferente porque utiliza secuencias de *opcodes* en vez de elementos tan fácilmente modificables como los bytes.

Sin embargo, los enfoques supervisados tienen el problema de que requieren una enorme cantidad de datos para el entrenamiento de los clasificadores. En nuestros experimentos, hemos utilizado un conjunto de datos, que pese a parecer relativamente grande (2.000 ejecutables), es muy pequeño si lo comparamos con el enorme volumen de las bases de datos de malware. Si el conjunto de datos crece, también lo va a hacer la dificultad de hacer el sistema escalable. Este problema con la dimensionalidad produce

unos altos requerimientos de almacenamiento, incrementa la complejidad temporal y empeora el rendimiento de los clasificadores [CHL06].

Para reducir estos costes desproporcionados de almacenamiento y tiempo, se necesita reducir el conjunto de entrenamiento [CJ06]. Para este fin, se utiliza la *reducción de datos* (*data reduction*) [Pyl99, TYW03]. Estas técnicas tienen ventajas como: reducir la medición, almacenamiento y transmisión; disminuir los tiempos de entrenamiento y de prueba; enfrentarse a la *maldición de la dimensionalidad* para mejorar la eficiencia, eficacia y simplicidad; y facilitar la visualización y la comprensión de los datos [Tor03, DL03].

La reducción de datos se puede realizar en dos sentidos. La *reducción de instancias* (*instance reduction*) (IS) que busca reducir el total de evidencias del conjunto de datos seleccionando las más relevantes o creando instancias artificiales [LM01]. La *reducción de características*, por su parte, reduce el numero de atributos del conjunto de datos [LM08]. En nuestro experimento hemos aplicado un paso de reducción de características para seleccionar las 1.000 secuencias de *opcodes* más relevantes.

A pesar de estas soluciones, el etiquetado es una necesidad real para los enfoques supervisados. Por tanto, hemos introducido el que es el primer enfoque de aprendizaje de clase única para la detección de malware desconocido. No existe, hasta donde alcanza nuestro conocimiento, un enfoque previo que se haya enfrentado a esta problemática de esta manera. Nuestro enfoque de aprendizaje de clase única ha demostrado que, a pesar de no ser tan eficaz como el enfoque supervisado, no requiere más del 30 % del etiquetado total para garantizar precisiones por encima del 85 %.

Por ende, consideramos que el uso del aprendizaje de clase única para la detección de malware desconocido es una contribución, ya no sólo al área académico-científica, sino a la industria anti-malware. Ésta puede verse altamente beneficiada de su utilización, reduciendo los esfuerzos de análisis para el etiquetado. Sin embargo, las herramientas de detección de malware desconocido, por su propia definición, son un tipo de detector *genérico* de malware. Existen ciertos dominios de aplicación que requieren de una mayor especificación, ya no sólo del sistema de detección de malware desconocido sino también de la que ofrece el sistema de detección de variantes. Un ejemplo claro es la banca electrónica, donde los ataques dirigidos han aumentado para convertirse en una verdadera preocupación [Pem05]. En este contexto, un enfoque como el nuestro, a pesar de ser capaz de identificar amenazas a la seguridad, no es capaz de proveer de la información sobre cómo esa amenaza está afectando a un banco o a un usuario concreto. Por ende, consideramos que, para estas situaciones, nuestros paradigmas,

se podrían utilizar como un filtro que decida qué muestras necesitan de un análisis más exhaustivo y, probablemente, manual. De este modo, se podría reducir el tiempo de procesamiento de enfoques más completos y costosos.

5.5 Sumario

En este capítulo hemos hecho dos contribuciones a la detección de malware desconocido: aprendizaje supervisado basado en la representación de frecuencias de secuencias de *opcodes* y aprendizaje de clase única basado en la misma representación.

En primer lugar, hemos utilizado la representación introducida en el capítulo 3 para el entrenamiento de varios algoritmos de aprendizaje automático supervisado. Hemos evaluado este enfoque con un conjunto de datos compuesto de 1.000 ejecutables maliciosos y de 1.000 ejecutables benignos. Los resultados que ofrece este enfoque se encuentran por encima del 90 % de precisión, con una tasa de falsos positivos inferior al 5 %, para la mayoría de clasificadores.

Sin embargo, debido a la gran cantidad de esfuerzo que supone el disponer de un conjunto de datos etiquetado, también hemos realizado, el que es, hasta la fecha, el primer enfoque de aprendizaje de clase única para la detección de malware. En este enfoque los datos no necesitan etiquetas más que en una de las clases (malware o software benigno). Hemos evaluado este enfoque demostrando que etiquetando un 90 % del software benigno, es decir, un 45 % del software total, podemos asegurar tasas de falsos positivos menores del 5 %. Si sólo nos interesa la precisión podemos garantizar resultados mayores del 87 % con un 60 % del software legítimo: un 30 % del total de ejecutables.

Creemos que este último enfoque puede suponer un cambio importante en la utilización del aprendizaje automático para la detección de malware desconocido. De este modo, reducimos los esfuerzos de etiquetar software, manteniendo niveles de precisión aceptables.

«Hay dos maneras de difundir la luz: ser la lámpara que la emite, o el espejo que la refleja.»

Lin Yutang (1895–1976)

Capítulo

6

Conclusiones

A través de este documento se han descrito y evaluado las contribuciones de la presente tesis doctoral. Ahora, es momento de evaluar el marco de estas contribuciones y el grado de consecución de las mismas, tanto en relación a la hipótesis inicial como a los objetivos de la tesis doctoral.

Lo que resta de capítulo está organizado como sigue. La sección 6.1 sintetiza las diferentes validaciones que se han llevado a cabo en esta investigación. La sección 6.2 resume los resultados y evalúa las contribuciones de esta tesis al área de la detección de malware. La sección 6.3 debate aplicaciones de esta tesis a un entorno real. La sección 6.4 detalla las limitaciones de las contribuciones de la tesis doctoral. La sección 6.5 orienta el trabajo futuro en esta línea. Finalmente, la sección 6.6 concluye esta disertación doctoral y describe nuestras opiniones personales acerca del futuro de la detección de malware.

6.1 Síntesis de la validación del sistema

Durante esta investigación doctoral hemos llevado a cabo varias validaciones empíricas de los diferentes componentes de la investigación global. En esta sección sintetizamos las conclusiones más importantes de éstas.

Para validar la representación de ejecutables basada en recuperación de la información, utilizamos un conjunto tanto de software malicioso como

software benigno y hemos extraído su representación en el esquema propuesto para diferentes valores de configuración del tamaño de secuencia. Hemos extraído igualmente funciones resumen o de *hash* para tener una línea de base para comparar nuestra solución en términos de velocidad. El objetivo de este experimento era el de comprobar si nuestro método era rápido o no. Por eso, lo hemos comparado con funciones de *hash* que son los métodos más rápidos para representar ejecutables.

Nuestro método fue especialmente rápido: únicamente entre 2,5 y 15 veces más lento, dependiendo en el tamaño de secuencia elegido. No obstante, el tamaño de secuencia elegido influye muy negativamente en la sobrecarga temporal de nuestro método. Por tanto, de cara a utilizar nuestro método de representación, siempre que los resultados de detección no disminuyesen, es preferible utilizar tamaños de secuencia de 2 o menos.

En lo referente a la detección de variantes, validamos nuestro sistema de una manera progresiva. En un primer lugar y para establecer un umbral de similitud a partir del cual establecer como variante de malware a un ejecutable, utilizamos varias familias de malware y comparamos cada uno de sus miembros entre sí para ver cuáles eran los valores de similitud entre éstas. De igual modo, se comparó cada miembro de cada familia con un conjunto de software legítimo para establecer el umbral sin que éste lanzase falsos positivos.

Una vez establecidos un rango de posibles umbrales, hemos realizado una prueba de detección propiamente dicha en la que se eligieron un grupo de muestras de cada familia al azar para que actuasen como variantes conocidas y el resto de las variantes se utilizaron como ejecutables a analizar. De igual modo, se emplearon 1.000 ejecutables benignos para medir la tasa de falsos positivos del sistema. Nuestro modelo de detección consiguió tasas tanto de falsos negativos como de falsos positivos menores del 15%, demostrando que nuestro sistema es capaz de detectar variantes de malware y establecer correctamente su familia.

Además, para validar el generador de modelos filogenéticos se construyeron los modelos de varias familias y se compararon sus resultados con el etiquetado habitual de las casas antivirus. Estos modelos aparentaron ser una muy buena opción para llevar un seguimiento de las tendencias de las familias de malware, así como para establecer un estándar de etiquetado de malware.

Por último, para validar la detección de malware desconocido se utilizó un conjunto de 1.000 ejecutables benignos y 1.000 maliciosos para entrenar diferentes modelos de aprendizaje. En un primer lugar, se utilizaron

modelos de aprendizaje supervisado que requieren que todos los datos de conjunto de entrenamiento estén debidamente etiquetados con el tipo de software (en nuestro caso, benigno o malware). Utilizando *cross-validation* de 10, que consiste en realizar 10 veces la división del 90% del conjunto de datos para entrenar y el 10% para probar, de tal modo que se prueben todas las instancias, se obtuvieron precisiones por encima del 95%.

Sin embargo, teniendo en cuenta que la cantidad de malware se está incrementando, realizamos también pruebas tratando de reducir el número requerido de ejecutables etiquetados. Con este fin, utilizamos Roc-SVM un modelo de aprendizaje de clase única para comparar sus resultados con los de los modelos de aprendizaje supervisado. Para ello, hicimos dos experimentos diferentes. En un primero, etiquetaríamos sólo ejecutables maliciosos mientras que en un segundo software benigno. Además, etiquetamos un número variable de la clase seleccionada para ver hasta dónde podíamos reducir el etiquetado. Hicimos todas las pruebas utilizando *cross-validation* para reducir la variabilidad de los resultados. Obtuvimos que podemos obtener una precisión del 86% etiquetando únicamente el 50% del software benigno. Es decir, que utilizando un 25% del esfuerzo de etiquetado que se realizaba para el aprendizaje supervisado, la precisión se reduce en tan sólo un 10%.

6.2 Resumen de los resultados

Esta sección resume las contribuciones de la tesis doctoral. En primer lugar, es momento de revisitar la hipótesis fundamental de la tesis doctoral presentada:

> «*Existe un método de representación rápido para ejecutables que es la base de un detector de malware capaz, tanto de identificar variantes de familias de malware conocidas como de detectar malware desconocido.*»

Los diferentes estudios empíricos que hemos realizado y que se encuentran resumidos en la sección 6.1 validan nuestra hipótesis de partida. La utilización del modelo de representación de ejecutables basado en frecuencias de secuencias de *opcodes*, que actúa como la $\mathscr{F}$ del modelo de IR propuesto por Baeza-Yates et ál. [BR99], ha resultado en las siguientes contribuciones al área de detección de malware:

1. **Un modelo de representación de ejecutables basado en técnicas de recuperación de la información.** El modelo de representación propuesto está basado en las frecuencias de aparición de las secuencias de *opcodes*, ponderando la secuencia en función de cuán relevantes son los *opcodes* que la componen. Hemos evaluado el modelo de representación en términos de recursos computacionales. Para ello, hemos comparado nuestro método con dos líneas de base: las funciones de resumen criptográfico MD5 y SHA1. Esta validación ha demostrado que nuestro enfoque, a pesar de, obviamente, ser más lento que las funciones resumen, se mantiene en unos márgenes de tiempo (en el peor caso 15 veces más lento y en el mejor caso 2,5 veces) que lo hacen viable para su adopción por la industria anti-malware.

2. **Una arquitectura escalable para la detección de variantes basada en recuperación de la información.** Hemos propuesto un enfoque basado en recuperación de la información para la detección de variantes, que está compuesto de: (i) una base de datos de malware, (ii) un marco de trabajo para representar ejecutables, (iii) un método para transformar el ejecutable a inspeccionar en una consulta a la base datos y (iv) una medida de similitud entre la consulta y los ejecutables almacenados en la base de datos. El enfoque es escalable, pudiéndose modificar la representación sin cambiar el entorno global.

3. **Un método para la detección de variantes de malware.** El enfoque compara una representación en forma vectorial de los ejecutables para proveer de un medida de similitud y comparar los ejecutables entre sí. La validación ha demostrado que el enfoque produce métricas de similitud que hacen posible establecer un rango para la detección de variantes ofuscadas de malware sin sacrificar el porcentaje de falsos positivos. En concreto, nuestro método es capaz de ofrecer una cobertura de un 45 % de toda una familia de malware, con una única representación para la familia. Además, hemos realizado una prueba de detección que muestra que nuestro modelo no obtiene tasas de falsos negativos ni de falsos positivos mayores del 15 %, clasificando específicamente la familia a la pertenece un ejecutable malicioso.

4. **Una técnica para organizar las variantes de las familias de malware y seguir su evolución real.** Basándonos en la representación de los ejecutables, hemos desarrollado un método que crea modelos filogenéticos de las familias de malware. Este método utiliza *clustering* o

agrupamiento jerárquico para monitorizar la evolución de una familia de malware.

5. **Un método supervisado para la detección de malware desconocido.** Hemos propuesto un enfoque de aprendizaje automático supervisado utilizando la representación de ejecutables en forma de frecuencias de secuencias de *opcodes*. Los resultados muestran que podemos obtener unas precisiones por encima del 95 % en la detección de malware desconocido utilizando este enfoque.

6. **El primer sistema basado en aprendizaje de clase única para la detección de malware desconocido.** A pesar de la habilidad de los enfoques supervisados, hemos adaptado un modelo de aprendizaje de clase única para la detección de malware desconocido. El modelo RocSVM es capaz de ofrecer precisiones por encima del 85 %, etiquetando únicamente un 30 % del total de ejecutables que serían necesarios para un aprendizaje supervisado.

Mediante el logro de estas contribuciones hemos sido capaces de cumplir con todos los objetivos específicos planteados en el capítulo 1:

- Desarrollar un método de representación rápido que actúe como marco de trabajo para una detección de malware completa.
- Desarrollar y evaluar un detector de malware capaz de detectar nuevas variantes de malware actual.
- Desarrollar y evaluar un detector de malware capaz de detectar malware desconocido.

Asimismo, hemos realizado ciertas contribuciones a fin de alcanzar los objetivos operacionales de la tesis:

- Diseñar e implementar un método para la representación de ejecutables, así como validarlo en términos de velocidad.
- Diseñar e implementar un marco de trabajo completo y escalable para la detección de variantes de malware.
- Desarrollar un método para seguir la evolución de las variantes de malware y organizarlas.

- Minimizar el número de ejecutables benignos incorrectamente clasificados como malware en la detección de variantes malware, proveyendo de un método para configurar lo estricto del detector.
- Desarrollar y validar un método para detectar malware desconocido.
- Desarrollar un método alternativo para la detección de malware desconocido que no minimice el esfuerzo previo.
- Minimizar el número de ejecutables benignos incorrectamente clasificados como malware en la detección de malware desconocido.

Con la consecución de todos los objetivos de la tesis, consideramos como logrado el objetivo principal de la tesis que no era sino: «Desarrollar un método de representación rápido que actúe como el marco de trabajo para una detección de malware completa» y, por ende, la hipótesis inicial de la tesis. A partir de un representación relativamente rápida, hemos sido capaces de construir un detector de malware completo, capaz de detectar tanto variantes de familias de malware como malware desconocido, minimizando además el esfuerzo en este último caso.

6.3 Aplicaciones de la investigación

Obviamente, la aplicación de las contribuciones de esta tesis doctoral han de estar situadas dentro de la industria del software de seguridad centrada en el análisis de malware.

En este contexto, el modelo de representación de ejecutables basado en recuperación de la información puede ser otro modo de almacenar la base de datos de malware de las casas comerciales de antivirus. La representación basada en frecuencias de secuencias de *opcodes* ha demostrado que es lo suficientemente rápida. Por tanto, es un método para generar perfiles de malware automáticamente y añadirlos con celeridad a la base de datos de malware. Además, la arquitectura propuesta para la detección de variantes es altamente escalable: podemos cambiar la representación de los ejecutables (que en nuestro caso son secuencias de *opcodes*) por otras representaciones como p. ej., llamadas al sistema, cadenas de bytes, o información de análisis de flujo de control, sin que este proceso de adaptación requiera demasiado esfuerzo. Siempre que la información pueda ser representada como un vector, es susceptible de ser modificada o de ser añadida al sistema de detección de variantes.

El detector de variantes ha sido capaz de proveer de una cobertura del 45% de una familia de malware con una única especificación. Hemos de tener en cuenta que, para nuestros experimentos, hemos utilizado algunas familias con un total de variantes mayor de 50 y hemos comparado un ejecutable con el resto de variantes de la misma.

Evidentemente, esta situación es la peor imaginable para una casa antivirus, ya que cuanto más evoluciona una familia (y por tanto, mayor número de variantes tiene) menos similitud debería existir entre los miembros iniciales y los finales de la misma. Un entorno en el que se haya adoptado el detector de variantes propuesto, sería capaz de detectar las primeras variantes de la forma original, añadiéndolas a la base de datos de malware y, por consiguiente, detectando las nuevas variantes generadas.

Con el generador de modelos filogenéticos se puede tanto optimizar el almacenamiento de las mismas, reduciendo el número de variantes almacenadas en la base de datos, como estandarizar el etiquetado de variantes de malware.

En lo que respecta a la detección de malware desconocido, tanto el método supervisado como el método de aprendizaje de clase única fueron capaces de detectar más del 90% de los ejecutables maliciosos, con una tasa de falsos positivos inferior al 5%. La metodología de aprendizaje supervisado es muy precisa (más del 95% de precisión) pero también más costosa, necesitando que todo el conjunto de datos de entrenamiento esté etiquetado.

Por contra, el aprendizaje de clase única es una excelente técnica en la que sólo se necesitaba que el 60% del software benigno estuviese etiquetado para garantizar un precisión por encima del 85%. Consideramos que estos últimos resultados tienen un gran impacto para la industria de software de seguridad, especialmente en la reducción de costes operativos y mejora del rendimiento de los sistemas anti-malware.

Consideramos que, por todo lo comentado, nuestras contribuciones tienen un alto grado de aplicabilidad en la industria, pudiendo ayudar a reducir costes y a mejorar la precisión de sus soluciones software.

Además, una parte del trabajo de esta tesis doctoral se ha realizado con la financiación del Ministerio de Industria CDTI [Min10] bajo el proyecto *Segur@: Seguridad y Confianza en la Sociedad de la Información*, que es el mayor proyecto de seguridad hasta la fecha del estado español. Así, hemos tenido la oportunidad de colaborar con dos grandes empresas de seguridad como son *Panda Security* [Sec10], pionera de la seguridad de la información en España; y *Optenet*, una de las líderes mundiales en filtrado de conteni-

dos; contando con dos magnificas vías para la aplicación de los resultados de esta investigación a entornos reales de la industria de seguridad internacional.

6.4 Limitaciones

La principal limitación de esta tesis doctoral, es que no es capaz de enfrentarse a malware *empaquetado*. Debido a su naturaleza estática, no le es posible asegurar que el código analizado, en este caso *opcodes*, sea realmente el utilizado por el malware. La mejor solución para resolver este inconveniente es utilizar una herramienta de desempaquetado dinámico como paso previo, como *PolyUnpack* [RHD+06], *Renovo* [KPY07], *OmniUnpack* [MCJ07] o *Eureka* [SYS+08]. Estos métodos son capaces de obtener una visión real del código. Sin embargo, introducen un importante incremento en el coste computacional del sistema global.

Por otro lado, tal y como comentamos en el capítulo 5, nuestro detector de malware es *genérico*. En ciertas circunstancias, es necesaria una mayor especificación en el análisis del malware. En concreto, el incremento en el número de troyanos bancarios que roban credenciales a fin de lucrarse ilícitamente, hace que un análisis exhaustivo del malware sea necesario, sobre todo por parte de las entidades bancarias [Pem05]. Dentro de este contexto, nuestro enfoque no es suficiente porque, como máximo, clasifica malware en familias. La necesidad de estas entidades es diferente que la de una casa comercial de antivirus: necesitan saber cómo el malware les afecta a ellos y a sus clientes. No obstante, nuestro enfoque podría ser utilizado como filtro para sistemas de análisis más exhaustivos y costosos, a fin de reducir tiempo de cómputo.

6.5 Trabajo futuro

Hemos identificado tres líneas de investigación para el trabajo futuro en torno a esta investigación doctoral. En esta sección se describen las mismas.

6.5.1 Mejora en la representación de ejecutables

Nuestro método de representación de ejecutables ha arrojado unos resultados muy correctos tanto en celeridad para la obtención de la representación, como en la precisión de los métodos que hemos desarrollado utilizán-

dola como base. No obstante, creemos que esta representación es mejorable en ciertos aspectos.

En primer lugar, existe un concepto en recuperación de la información textual, al que se le conoce como *stop word removal* [MS83, BR99, SR03] (en castellano, borrado de palabras no relevantes) que consiste en eliminar palabras carentes de significado, p. ej. preposiciones, determinantes o conjunciones. Debido a que nuestro enfoque está inspirado en recuperación de información, un enfoque similar puede mejorar tanto la detección de variantes como la detección de malware desconocido. En particular, sugerimos nuestra medida de relevancia de *opcodes* individuales como un punto de partida para establecer listas de *stop opcodes* (*opcodes* de parada en castellano). A partir de ésta, podemos establecer un rango de relevancia de *opcodes* por debajo del cual, eliminemos los *opcodes* de la representación del ejecutable.

En segundo lugar, nuestra representación se obtiene de una manera estática y, por tanto, no se puede asegurar que el código desensamblado sea el que se vaya a ejecutar. Una línea de mejora para la representación sería añadir características dinámicas en la representación. Éstas pueden ser, por ejemplo, llamadas al sistema peligrosas obtenidas en un entorno controlado de ejecución.

Por último, creemos que nuestra representación también puede ser mejorada utilizando recursos adicionales para la creación de modelos de representación más complejos que el VSM [Sal89, MS83]. En la categorización textual, el uso de bases de datos léxicas puede mejorar la recuperación de información [Gd97, GdU+02, Gó04]. A pesar de que el código máquina es un lenguaje artificial y, por tanto, más estructurado y ordenado, también podemos considerar que existen ciertas relaciones léxicas en sus componentes.

Por ejemplo, un conjunto de secuencias de instrucciones en código máquina puede ser equivalente a otro, es decir, las secuencias pueden ser *sinónimas* una de la otra. Además, este tipo de relaciones puede explotarse por los creadores de malware para ofuscar una amenaza conocida de los detectores de malware. De esta forma, una dirección de trabajo futuro en la representación de ejecutables, es la de integrar todo este conocimiento experto acerca de similitudes en secuencias de instrucciones en bases de datos léxicas de lenguaje máquina.

Mediante estas ampliaciones y mejoras del sistema de representación de ejecutables, consideramos que la representación que hemos introducido en esta tesis doctoral se vería fuertemente reforzada. Por tanto, el sistema de

detección se vería, a su vez, mejorado, pudiendo afrontar mejor amenazas nuevas y más complejas.

6.5.2 Ampliación del sistema de detección

El sistema de detección completo que hemos presentado, compuesto a su vez de métodos de detección de variantes y malware desconocido, es el primer enfoque que unifica los conceptos de detección de variantes y de detección de malware desconocido.

Primeramente, consideramos que es necesaria una mejora en el número de ejecutables almacenados en la base de datos de malware. Vamos a utilizar los modelos filogenéticos como la base sobre la que apoyar un sistema de reducción del total de variantes almacenadas en la base de datos.

En segundo lugar, vamos a desarrollar un sistema para reconocer los patrones temporales que, creemos, se repiten en la propagación de malware. *Topic Detection and Tracking* (TDT) [All02, WY10] (en castellano, detección y seguimiento de temas) es una técnica de procesado de lenguaje natural que es capaz de identificar un tema en el lenguaje y de monitorizarlo en el tiempo. Queremos adaptar TDT a la detección y monitorización de comportamientos del malware, añadiendo una dimensión temporal a los modelos filogenéticos que obtenga una información más completa sobre el comportamiento y las conductas de propagación de los ejecutables maliciosos.

Finalmente, vamos a integrar los métodos presentados en esta disertación con técnicas más habituales, para crear un motor anti-malware completo. Este sistema estará al menos compuesto por nuestros modelos de detección de malware, métodos clásicos de firmas y un entorno dinámico para la monitorización de ejecutables.

6.5.3 Defender los medios actuales de propagación de malware

Además de detectar malware una vez que éste ha sido liberado, es necesario proteger los medios de propagación de los ejecutables maliciosos.

Históricamente, uno de los medios más utilizados para propagación de malware ha sido y sigue siendo el correo electrónico [WSW09]. En concreto, el *spam*, que se define como correo comercial no solicitado, es uno de los medios más habituales para la propagación de malware [BFC+06]. Los filtros bayesianos son los más utilizados para la detección de spam [APK+00,

Sch03b, AKC+00, AKCS00, See07]. Sin embargo, no distinguen entre categorías de los mensajes clasificados como *spam*. En particular, nuestra intención es categorizar los mensajes de correo *spam* en distintas categorías (v. gr., comerciales, *phising*[1] o malware) a fin de centrarnos en la propagación de malware a partir de los buzones de correo electrónico.

Las redes sociales son unos de los servicios más utilizado dentro de Internet, con un número de usuarios en constante crecimiento [Fac10]. Como tal, están predispuestas a ser utilizadas por los creadores de malware a fin de maximizar sus infecciones. Por ejemplo, *Koobface* [CNe10] es un ejecutable malicioso reciente que, tras una infección recoge información de los usuarios (p. ej., números de tarjeta de crédito) en redes sociales.

Este tipo de ataques está aumentando de forma exponencial [MMMK09, LLLF09]. Por todo esto, iniciaremos una línea de investigación para proteger a las redes sociales de estas amenazas, adaptando las soluciones a las necesidades de este entorno de distribución e infección de malware.

6.6 Consideraciones finales

El futuro de la detección de malware es un asunto espinoso y delicado que, bajo nuestra opinión, no se solucionará nunca. En esta disertación hemos presentado una investigación que, si bien realiza importantes contribuciones al área de la detección de malware, no soluciona el problema general relativo al malware. Cualquier tipo de solución que se plantee para detectar malware puede ser superada con nuevas técnicas en creación de malware. De hecho, es directamente imposible garantizar un detección perfecta del mismo [Coh87]. Por tanto, hemos de ser realistas y pensar en soluciones dirigidas no a solucionar este problema sino a mitigar sus efectos perniciosos.

Es curioso que, pese a existir malware en redes móviles [FLJ+07], éste no haya llegado todavía a la inasumible cantidad que está presente en los equipos personales. Para nosotros, la razón subyacente de esta aparente diferencia en el malware, es que una aplicación móvil casi siempre pasa por un proceso de certificación, que garantiza su legitimidad. Por ende, consideramos que la certificación de software es un paso necesario y obligatorio para, al menos, reducir el número de amenazas. Por supuesto, seguirán existiendo amenazas y vulnerabilidades que, desde luego, los creadores de malware explotarán, pero el poner una barrera de entrada que les obligue

[1] El *phishing* busca engañar a un usuario haciéndose pasar por un remitente de confianza para conseguir las credenciales del usuario.

a falsificar certificaciones creemos que reducirá notablemente el total de amenazas.

En torno a estas infraestructuras, los detectores de malware, como nuestro enfoque basado en recuperación de la información, tendrían más tiempo para adaptarse a la siempre cambiante realidad. Minimizando la cantidad de malware, se podría además dedicar más tiempo a la modificación de los paradigmas de detección y estudiar sus limitaciones y, por otro lado, se podrían analizar exhaustivamente las nuevas amenazas capaces de superar los nuevos sistemas de detección. Al fin y al cabo,

> *«Conoce a tu enemigo y conócete a ti mismo; en cien batallas, nunca saldrás derrotado.»*
>
> *Sun Tzu – 孙子 (544 a. C.–481 a. C.).*
> *El Arte de la Guerra.*

Bibliografía

[ACKS04a] T. Abou Assaleh, N. Cercone, V. Keselj, y R. Sweidan. Detection of new malicious code using n-grams signatures. En *Proceedings of Second Annual Conference on Privacy, Security and Trust*, páginas 193–196. 2004.

[ACKS04b] Tony Abou Assaleh, Nick Cercone, Vlado Keselj, y Ray Sweidan. N-gram-based detection of new malicious code. En *Proceedings of COMPSAC Workshops*, páginas 41–42. 2004.

[AKC+00] I. Androutsopoulos, J. Koutsias, K. V. Chandrinos, G. Paliouras, y C. D. Spyropoulos. An evaluation of naive bayesian anti-spam filtering. En *Proceedings of the workshop on Machine Learning in the New Information Age*, páginas 9–17. 2000.

[AKCS00] I. Androutsopoulos, J. Koutsias, K. V. Chandrinos, y C. D. Spyropoulos. An experimental comparison of naive Bayesian and keyword-based anti-spam filtering with personal e-mail messages. En *Proceedings of the 23rd annual international ACM SIGIR conference on Research and development in information retrieval*, páginas 160–167. 2000.

[All70] F.E. Allen. Control flow analysis. *ACM Sigplan Notices*, 5(7):1–19, 1970.

[All02] J. Allan. *Topic detection and tracking: event-based information organization*. 2002.

[AN08] I. Aktug y K. Naliuka. ConSpec–a formal language for policy specification. *Electronic Notes in Theoretical Computer Science*, 197(1):45–58, 2008.

[APK+00] I. Androutsopoulos, G. Paliouras, V. Karkaletsis, G. Sakkis, C. D. Spyropoulos, y P. Stamatopoulos. Learning to filter spam e-mail: A comparison of a naive bayesian and a memory-based approach. En *Proceedings of the Machine Learning and Textual Information Access Workshop of the 4th European Conference on Principles and Practice of Knowledge Discovery in Databases*. 2000.

[AVMD02] G. Antoniol, U. Villano, E. Merlo, y M. Di Penta. Analyzing cloning evolution in the linux kernel. *Information and Software Technology*, 44(13):755–765, 2002.

[AW99] S. Amari y S. Wu. Improving support vector machine classifiers by modifying kernel functions. *Neural Networks*, 12(6):783–789, 1999.

[Bay63] T. Bayes. An essay towards solving a problem in the doctrine of chances. *Philosophical Transactions of the Royal Society*, 53:370–418, 1763.

[BC07] M. W. Berry y M. Castellanos. *Survey of text mining II: clustering, classification, and retrieval*. Springer-Verlag New York Inc, 2007.

[BD92] H. J. Bandelt y A. W. M. Dress. A canonical decomposition theory for metrics on a finite set. *Advances in mathematics*, 92(1):47–105, 1992.

[BDD+01] J. Bergeron, M. Debbabi, J. Desharnais, M. M. Erhioui, Y. Lavoie, y N. Tawbi. Static detection of malicious code in executable programs. *International Journal of Requirements Engineering*, páginas 184–189, 2001.

[BDEK99] J. Bergeron, M. Debbabi, M. M. Erhioui, y B. Ktari. Static Analysis of Binary Code to Isolate Malicious Behaviors. En *Proceedings of the 8th Workshop on Enabling Technologies on Infrastructure for Collaborative Enterprises*, páginas 184–189. IEEE Computer Society Washington, DC, USA, 1999.

[BEd08] O. Banerjee, L. El Ghaoui, y A. dÁspremont. Model selection through sparse maximum likelihood estimation for multiva-

riate Gaussian or binary data. *The Journal of Machine Learning Research*, 9:485–516, 2008.

[Ber89] B. L. Berg. *Qualitative research methods for the social sciences*. Allyn and Bacon Boston, 1989.

[Ber03] M. W. Berry. *Survey of text mining: clustering, classification, and retrieval*. Springer-Verlag New York Inc, 2003.

[BFC+06] A. Bratko, B. Filipič, G. V. Cormack, T. R. Lynam, y B. Zupan. Spam filtering using statistical data compression models. *The Journal of Machine Learning Research*, 7:2673–2698, 2006.

[BHB+09] U. Bayer, I. Habibi, D. Balzarotti, E. Kirda, y C. Kruegel. A view on current malware behaviors. En *Proceedings. of the Usenix Workshop on Large-scale Exploits and Emergent Threats (LEET)*. 2009.

[Bil07a] D. Bilar. Fingerprinting malicious code through statistical opcode analysis. En *Proceedings of the 3rd International Conference on Global E-Security (ICGeS)*. 2007.

[Bil07b] D. Bilar. Opcodes as predictor for malware. *International Journal of Electronic Security and Digital Forensics*, 1(2):156–168, 2007.

[Bis06] C. M. Bishop. *Pattern recognition and machine learning*. Springer New York., 2006.

[BKK06] U. Bayer, C. Kruegel, y E. Kirda. TTAnalyze: A tool for analyzing malware. En *Proceedings of the 15th Annual Conference of the European Institute for Computer Antivirus Research (EICAR)*. 2006.

[BM08] J. M. Borello y L. Mé. Code obfuscation techniques for metamorphic viruses. *Journal in Computer Virology*, 4(3):211–220, 2008.

[BMKK06] U. Bayer, A. Moser, C. Kruegel, y E. Kirda. Dynamic analysis of malicious code. *Journal in Computer Virology*, 2(1):67–77, 2006.

[BMM06] D. Bruschi, L. Martignoni, y M. Monga. Detecting self-mutating malware using control-flow graph matching. *Lecture Notes in Computer Science*, 4064:129, 2006.

[Bon04] V. Bontchev. Anti-virus spamming and the virus-naming mess: Part 2. *Virus Bulletin*, páginas 13–15, 2004.

[BR99] Ricardo A. Baeza-Yates y Berthier Ribeiro-Neto. *Modern Information Retrieval*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1999.

[BRB+09] B. D. Birrer, R .A. Raines, R. O. Baldwin, M. E. Oxley, y S. K. Rogers. Using Qualia and Hierarchical Models in Malware Detection. *Journal of Information Assurance and Security*, 4:247–255, 2009.

[Bre01] L. Breiman. Random forests. *Machine learning*, 45(1):5–32, 2001.

[Bur09] S. Burji. *Reverse Engineering of a Malware eyeing the Future of Security*. Tesis Doctoral, The University of Akron, 2009.

[CDKT09] K. Coogan, S. Debray, T. Kaochar, y G. Townsend. Automatic Static Unpacking of Malware Binaries. En *Proceedings of the 16th Working Conference on Reverse Engineering*, páginas 167–176. IEEE Computer Society, 2009.

[CGH96] Enrique Castillo, José M. Gutiérrez, y Ali S. Hadi. *Expert Systems and Probabilistic Network Models*. Springer, New York, NY, USA, 1996. ISBN 0387948589.

[CH91] Gregory F. Cooper y Edward Herskovits. A bayesian method for constructing bayesian belief networks from databases. En *Proceedings of the 7th conference on Uncertainty in artificial intelligence*. 1991.

[Che76] P. P. S. Chen. The entity-relationship model—toward a unified view of data. *ACM Transactions on Database Systems (TODS)*, 1(1):9–36, 1976.

[CHL06] J.R. Cano, F. Herrera, y M. Lozano. On the combination of evolutionary algorithms and stratified strategies for training

set selection in data mining. *Applied Soft Computing Journal*, 6(3):323–332, 2006.

[Cho08] S. N. Chong. *Expressive and enforceable information security policies*. Tesis Doctoral, Cornell University, 2008.

[Chr07] M. Christodorescu. *Behavior-based malware detection*. Tesis Doctoral, University of Wisconsin-Madison, 2007.

[CJ03] Mihai Christodorescu y Somesh Jha. Static analysis of executables to detect malicious patterns. En *Proceedings of the 12th USENIX Security Symposium*, páginas 169–186. 2003.

[CJ04] Mihai Christodorescu y Somesh Jha. Testing malware detectors. *ACM SIGSOFT Software Engineering Notes*, 29(4):34–44, 2004.

[CJ06] I. Czarnowski y P. Jedrzejowicz. Instance reduction approach to machine learning and multi-database mining. En *Proceedings of the Scientific Session organized during XXI Fall Meeting of the Polish Information Processing Society, Informatica, ANNALES Universitatis Mariae Curie-Skłodowska, Lublin*, páginas 60–71. 2006.

[CJK07] M. Christodorescu, S. Jha, y C. Kruegel. Mining specifications of malicious behavior. En *Proceedings of the the 6th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering*, páginas 5–14. ACM, 2007.

[CJS+05] M. Christodorescu, S. Jha, S. A. Seshia, D. Song, y R. E. Bryant. Semantics-aware malware detection. En *Proceedings of the 2005 IEEE Symposium on Security and Privacy*, páginas 32–46. 2005.

[CKJ+05] M. Christodorescu, J. Kinder, S. Jha, S. Katzenbeisser, y H. Veith. Malware normalization. Informe técnico, University of Wisconsin, 2005.

[CL03] J. Connor-Linton. Chi square tutorial, 2003. Disponible en: `http://www.georgetown.edu/faculty/ballc/webtools/web_chi_tut.html`.

[CL06a] M. R. Chouchane y A. Lakhotia. Using engine signature to detect metamorphic malware. En *Proceedings of the 4th ACM workshop on Recurring malcode*, páginas 73–78. ACM, 2006.

[CL06b] M. R. Chouchane y A. Lakhotia. Using engine signature to detect metamorphic malware. En *Proceedings of the 4th ACM workshop on Recurring malcode*, páginas 73–78. 2006.

[CNe10] CNet. Koobface virus hits facebook, 2010. Disponible en: `http://news.cnet.com/koobface-virus-hits-facebook/`.

[Coh84] Fred Cohen. *Experiments with Computer Viruses*. 1984.

[Coh86] Q. Cohen. *Computer viruses*. Tesis Doctoral, 1986.

[Coh87] F. Cohen. Computer viruses. *Computers & security*, 6(1):22–35, 1987.

[Coh94] F. Cohen. *A short course on computer viruses*. John Wiley & Sons, Inc. New York, NY, USA, 1994.

[Coh95] W. W. Cohen. Fast effective rule induction. En *Proceedings of the 12th International Conference on Machine Learning*, páginas 115–123. 1995.

[CP89] S. Crocker y M. Pozzo. A Proposal for a Verification-Based Virus Filter. En *Proceedings of the 1989 IEEE Symposium on Security and Privacy.*, páginas 319–324. 1989.

[CSZ06] O. Chapelle, B. Schölkopf, y A. Zien. *Semi-supervised learning*. MIT Press, 2006.

[CT98] C. Collberg y C. Thomborson. Breaking abstractions and unstructural data structures. En *Proceedings of the IEEE International Conference on Computer Languages (ICCL '98)*, páginas 28–37. 1998.

[CTG05] D. M. Cai, J. Theiler, y M. Gokhale. Detecting a malicious executable without prior knowledge of its patterns. En *Proceedings of the Defense and Security Symposium. Information Assurance, and Data Network Security*, páginas 1–12. 2005.

[CTL97] C. Collberg, C. Thomborson, y D. Low. A taxonomy of obfuscating transformations. Informe técnico, Department of Computer Science, The University of Auckland, New Zealand, 1997.

[CTL98] C. Collberg, C. Thomborson, y D. Low. Manufacturing cheap, resilient, and stealthy opaque constructs. En *Proceedings of the 25th ACM Sigplan-Sigact symposium on Principles of programming languages*, páginas 184–196. 1998.

[CW00] D.M. Chess y S.R. White. An undetectable computer virus. En *Proceedings of the Virus Bulletin Conference (VB)*. 2000.

[Dal07] M. Dalla Preda. *Code Obfuscation and Malware Detection by Abstract Interpretation*. Tesis Doctoral, Universit'a degli Studi di Verona, 2007.

[Dat10a] Rescue Data. IDA Pro Disassembler - multi-processor, windows hosted disassembler and debugger, 2010. Disponible en: `http://www.hex-rays.com/idapro/`.

[Dat10b] Data Rescue. Universal pe unpacker plug-in, 2010. Disponible en: `http://www.datarescue.com/idabase/unpack_pe`.

[DG05] M. Dalla Preda y R. Giacobazzi. Semantics-based code obfuscation by abstract interpretation. En *Proceedings of the 32nd International Colloquium on Automata, Languages andProgramming* (*ICALP '05*), tomo 3580 de *Lecture Notes in Computer Science*, páginas 1325–1336. 2005.

[DGK^{+}00] M. Debbabi, E. Giasson, B. Ktari, F. Michaud, y N. Tawbi. Secure self-certified cots. En *Proceedings of the 9th IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises*, páginas 183–188. 2000.

[DGP08] L. M. De Campos, J. A. Gámez Martín, y J. M. Puerta. Learning bayesian networks by ant colony optimisation: searching in two different spaces. *Mathware & Soft Computing*, 9(3):251–268, 2008.

[DL03] M. Dash y H. Liu. Consistency-based search in feature selection. *Artificial Intelligence*, 151(1-2):155–176, 2003.

[DMJRM00] R. De Maesschalck, D. Jouan-Rimbaud, y D. L. Massart. The mahalanobis distance. *Chemometrics and Intelligent Laboratory Systems*, 50(1):1–18, 2000.

[DP08] D. Dolz y G. Parra. Using exception handling to build opaque predicates in intermediate code obfuscation techniques. *Journal of Computer Science & Technology*, 8(2):71–77, 2008.

[DRSL08] A. Dinaburg, P. Royal, M. Sharif, y W. Lee. Ether: Malware analysis via hardware virtualization extensions. En *Proceedings of the 15th ACM conference on Computer and communications security*, páginas 51–62. ACM New York, NY, USA, 2008.

[DSC+10] J. Devesa, I. Santos, X. Cantero, Y. K. Penya, y P. G. Bringas. Automatic Behaviour-based Analysis and Classification System for Malware Detection. En *Proceedings of the 12th International Conference on Enterprise Information Systems (ICEIS)*. 2010.

[DWS+03] P. S. Deng, J. H. Wang, W. G. Shieh, C. P. Yen, y C. T. Tung. Intelligent automatic malicious code signatures extraction. En *Proceedings of the 37th IEEE Annual International Carnahan Conference on Security Technology,*, páginas 600–603. 2003.

[Ese10] Eset-Software. Eset - Antivirus Software with Spyware and Malware Protection, 2010. Disponible en: `http://www.eset.com/`.

[Fac10] Facebook. Facebook statistics, 2010. Disponible en: `http://http://www.facebook.com/press/info.php?statistics`.

[FB92] W. B. Frakes y R. Baeza-Yates. *Information retrieval: data structures and algorithms*. Prentice-Hall, Inc. Upper Saddle River, NJ, USA, 1992.

[Fel85] J. Felsenstein. Confidence limits on phylogenies: an approach using the bootstrap. *Evolution*, 39(4):783–791, 1985.

[Fer92] D. Ferbrache. *A pathology of computer viruses*. Springer-Verlag New York, Inc. Secaucus, NJ, USA, 1992.

[FH52] E. Fix y J. L. Hodges. Discriminatory analysis: Nonparametric discrimination: Small sample performance. *Technical Report Project 21-49-004, Report Number 11*, 1952.

[Fil07] É. Filiol. Metamorphism, formal grammars and undecidable code mutation. *International Journal of Computer Science*, 2(1):70–75, 2007.

[FIP01] N. FIPS. 180-2: Secure hash standard (SHS). Informe técnico, National Institute of Standards and Technology (NIST), 2001. Disponible en: `http://csrc.nist.gov/publications/fips/fips180-2/fips180-2withchangenotice.pdf`.

[FLJ+07] C. Fleizach, M. Liljenstam, P. Johansson, G. M. Voelker, y A. Mehes. Can you infect me now?: malware propagation in mobile phone networks. En *Proceedings of the 2007 ACM workshop on Recurring malcode*, página 68. ACM, 2007.

[Gó04] J. M. Gómez-Hidalgo. *Integración de Bases de Datos Léxicas y Colecciones de Entrenamiento en la categorización Automática de Documentos*. Tesis Doctoral, Universidad Complutense de Madrid, 2004.

[Gar95] S. R. Garner. Weka: The Waikato environment for knowledge analysis. En *Proceedings of the New Zealand Computer Science Research Students Conference*, páginas 57–64. 1995.

[Gd97] J. M. Gómez-Hidalgo y M. de Buenaga. Integrating a lexical database and a training collection for text categorization. En *Proceedings of the ACL/EACL Workshop on Automatic Information Extraction and Building of Lexical Semantic Resources*. 1997.

[GdU+02] J.M. Gómez-Hidalgo, M. de Buenaga, L. Ureña, M. Martín, y M. García. Integrating lexical knowledge in learning-based text categorization. En *Proceedings of the 6^{th} International Conference on the Statistical Analysis of Textual Data (JADT)*. 2002.

[Ger72] David Gerrold. *When Harlie Was One*. Doubleday, 1972.

[GFC08] F. Guo, P. Ferrie, y T. Chiueh. A study of the packer problem and its solutions. En *Proceedings of the 2008 Conference on Recent Advances in Intrusion Detection (RAID)*, páginas 98–115. 2008.

[GGP+97] Dan Geiger, Moises Goldszmidt, G. Provan, P. Langley, y P. Smyth. Bayesian network classifiers. En *Machine Learning*, páginas 131–163. 1997.

[GJSB05] J. Gosling, B. Joy, G. Steele, y G. Bracha. *Java (TM) Language Specification*. Addison-Wesley Professional, 2005.

[GL06] V. Gruhn y R. Laue. Complexity metrics for business process models. En *Proceedings of the 9th international conference on business information systems (BIS 2006)*, tomo 85, páginas 1–12. 2006.

[Gor94] S. Gordon. The generic virus writer. En *Proceedings of the International Virus Bulletin Conference*, páginas 121–138. 1994.

[Gor95] S. Gordon. Technologically enabled crime: Shifting paradigms for the year 2000. *Virus Bulletin*, 14(5):417, 1995.

[Gra10] GrammaTech. GrammaTech: Products: CodeSurfer, 2010. Disponible en: `http://www.grammatech.com/products/codesurfer/overview.html`.

[GSHC09] K. Griffin, S. Schneider, X. Hu, y T. Chiueh. Automatic Generation of String Signatures for Malware Detection. En *Proceedings of the 12th International Symposium in Recent Advances in Intrusion Detection (RAID 2009)*, páginas 101–120. 2009.

[GT01] M. Godfrey y Q. Tu. Growth, evolution, and structural change in open source software. En *Proceedings of the 4th International Workshop on Principles of Software Evolution*, páginas 103–106. ACM, 2001.

[GZ06] M. Graves y M. Zulkernine. Bridging the gap: software specification meets intrusion detector. En *Proceedings of the International Conference on Privacy, Security and Trust: Bridge the Gap Between PST Technologies and Business Services*, páginas 1–8. 2006.

[Hab73] S. J. Haberman. The analysis of residuals in cross-classified tables. *Biometrics*, 29(1):205–220, 1973.

[Hal77] M. H. Halstead. *Elements of software science*. Elsevier New York, 1977.

[HB05] G. Hoglund y J. Butler. *Rootkits: Subverting the Windows Kernel*. Addison-Wesley Professional, 2005.

[HF98] C. Herley y D. Florencio. Breaking abstractions and unstructural data structures. En *Proceedings of the 1998 IEEE International Conference on Computer Languages (ICCL '98)*, páginas 28–37. 1998.

[HF06] C. Herley y D. Florencio. How to login from an Internet cafe without worrying about keyloggers. En *Proceedings of the Symposium on Usable Privacy and Security (SOUPS)*. 2006.

[HFS98] S. A. Hofmeyr, S. Forrest, y A. Somayaji. Intrusion detection using sequences of system calls. *Journal of Computer Security*, 6(3):151–180, 1998.

[HJ06] O. Henchiri y N. Japkowicz. A feature selection and evaluation scheme for computer virus detection. En *IEEE International Conference on Data Mining (ICDM)*, páginas 891–895. 2006.

[HLY92] L. Harn, H. Y. Lin, y S. Yang. A software authentication system for the prevention of computer viruses. En *Proceedings of the ACM Annual cCnference on Communications*, páginas 447–450. ACM, 1992.

[HVSC07] G. Hu, D. Venugopal, S. M. Syst, y O. H. Columbus. A malware signature extraction and detection method applied to mobile networks. En *Proceedings of the IEEE International Performance, Computing, and Communications Conference (IPCCC)*, páginas 19–26. 2007.

[IM07] N. Idika y A. P. Mathur. A survey of malware detection techniques. Informe técnico, Department of Computer Science, Purdue University, 2007.

[Inf04] National Center For Biotechnology Information. A basic introduction to the science underlying the ncbi resources. 2004. Disponible en: http://www.ncbi.nlm.nih.gov/About/primer/phylo.html.

[Ins10] Instruction Counter. A plugin for the Disassembler IDA Pro, 2010. Disponible en: http://www.the-interweb.com/serendipity/index.php?/archives/57-InstructionCounter-plugin-for-IDA-Pro.html.

[JDF08] G. Jacob, H. Debar, y E. Filiol. Behavioral detection of malware: from a survey towards an established taxonomy. *Journal in Computer Virology*, 4(3):251–266, 2008.

[Jol02] I. T. Jolliffe. *Principal component analysis*. Springer verlag, 2002.

[Jon08] C. Jones. *Applied software measurement*. McGraw-Hill Osborne Media, 2008.

[KA94] J. O. Kephart y W. C. Arnold. Automatic extraction of computer virus signatures. En *Proceedings of the 4th Virus Bulletin International Conference*, páginas 178–184. 1994.

[Kar03] G. Karypis. CLUTO: A clustering toolkit, release 2.1. 1. Informe técnico, University of Minnesota, Department of Computer Science, Minneapolis, MN, 2003.

[Kar10] Karsperky Labs. Kaspersky Security Bulletin: Statistics 2008, 2010. Disponible en: http://www.viruslist.com/en/analysis?pubid=204792052.

[KCK+09] C. Kolbitsch, P. M. Comparetti, C. Kruegel, E. Kirda, X. Zhou, y X.F. Wang. Effective and efficient malware detection at the end host. En *Proceedings of the 18th USENIX Security Symposium*. 2009.

[Ken83] J. T. Kent. Information gain and a general measure of correlation. *Biometrika*, 70(1):163–173, 1983.

[KKSV05] J. Kinder, S. Katzenbeisser, C. Schallhart, y H. Veith. Detecting malicious code by model checking. En *Proceedings of the*

2005 Conference on Detection of Intrusions and Malware & Vulnerability Assessment (DIMVA), páginas 174–187. 2005.

[Kle96] E. M. Kleinberg. An overtraining-resistant stochastic modeling method for pattern recognition. *The annals of statistics*, 24(6):2319–2349, 1996.

[KLT+01] R. Kapur, M. Lousberg, T. Taylor, B. Keller, P. Reuter, y D. Kay. CTL the language for describing core-based test. En *International Test Conference*, páginas 131–139. 2001.

[KM04a] J. Z. Kolter y M. A. Maloof. Learning to detect malicious executables in the wild. En *Proceedings of the 10th ACM SIGKDD international conference on Knowledge discovery and data mining*, páginas 470–478. ACM New York, NY, USA, 2004.

[KM04b] Jeremy Z. Kolter y Marcus A. Maloof. Learning to detect malicious executables in the wild. En *Proceedings of the 10th ACM SIGKDD international conference on Knowledge discovery and data mining (KDD)*, páginas 470–478. 2004.

[KM06] J. Z. Kolter y M. A. Maloof. Learning to detect and classify malicious executables in the wild. *The Journal of Machine Learning Research*, 7:2721–2744, 2006.

[KM07] K. Kendall y C. McMillan. Practical Malware Analysis. En *Black Hat Conference, USA*. 2007.

[Kot07] S. B. Kotsiantis. Supervised Machine Learning: A Review of Classification Techniques. En *Proceeding of the Conference on Emerging Artificial Intelligence Applications in Computer Engineering: Real Word AI Systems with Applications in eHealth, HCI, Information Retrieval and Pervasive Technologies*, páginas 3–24. 2007.

[KP04] S. Kotsiantis y P. E. Pintelas. Recent advances in clustering: A brief survey. *WSEAS Transactions on Information Science and Applications*, 1(1):73–81, 2004.

[KPY07] M.G. Kang, P. Poosankam, y H. Yin. Renovo: A hidden code extractor for packed executables. En *Proceedings of the 2007 ACM workshop on Recurring malcode*, páginas 46–53. 2007.

[KS92] S. Kumar y E. H. Spafford. A generic virus scanner in C++. En *Proceedings of the 8th Computer Security Applications Conference*, páginas 210–219. 1992.

[KSVZ07] N. Kuzurin, A. Shokurov, N. Varnovsky, y V. Zakharov. On the Concept of Software Obfuscation in Computer Security. *Lecture notes in computer science*, 4779:281–298, 2007.

[KWLP05a] M. E. Karim, A. Walenstein, A. Lakhotia, y L. Parida. Malware phylogeny generation using permutations of code. *Journal in Computer Virology*, 1(1):13–23, 2005.

[KWLP05b] M. E. Karim, A. Walenstein, A. Lakhotia, y L. Parida. Malware Phylogeny Using Maximal πPatterns. En *Proceedings of the 2005 EICAR 2005 Conference*, páginas 156–174. 2005.

[Lab05] Panda Labs. Pandalabs 2005 annual malware report, 2005. Disponible en: `http://www.pandasecurity.com/homeusers/media/press-releases/viewnews?noticia=7066`.

[LDL+03] B. Liu, Y. Dai, X. Li, W. S. Lee, y P. S. Yu. Building text classifiers using positive and unlabeled examples. En *Proceedings of the 3rd IEEE International Conference on Data Mining(ICDM)*, páginas 179–186. 2003.

[Lew98] D. D. Lewis. Naive (Bayes) at forty: The independence assumption in information retrieval. *Lecture Notes in Computer Science*, 1398:4–18, 1998.

[LH07] R. Lyda y J. Hamrock. Using entropy analysis to find encrypted and packed malware. *IEEE Security & Privacy*, 5(2):40–45, 2007.

[LKK04] A. Lakhotia, A. Kapoor, y E. U. Kumar. Are metamorphic viruses really invincible? *Virus Bulletin*, páginas 5–7, 2004.

[LL03] X. Li y B. Liu. Learning to classify texts using positive and unlabeled data. En *Proceedings of the International Joint Conference on Artificial Intelligence (IJCAI)*, tomo 18, páginas 587–594. 2003.

[LLLF09] Weimin Luo, Jingbo Liu, Jing Liu, y Chengyu Fan. An analysis of security in social networks. *Dependable, Autonomic and Secure Computing, IEEE International Symposium on*, 0:648–651, 2009. doi:http://doi.ieeecomputersociety.org/10.1109/DASC.2009.100.

[LLO95] R. W. Lo, K. N. Levitt, y R. A. Olsson. MCF: A malicious code filter. *Computers & Security*, 14(6):541–566, 1995.

[LM01] H. Liu y H. Motoda. *Instance selection and construction for data mining*. Kluwer Academic Pub, 2001.

[LM08] H. Liu y H. Motoda. *Computational methods of feature selection*. Chapman & Hall/CRC, 2008.

[LRS04] E. Lamma, F. Riguzzi, y S. Storari. Improving the K2 Algorithm Using Association Rule Parameters. En *Proceedings of the 2004 Information Processing and Management of Uncertainty in Knowledge-Based Systems (IPMU)*, páginas 1667–1674. 2004.

[LWM09] F. Leder, T. Werner, y P. Martini. Proactive Botnet Countermeasures–An Offensive Approach. En *Cooperative Cyber Defence Centre of Excellence Tallinn*. 2009.

[LWSH05] W. Li, K. Wang, S. Stolfo, y B. Herzog. Fileprints: Identifying file types by n-gram analysis. En *the Proceedings of the IEEE Workshop on Information Assurance and Security*. 2005.

[Mar94] J.J. Marciniak. *Encyclopedia of software engineering*. Wiley, 1994.

[Mas10] Y. Mashevsky. Watershed in malicious code evolution, 2010. Disponible en: `http://www.viruslist.com/en/analysis?pubid=167798878`.

[McA10] McAfee Avert Lab. Mcafee avert labs unveils predictions for top ten security threats in 2007 as hacking comes of age, 2010. Disponible en: `http://www.mcafee.com/us/about/press/corporate/2006/20061129_080000_f.html`.

[MCJ07] L. Martignoni, M. Christodorescu, y S. Jha. Omniunpack: Fast, generic, and safe unpacking of malware. En *Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC)*, páginas 431–441. 2007.

[ME08] R. Moskovitch y Y. Elovici. Unknown Malicious Code Detection–Practical Issues. En *Proceedings of the 7th European Conference on Information Warfare*, páginas 145–153. 2008.

[Mic09] Microsoft. Signing and checking code with authenticode. 2009. Disponible en: `http://msdn.microsoft.com/en-us/library/ms537364(VS.85).aspx`.

[Min10] Ministerio de Industria. Centro para el Desarrollo Tecnológico Industrial, 2010. Disponible en: `www.cdti.es/`,.

[MISI06] A. Mori, T. Izumida, T. Sawada, y T. Inoue. A tool for analyzing and detecting malicious mobile code. En *Proceedings of the 28th International Conference on Software Engineering*, páginas 831–834. ACM, 2006.

[MKK07a] A. Moser, C. Kruegel, y E. Kirda. Exploring multiple execution paths for malware analysis. En *Proceedings of the 28th IEEE Symposium on Security and Privacy*. 2007.

[MKK07b] A. Moser, C. Kruegel, y E. Kirda. Limits of static analysis for malware detection. En *Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC)*, páginas 421–430. 2007.

[MM00] Gary McGraw y Greg Morrisett. Attacking malicious code: A report to the infosec research council. *IEEE Software*, 17:33–41, 2000.

[MMJ05] M. Milenković, A. Milenković, y E. Jovanov. Using instruction block signatures to counter code injection attacks. *ACM SIGARCH Computer Architecture News*, 33(1):108–117, 2005.

[MMMK09] Z. Mazur, H. Mazur, y T. Mendyk-Krajewska. Security of Internet Transactions. *Internet-Technical Development and Applications*, páginas 243–251, 2009.

[Moo96] T. K. Moon. The expectation-maximization algorithm. *IEEE Signal processing magazine*, 13(6):47–60, 1996.

[Mor01] P. Morley. Processing virus collections. En *Proceedings of the 2001 Virus Bulletin Conference (VB1)*, páginas 129–134. 2001.

[MP05] W. Masri y A. Podgurski. Using dynamic information flow analysis to detect attacks against applications. En *Proceedings of the Workshop on Software Engineering for Secure Systems—Building Trustworthy Applications*, páginas 1–7. ACM, 2005.

[MS83] M. J. McGill y G. Salton. *Introduction to modern information retrieval*. McGraw-Hill, 1983.

[MS09] A. Mooijaart y A. Satorra. On Insensitivity of the Chi-Square Model Test to Nonlinear Misspecification in Structural Equation Models. *Psychometrika*, 74(3):443–455, 2009.

[MSF+08] R. Moskovitch, D. Stopel, C. Feher, N. Nissim, y Y. Elovici. Unknown malcode detection via text categorization and the imbalance problem. En *Proceedings of the 6^{th} IEEE International Conference on Intelligence and Security Informatics (ISI)*, páginas 156–161. 2008.

[NB66] J. Neumann y A. W. Burks. *Theory of self-reproducing automata*. Univ. of Illinois Press, 1966.

[New10] NewBasic. NewBasic - An x86 Assembler/Disassembler for DOS, 2010. Disponible en: `http://www.frontiernet.net/~fys/newbasic.htm`.

[OF79] P. C. O'Brien y T. R. Fleming. A multiple testing procedure for clinical trials. *Biometrics*, 35(3):549–556, 1979.

[OM90] E. Okamoto y H. Masumoto. ID-based authentication system for computer virus detection. *Electronics Letters*, 26:1169–1170, 1990.

[Orl67] L. Orloci. An agglomerative method for classification of plant communities. *The Journal of Ecology*, 55(1):193–206, 1967.

[Pan10] Panda Research. Malware formation statistics, 2010. Disponible en: http://research.pandasecurity.com/malwareformation-statistics.

[PCJD08] M. D. Preda, M. Christodorescu, S. Jha, y S. Debray. A semantics-based approach to malware detection. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 30(5):25, 2008.

[Pea82] Judea Pearl. Reverend bayes on inference engines: a distributed hierarchical approach. En *Proceedings of the National Conference on Artificial Intelligence*, páginas 133–136. 1982.

[PeI10] PeID, 2010. Disponible en: http://www.peid.info/.

[Pem05] M. Pemble. Evolutionary trends in bank customer-targeted malware. *Network Security*, 2005(10):4–7, 2005.

[Pen59] L. S. Penrose. Self-reproducing machines. *Scientific American*, 200(6):105–114, 1959.

[PF04] F. Perriot y P. Ferrie. Principles and practise of x-raying. En *Proceedings of the Virus Bulletin International Conference*, páginas 51–66. 2004.

[Pla99] J. Platt. Sequential minimal optimization: A fast algorithm for training support vector machines. *Advances in Kernel Methods-Support Vector Learning*, 208, 1999.

[PLD05] H. Peng, F. Long, y C. Ding. Feature selection based on mutual information: criteria of max-dependency, max-relevance, and min-redundancy. *IEEE Transactions on Pattern Analysis and Machine Intelligence*, páginas 1226–1238, 2005.

[PLL08] R. Perdisci, A. Lanzi, y W. Lee. McBoost: Boosting Scalability in Malware Collection and Analysis Using Statistical Classification of Executables. En *Proceedings of the 24th Annual Computer Security Applications Conference (ACSAC)*, páginas 301–310. 2008.

[Por09] P. Porras. Inside risks Reflections on Conficker. *Communications of the ACM*, 52(10):23–24, 2009.

[Pyl99] D. Pyle. *Data preparation for data mining*. Morgan Kaufmann, 1999.

[QEM10] QEMU. QEMU. Open Source Processor Emulator, 2010. Disponible en: `http://www.qemu.org`.

[Qui86] J. R. Quinlan. Induction of decision trees. *Machine learning*, 1(1):81–106, 1986.

[Qui93] J. R. Quinlan. *C4. 5 programs for machine learning*. Morgan Kaufmann Publishers, 1993.

[Rad92] Y. Radai. Checksumming techniques for anti-viral purposes. En *Proceedings of the IFIP 12th World Computer Congress on Education and Society-Information Processing'92-Volume 2-Volume 2*, páginas 511–517. North-Holland Publishing Co., 1992.

[RBR+07] N. Rodriguez Ezpeleta, H. Brinkmann, B. Roure, N. Lartillot, B. F. Lang, y H. Philippe. Detecting and overcoming systematic errors in genome-scale phylogenies. *Systematic biology*, 56(3):389, 2007.

[RHD+06] P. Royal, M. Halpin, D. Dagon, R. Edmonds, y W. Lee. Polyunpack: Automating the hidden-code extraction of unpack-executing malware. En *Proceedings of the 22nd Annual Computer Security Applications Conference (ACSAC)*, páginas 289–300. 2006.

[RHW+08] K. Rieck, T. Holz, C. Willems, P. Dussel, y P. Laskov. Learning and classification of malware behavior. *Lecture Notes in Computer Science*, 5137:108–125, 2008.

[Riv92] R. Rivest. RFC1321: The MD5 message-digest algorithm. Informe técnico, RFC Editor United States, 1992.

[RKLC03] J. C. Rabek, R. I. Khazan, S. M. Lewandowski, y R. K. Cunningham. Detection of injected, dynamically generated, and obfuscated malicious code. En *Proceedings of the ACM Workshop on Rapid malcode*, páginas 76–82. ACM New York, NY, USA, 2003.

[RN03] S. J. Russell y Norvig. *Artificial Intelligence: A Modern Approach*. Prentice Hall, 2003.

[Rob04] S. Robertson. Understanding inverse document frequency: on theoretical arguments for IDF. *Journal of Documentation*, 60(6):503–520, 2004.

[Roc71] J. J. Rocchio. Relevance feedback in information retrieval. *The SMART retrieval system: experiments in automatic document processing*, páginas 313–323, 1971.

[RW06] C. E. Rasmussen y C. K. I. Williams. *Gaussian processes for machine learning*. Springer, 2006.

[Sal89] G. Salton. *Automatic text processing: the transformation, analysis, and retrieval of information by computer*. 1989.

[SBN+10] Igor Santos, Felix Brezo, Javier Nieves, Yoseba Penya, Borja Sanz, Carlos Laorden, y Pablo Bringas. Idea: Opcode-sequence-based malware detection. En *Engineering Secure Software and Systems*, tomo 5965 de *LNCS*, páginas 35–43. 2010. 10.1007/978-3-642-11747-3_3.

[SBS99] R. Sekar, T. Bowen, y M. Segal. On preventing intrusions by process behavior monitoring. En *Proceedings of the Workshop on Intrusion Detection and Network Monitoring*, páginas 29–40. 1999.

[SCF06] N. Smith, A. Capiluppi, y J. Fernández-Ramil. Agent-based simulation of open source evolution. *Software Process Improvement and Practice*, 11(4):423–434, 2006.

[Sch03a] R. E. Schapire. The boosting approach to machine learning: An overview. *Lecture Notes in Statistics*, páginas 149–172, 2003.

[Sch03b] K. M. Schneider. A comparison of event models for Naive Bayes anti-spam e-mail filtering. En *Proceedings of the 10th Conference of the European Chapter of the Association for Computational Linguistics*, páginas 307–314. 2003.

[SCH08] N. Swamy, B. J. Corcoran, y M. Hicks. Fable: A language for enforcing user-defined security policies. En *Proceedings of the 29th IEEE Symposium on Security and Privacy*, páginas 369–383. 2008.

[Sec10] Panda Security, 2010. Disponible en: `http://www.pandasecurity.com`.

[See07] A. K. Seewald. An evaluation of naive Bayes variants in content-based learning for spam filtering. *Intelligent Data Analysis*, 11(5):497–524, 2007.

[SEZS01] M. G. Schultz, E. Eskin, F. Zadok, y S. J. Stolfo. Data mining methods for detection of new malicious executables. En *Proceedings of the 22th IEEE Symposium on Security and Privacy.*, páginas 38–49. 2001.

[SF01] P. Ször y P. Ferrie. Hunting for metamorphic. páginas 123–144. 2001.

[Sha51] C. E. Shannon. Prediction and Entropy of Printed English. *Bell Systems Technical Journal*, 30:50–64, 1951.

[Sha84] A. Shamir. Identity-based cryptosystems and signature schemes. En *Proceedings of CRYPTO*, páginas 47–53. Springer, 1984.

[Shi88] O. Shivers. Control flow analysis in scheme. *ACM SIGPLAN Notices*, 23(7):174, 1988.

[SJB04] S. E. Schechter, J. Jung, y A. W. Berger. Fast detection of scanning worm infections. *Lecture notes in computer science*, páginas 59–81, 2004.

[SKB08] V. S. Sathyanarayan, P. Kohli, y B. Bruhadeshwar. Signature Generation and Detection of Malware Families. *Lecture Notes in Computer Science*, 5107:336–349, 2008.

[SKF08] M. Z. Shafiq, S. A. Khayam, y M. Farooq. Embedded Malware Detection using Markov n-grams. *Lecture Notes in Computer Science*, 5137:88–107, 2008.

[SLRT91] Eugene H. Spafford, C. Langton, S. Rasmussen, y C. Taylor. Computer viruses - a form of artificial life? En *Artificial Life II*, páginas 727–745. Addison-Wesley, 1991.

[SLTW04] V. Svetnik, A. Liaw, C. Tong, y T. Wang. Application of Breiman's random forest to modeling structure-activity relationships of pharmaceutical molecules. *Multiple Classifier Systems*, páginas 334–343, 2004.

[SMEG09] A. Shabtai, R. Moskovitch, Y. Elovici, y C. Glezer. Detection of malicious code by applying machine learning classifiers on static features: A state-of-the-art survey. *Information Security Technical Report*, 14(1):16–29, 2009.

[SOG02] I. Sato, Y. Okazaki, y S. Goto. An improved intrusion detecting method based on process profiling. *Transactions of Information Processing Society of Japan (IPSJ)*, 43(11):3316–3326, 2002.

[SPDB09] I. Santos, Y.K. Penya, J. Devesa, y P. G. Bringas. N-Grams-based file signatures for malware detection. En *Proceedings of the 11th International Conference on Enterprise Information Systems (ICEIS), Volume AIDSS*, páginas 317–320. 2009.

[Spi03] D. Spinellis. Reliable identification of bounded-length viruses is NP-complete. *IEEE Transactions on Information Theory*, 49(1):280–284, 2003.

[SR03] C. Silva y B. Ribeiro. The importance of stop word removal on recall values in text categorization. En *Proceedings of the International Joint Conference on Neural Networks*, tomo 3, páginas 1661–1666. 2003.

[SS73] P. H. A. Sneath y R. R. Sokal. *Numerical taxonomy: the principles and practice of numerical classification*. WH Freeman San Francisco, 1973.

[Stå08] M. Ståhlberg. The Trojan Money Spinner. En *Virus Bulletin Conference*. 2008.

[Ste05] A. E. Stepan. Defeating polymorphism: Beyond emulation. En *Proceedings of the Virus Bulletin International Conference*. 2005.

[Ste10] Joe Stewart. Ollybone, 2010. Disponible en: http://www.joestewart.org/ollybone/.

[STF09] M. Z. Shafiq, S. M. Tabish, y M. Farooq. PE-Probe: Leveraging Packer Detection and Structural Information to Detect Malicious Portable Executables. En *Proceedings of the Virus Bulletin Conference*. 2009.

[Sub08] R. Subramanian. *Computer Security, Privacy, and Politics: Current Issues, Challenges, and Solutions*. IGI Global, 2008.

[SW49] C. E. Shannon y W. Weaver. The mathematical theory of communication. 1949.

[Swa10] William Swanson. Introduction to assembly language, 2010. Disponible en: http://www.swansontec.com/sprogram.html.

[SXCM04] A. H. Sung, J. Xu, P. Chavez, y S. Mukkamala. Static analyzer of vicious executables (SAVE). En *Proceedings of the 20th Annual Computer Security Applications Conference (ACSAC)*, páginas 326–334. 2004.

[Sym10] Symantec. Symantec webpage, 2010. Disponible en: http://www.symantec.com.

[SYS+08] M. Sharif, V. Yegneswaran, H. Saidi, P. Porras, y W. Lee. Eureka: A Framework for Enabling Static Malware Analysis. En *Proceedings of the European Symposium on Research in Computer Security (ESORICS)*, páginas 481–500. 2008.

[SZ03] Ed Skoudis y Lenny Zeltser. *Malware: Fighting Malicious Code*. Prentice Hall PTR, Upper Saddle River, NJ, USA, 2003. ISBN 0-13-101405-6.

[Szö05] P. Ször. *The art of computer virus research and defense*. Addison-Wesley Professional, 2005.

[Tor03] K. Torkkola. Feature extraction by non parametric mutual information maximization. *The Journal of Machine Learning Research*, 3:1415–1438, 2003.

[TP07] S. Tata y J. M. Patel. Estimating the Selectivity of tf-idf based Cosine Similarity Predicates. *ACM SIGMOD Record*, 36(2):75–80, 2007.

[TSA08] M. Toorani, A. A. B. Shirazi, y A. Asghar. LPKI-a Lightweight Public Key Infrastructure for the mobile environments. En *Proceedings of the 11th IEEE Singapore International Conference on Communication Systems (ICCS)*, páginas 162–166. 2008.

[TYW03] E. C. C. Tsang, D. S. Yeung, y X. Z. Wang. OFFSS: optimal fuzzy-valued feature subset selection. *IEEE transactions on fuzzy systems*, 11(2):202–213, 2003.

[UMB06] B. Üstün, W.J. Melssen, y L.M.C. Buydens. Facilitating the application of Support Vector Regression by using a universal Pearson VII function based kernel. *Chemometrics and Intelligent Laboratory Systems*, 81(1):29–40, 2006.

[Vap00] V. N. Vapnik. *The nature of statistical learning theory*. Springer, 2000.

[Vas07] A. Vasudevan. *WiLDCAT: An integrated stealth environment for dynamic malware analysis*. Tesis Doctoral, The University of Texas at Arlington, 2007.

[VH08] D. Venugopal y G. Hu. Efficient signature based malware detection on mobile devices. *Mobile Information Systems*, 4(1):33–49, 2008.

[Vir10a] Viruslist. Viruslist.com - 1970s, 2010. Disponible en: `http://www.securelist.com/en/threats/detect?chapter=106`.

[Vir10b] Viruslist. Viruslist.com - 1990s, 2010. Disponible en: `http://www.securelist.com/en/threats/detect?chapter=111`.

[Vir10c] Viruslist. Viruslist.com - early 1980s, 2010. Disponible en: `http://www.securelist.com/en/threats/detect?chapter=107`.

[Vir10d] Viruslist. Viruslist.com - the beginning - a little archeology, 2010. Disponible en: `http://www.securelist.com/en/threats/detect?chapter=105`.

[Vir10e] Viruslist. Viruslist.com - who writes malicious programs and why?, 2010. Disponible en: http://www.viruslist.com/en/viruses/encyclopedia?chapter=153280553.

[VX10] VX. VX heavens, 2010. Disponible en: http://vx.netlux.org/.

[WBV+05] Y. M. Wang, D. Beck, B. Vo, R. Roussev, C. Verbowski, y A. Johnson. Detecting stealth software with Strider GhostBuster. En *Proceedings of the International Conference on Dependable Systems and Networks*, páginas 368––377. 2005.

[WGC+02] C. Wiggers, B. Galbraith, V. Chopra, S. Li, D. Bhattacharjee, A. Bakore, R. Irani, S. Bhattacharya, y C. Fowler. *Professional Apache Tomcat*. Wrox Press Ltd. Birmingham, UK, UK, 2002.

[WH10] Wine-HQ. WineHQ - Run Windows applicaitons on Linux, BSD, Solaris and Mac OS X, 2010. Disponible en: http://www.winehq.org/.

[WHF07] C. Willems, T. Holz, y F. Freiling. Toward automated dynamic malware analysis using cwsandbox. *IEEE Security & Privacy*, páginas 32–39, 2007.

[WMCL07] A. Walenstein, R. Mathur, M. R. Chouchane, y A. Lakhotia. The design space of metamorphic malware. En *Proceedings of the 2nd International Conference on Information Warfare & Security*, páginas 241–248. 2007.

[WS04] K. Wang y S. J. Stolfo. Anomalous payload-based network intrusion detection. *Lecture Notes in Computer Science*, páginas 203–222, 2004.

[WSW09] C. Wei, A. Sprague, y G. Warner. Clustering malware-generated spam emails with a novel fuzzy string matching algorithm. En *Proceedings of the 2009 ACM symposium on Applied Computing*, páginas 889–890. ACM, 2009.

[WY10] X. Wan y J. Yang. A practical system for harvesting and monitoring hot topics on the web. En *Proceedings of the 19th international conference on World wide web*, páginas 1197–1198. 2010.

[WYY05] X. Wang, Y.L. Yin, y H. Yu. Finding collisions in the full SHA-1. *Lecture Notes in Computer Science*, 3621:17–36, 2005.

[Yam10] Motoaki Yamamura. Symantec: W95.CIH, 2010. Disponible en: `http://www.symantec.com/security_response/writeup.jsp?docid=2004-011618-0828-99`.

[Zha08] Q. Zhang. *Polymorphic and Metamorphic Malware Detection*. Tesis Doctoral, North Carolina State University, 2008.

[ZR07] Q. Zhang y D. S. Reeves. Metaaware: Identifying metamorphic malware. En *Proceedings of the 23rd Annual Computer Security Applications Conference (ACSAC)*, páginas 411–420. 2007.

[ZTW06] W. Zhu, C. Thomborson, y F. Y. Wang. Obfuscate arrays by homomorphic functions. En *Proceedings of the IEEE International Conference on Granular Computing*, páginas 770–773. 2006.

Apéndices

Apéndice

A

Desarrollo del prototipo de detección de malware NOA

En este apéndice se muestra cómo ha sido el desarrollo de un prototipo basado en los resultados obtenidos en la fase experimental de esta tesis doctoral. El prototipo es un sistema de detección de malware llamado *N-Opcode Analyser* (NOA). NOA es el resultado final de esta investigación y es prácticamente una solución anti-malware completa.

De este modo, a partir del diseño propuesto, hemos implementado un sistema anti-malware capaz de incorporar todas las funcionalidades de un sistema completo. Además, de lo expuesto en la sección anterior, hemos incluido un detector de malware desconocido basado en aprendizaje automático supervisado. El sistema consiste en una aplicación desarrollada en la plataforma *Microsoft .NET*, más concretamente en el lenguaje C#, haciendo uso del *.NET Framework 3.5* y *Windows Forms* para la implementación de la interfaz gráfica.

Por otro lado, se ha desarrollado una base de datos en *SQL Server 2008* que da soporte al proceso de detección de variantes, almacenando las representaciones de las variantes conocidas de diferentes familias de malware.

Lo que resta de apéndice está estructurado como sigue. La sección A.1 especifica cómo se ha realizado el *framework* de representación. La sección A.2 describe cómo se ha adaptado el modelo de datos propuesto para el desarrollo de NOA. En particular, se describen todas las modificaciones que

se han realizado para mejorar la eficiencia de la misma. La sección A.3 detalla cómo se ha realizado la aplicación cliente y muestra el diseño de la misma. Finalmente, la sección A.4 describe la interfaz gráfica de usuario que se ha desarrollado para realizar las diferentes tareas posibles en NOA.

A.1 *Framework* de representación

El *framework* de representación genera las representaciones de los ejecutables, tal y como se mostró en el capítulo 3. Este método genera representaciones tanto para poblar la base de datos como para formar la consulta a la misma. En concreto, este proceso consta de tres pasos:

1. **Desensamblado.** El primer paso utiliza un desensamblador externo y un módulo que procesa la salida del código ensamblador del mismo para generar la representación del ejecutable.

2. **Generación de la *frecuencia de secuencia*.** En este paso, creamos una lista que contiene la frecuencia de cada secuencia de *opcodes* posible. Cabe remarcar que las secuencias pueden ser de longitud n o $1..n$ si se selecciona la opción *recursivo*. Para calcular este valor, se generan secuencias del fichero ensamblador, contando el número de ocurrencias de cada secuencia en el programa completo. Después, es necesario dividir cada valor por el total de secuencias en el programa.

3. **Generación de la *frecuencia ponderada de secuencia*.** Finalmente, actualizamos esos valores de cada secuencia de acuerdo con los valores de relevancia de cada *opcode*. Para esto, multiplicamos la frecuencia obtenida por el valor de relevancia de todos los *opcodes* en la secuencia. Estos valores se han almacenado previamente en la base de datos.

A.2 Desarrollo de la base de datos

A partir del modelo datos propuesto, hemos realizado un diseño alternativo con el fin de maximizar su eficiencia (véase la Figura A.1). Desde el punto de vista formal, el diseño no es óptimo ya que no es lo suficientemente representativo. No obstante, se maximiza la eficiencia y escalabilidad del mismo eliminando tablas intermedias como *Sequence_model*. Por la misma razón, hemos eliminado la entidad *malware family* del modelo de datos.

En concreto, el nuevo diseño de la BBDD está formado por las siguientes tablas:

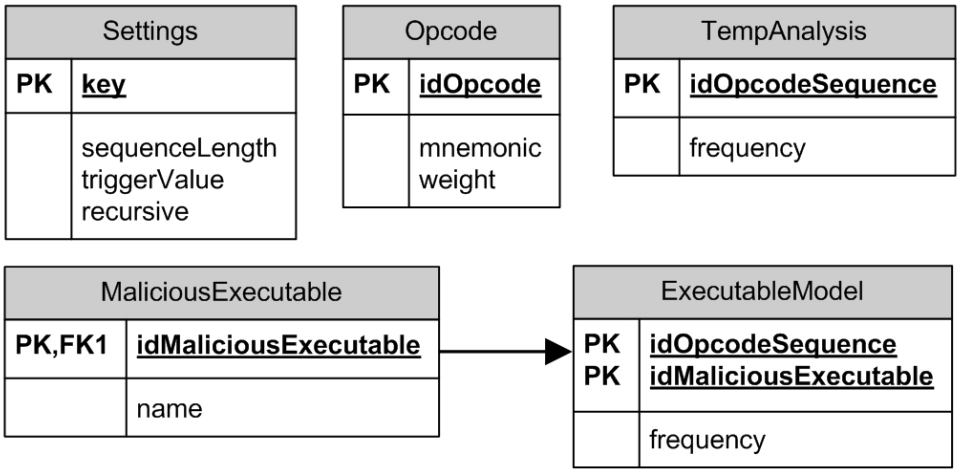

Figura A.1: Diseño final de la base de datos. Se han realizado ciertas modificaciones respecto al modelo propuesto para reducir el coste computacional derivado de las consultas a la Base de Datos.

- **Settings.** Almacena la configuración del sistema anti-malware. La Tabla A.1 muestra una especificación detallada de las columnas de esta entidad de la BBDD.
- **MaliciousExecutable.** Esta tabla de la BBDD final está destinada a almacenar el conjunto de ejecutables añadidos a la base de datos como muestras de malware. La Tabla A.2 muestra una especificación detallada de las columnas de la entidad *MaliciousExecutable* de la BBDD.
- **Opcode.** En esta tabla se almacena una fila por cada posible *opcode*. Al eliminar las tablas *SequenceModel* y *ExecutableOpcode* del diseño

Tabla A.1: Especificación de la tabla de la BBDD *Settings*.

Columna	Tipo	Descripción
sequenceLength	int	Determina el tamaño de la secuencia a analizar.
triggerValue	real	Indica el umbral de similitud para determinar un ejecutable como malware.
Recursive	bool	Determina si el análisis de un tamaño de secuencia n tiene en cuenta todas las secuencias de tamaño inferior.

Tabla A.2: Especificación de la tabla de la BBDD *MaliciousExecutable*. PK significa que ese atributo es clave primaria de la tabla.

Columna	Tipo	Descripción
idMaliciousExecutable	int (PK)	Clave de la tabla, aporta un identificador único a cada muestra de malware.
name	varchar	Determina el nombre del fichero cuya representación se agrega a la base de datos.

Tabla A.3: Especificación de la tabla de la BBDD *Opcode*. PK significa que ese atributo es clave primaria de la tabla.

Columna	Tipo	Descripción
idOpcode	int (PK)	Identificador numérico para cada *opcode*.
mnemonic	varchar	Identificador textual del *opcode*. Es necesario para su posterior comparación con los resultados del desensamblado de ejecutables, ya que estos consisten en una representación textual del código máquina.
weight	int	Valor de relevancia de cada *opcode* en la determinación de si es o no malware. Comprende valores entre 0 y 100.

inicial, por motivos de eficiencia, se eliminan también las relaciones de *Opcode* con el resto de tablas de la BBDD. La Tabla A.4 muestra una especificación detallada de las columnas de la entidad *Opcode* de la BBDD.

- **ExecutableModel.** Almacena una fila por cada posible secuencia de *opcodes*. La Tabla A.4 muestra una especificación detallada de las columnas de la entidad *ExecutableModel* de la BBDD.

  Para el almacenamiento del identificador de secuencia, decidimos generar un valor calculado a partir de los opcode que la componen. De este modo, se eliminan las entidades *sequenceModel* y *OpcodeSequence*, reduciendo el número de consultas a la BBDD y, por tanto, el proce-

Tabla A.4: Especificación de la tabla de la BBDD *ExecutableModel*. PK significa que ese atributo es clave primaria de la tabla.

Columna	Tipo	Descripción
idOpcodeSequence	int (PK)	Identificador numérico para cada secuencia.
idMaliciousExecutable	int (FK)	Columna que relaciona cada secuencia con su correspondiente fila en la tabla MaliciousExecutable.
frequency	real	Valor de frecuencia de aparición de la secuencia dada en el ejecutable indicado. Esta frecuencia se almacena ponderada con el peso de cada opcode.

Tabla A.5: Especificación de la tabla de la BBDD *TempAnalysis*. PK significa que ese atributo es clave primaria de la tabla.

Columna	Tipo	Descripción
idOpcodeSequence	int (PK)	Identificador numérico para cada secuencia.
frequency	real	Valor de frecuencia de aparición de la secuencia dada en el ejecutable indicado. Esta frecuencia se almacena ponderada con el peso de cada opcode.

samiento asociado a las restricciones de integridad en las relaciones entre esas tablas. Así, podemos calcular el identificador de secuencia a partir de los identificadores de *opcodes* obtenidos tras el desensamblado de un ejecutable (véase la ecuación A.1).

$$SequenceKey = \sum_{i=1}^{n} OpcodeKey_i \cdot 1000^{i-1} \tag{A.1}$$

donde n es la longitud seleccionada y $OpcodeKey_i$ es la clave del *opcode* en la posición i-ésima de la secuencia de *opcodes*.

De esta forma, durante el proceso de análisis (el cálculo de la similitud del coseno), únicamente se tendrán en cuenta estos identificadores a la hora de combinar los datos de las diferentes tablas.

- **TempAnalysis.** Esta tabla se utiliza para almacenar de forma temporal la representación del ejecutable que está siendo analizado. La Tabla A.5 muestra una especificación detallada de las columnas de la entidad *TempAnalysis* de la BBDD.

  Esta forma resulta más eficiente y escalable que cargar en memoria las representaciones de los ejecutables almacenados. De este modo, una vez se guarda la representación de forma temporal, un procedimiento almacenado calcula la similitud del coseno con las muestras existentes y devuelve como resultado una lista de posibles variantes con un grado de similitud superior a un umbral pasado por parámetro.

Además de modificar el diseño de la base de datos, se ha decidido implementar parte de la funcionalidad del analizador a nivel de base de datos. De este modo, se optimizan los cálculos y el número de accesos a memoria física, simplificando el manejo de la base de datos desde la propia aplicación, y permitiendo que el sistema sea más escalable.

Para ello, se han implementado una serie de funciones y procedimientos almacenados:

- **ModuleProduct.** Función que calcula el producto de los módulos de los vectores. Para ello, primero calcula el módulo de cada vector, filtrando los vectores en función del identificador de secuencia de acuerdo con el tamaño de secuencia y valor de recursividad proporcionados. La Figura A.2 muestra el código fuente en *Transact-SQL* de la función *ModuleProduct*.

- **DotProduct.** Función que calcula el producto escalar de los vectores. En este caso, los vectores están formados por las frecuencias asociadas al conjunto de secuencias de los ejecutables a comparar. La Figura A.3 muestra el código fuente en *Transact-SQL* de la función *DotProduct*.

- **CosineSimilarity.** La función *cosineSimilarity* calcula el grado de similitud entre dos ejecutables, uno presente en la tabla *tempAnalysis*, y otro de la tabla *executableModel*.

  Para ello, es necesario pasar tres parámetros a la misma: id del ejecutable a comparar, longitud de secuencia y recursividad (para incluir

```
CREATE FUNCTION [dbo].[moduleProduct]
(
   @idExecutable int,
   @sequenceLength int,
   @recursive bit
) RETURNS Real
AS
BEGIN
   DECLARE @Result REAL;
   DECLARE @a REAL;
   DECLARE @b REAL;
   DECLARE @idMinValue bigint;
   DECLARE @idMaxValue bigint;
   SET @idMaxValue = (POWER(CAST (1000 CAST bigint),
CAST(@sequenceLength CAST bigint))) -1;
   IF @recursive = 0
      SET @idMinValue = POWER(CAST(1000 AS bigint),
CAST(@sequenceLength AS bigint) - 1);
   ELSE
      SET @idMinValue = 0;
   SELECT @a = SQRT(SUM(a.[frequency]
*a.[frequency]))
   FROM [NOA].[dbo].[tempAnalysis] AS a;
   SELECT @b = SQRT(SUM(b.[frequency]
*b.[frequency]))
   FROM [NOA].[dbo].[executablemodel] AS b
   WHERE b.[idMaliciousExecutable] = @idExecutable
AND
   b.[idOpcodeSequence] <= @idMaxValue AND
b.[idOpcodeSequence] >= @idMinValue;
   SET @Result = @a * @b;
   RETURN @Result;
END
```

Figura A.2: El producto de módulos implementado en *Transact-SQL*.

en los cálculos solamente las secuencias que correspondan). A su vez, esta función hace uso de otras dos funciones, *dotProduct* y *moduleProduct*. La Figura A.4 muestra el código fuente en *Transact-SQL* de la

```
CREATE FUNCTION [dbo].[dotProduct]
(
   @idExecutable int
)
RETURNS REAL
AS
BEGIN
   DECLARE @Result REAL;
   SELECT @Result = SUM(a.[frequency] *b.[frequency])
   FROM dbo.tempAnalysis AS a INNER JOIN
   dbo.executablemodel AS b
   ON a.[idOpcodeSequence] = b.[idOpcodeSequence]
   WHERE b.[idMaliciousExecutable] = @idExecutable;
   RETURN @Result;
END
```

Figura A.3: El producto escalar implementado en *Transact-SQL*.

función *cosineSimilarity*.

- **AnalyzeExecutable.** Procedimiento almacenado en la base de datos,

```
CREATE FUNCTION [dbo].[cosineSimilarity]
(
   @idExecutable int,
   @sequenceLength int,
   @recursive bit
)
RETURNS Real
AS
BEGIN
   DECLARE @Result Real
   SET @Result = dbo.dotProduct(@idExecutable) /
   dbo.moduleProduct(@idExecutable,
   @sequenceLength, @recursive);
   RETURN @Result
END
```

Figura A.4: La función de similitud del coseno implementada en *Transact-SQL*.

```
CREATE PROCEDURE [dbo].[analyzeExecutable]
   @triggerValue REAL,
   @sequenceLength int,
   @recursive bit
)
AS
BEGIN
   SELECT temp.name, temp.similarity FROM
   (SELECT name, dbo.cosineSimilarity(idMaliciousExecutable,
@sequenceLength, @recursive) AS similarity
    FROM dbo.maliciousexecutable) temp
   WHERE temp.similarity >= @thresholdValue
   ORDER BY temp.similarity desc;
END
```

Figura A.5: El procedimiento almacenado que se utiliza para recuperar una lista de variantes similar al ejecutable en inspección, implementado en *Transact-SQL*.

que haciendo uso de la función *cosineSimilarity*, devuelve una lista con todos los ejecutables almacenados en la base de datos con un valor de similitud del coseno superior a un cierto umbral indicado como parámetro. La Figura A.5 muestra el código fuente en *Transact-SQL* de la función *AnalyzeExecutable*.

Con estas funciones, el sistema NOA es capaz de, dada una representación perteneciente al ejecutable de inspección, recuperar las variantes más similares a ese ejecutable que superen cierto umbral de similitud. En otras palabras, el sistema de *recuperación de variantes de malware*.

A.3 Diseño de la aplicación

La aplicación se ha desarrollado como herramienta de escritorio *Windows Forms*. Los diferentes módulos de la misma se han dividido en varios proyectos que se compilan en ficheros *dll* independientes. Cada paquete del diseño de clases mostrado se corresponde con un proyecto, y por tanto, con un espacio de nombres dentro de la solución. En concreto, los diferentes proyectos que hemos realizado son:

- **DataAccess.** Proporciona acceso a la base de datos a través de una única instancia (patrón *singleton*). Proporciona, además, acceso al procedimiento almacenado de comparación de ejecutables.
- **DissassembleHandler.** Da acceso al desensamblador utilizado para obtener los opcodes relativos a cada ejecutable.
- **GUI.** Implementa la interfaz gráfica de la aplicación, así como el flujo de control de los diferentes procesos de análisis y actualización de la base de datos.
- **OpcodeAnalyzer.** Proporciona la funcionalidad necesaria para calcular la representación de un ejecutable a partir de su código desensamblado.
- **UnknownMalwareDetection.** Encapsula el uso del clasificador de malware desconocido.
- **WeightReader.** Proporciona acceso de lectura al fichero xml que contiene las relevancias de los distintos opcodes.

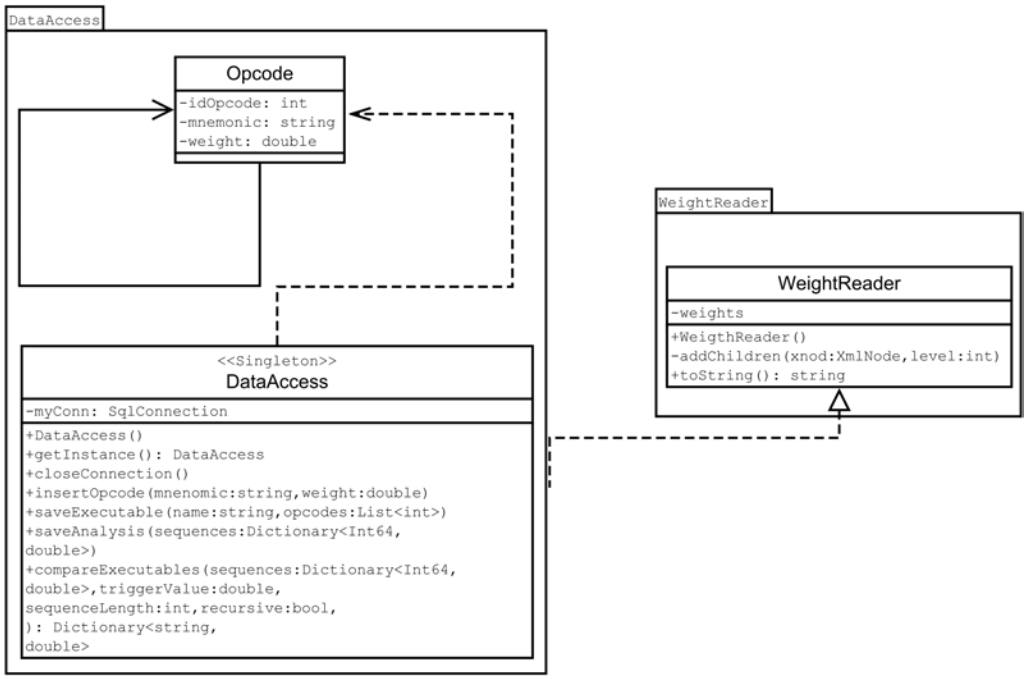

Figura A.6: Diagrama de clases para los proyectos *DataAccess* y *WeightReader*.

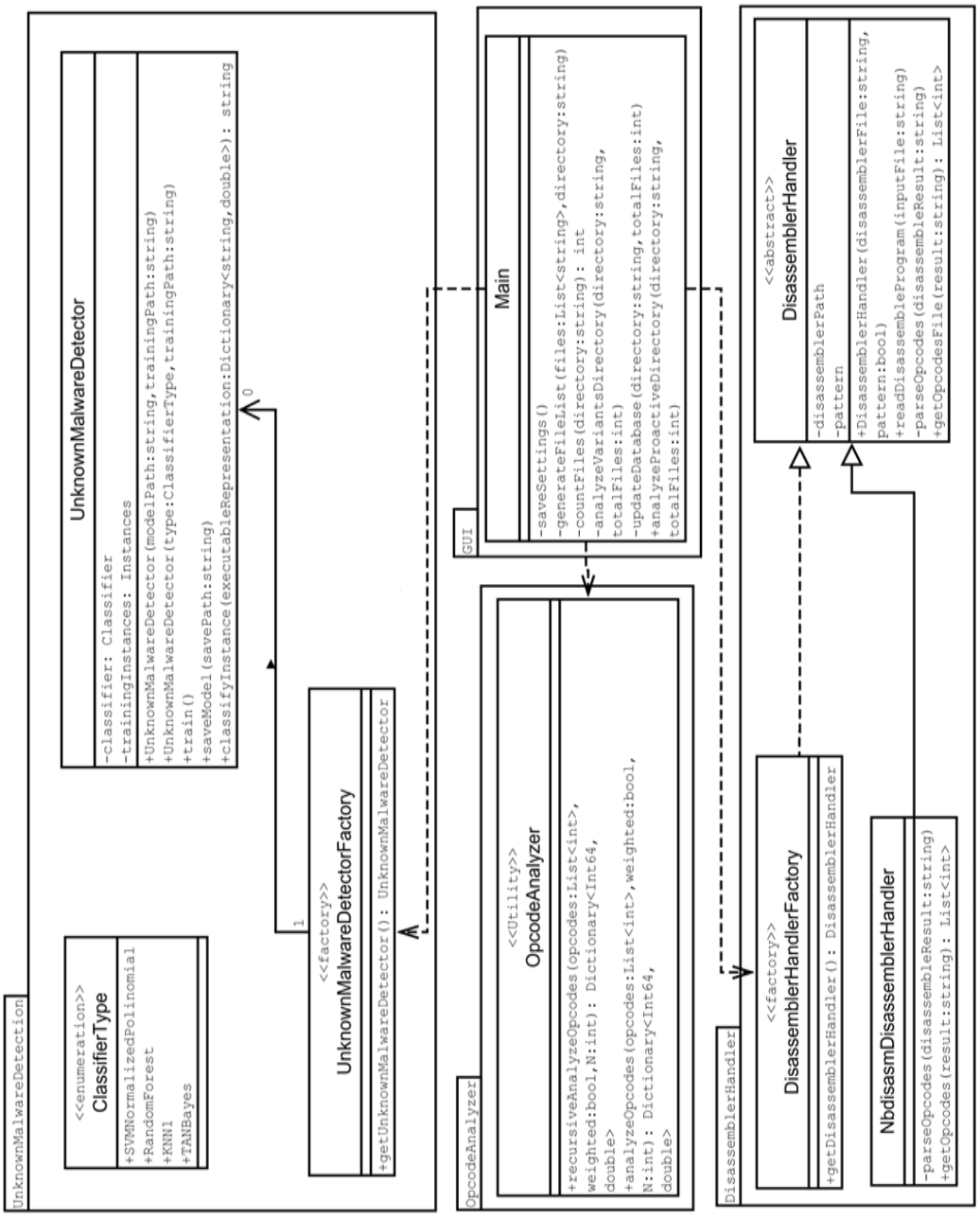

Figura A.7: Diagrama de clases para los proyectos *UnknownMalwareDetection*, *DissassembleHandler*, *GUI* y *OpcodeAnalyzer*.

La Figura A.6 muestra el diagrama de clases asociado a las clases de acceso a datos. La clase *DataAccess* proporciona el método *updateWeights* que haciendo uso de la clase *WeightReader* vuelca sobre la base de datos

las relevancias asociadas a cada *opcode*, leídas de un fichero *xml*. Además de los métodos de volcado de ejecutables sobre la base de datos, *DataAccess* proporciona acceso al procedimiento almacenado de comparación de ejecutables mediante la operación *compareExecutable*.

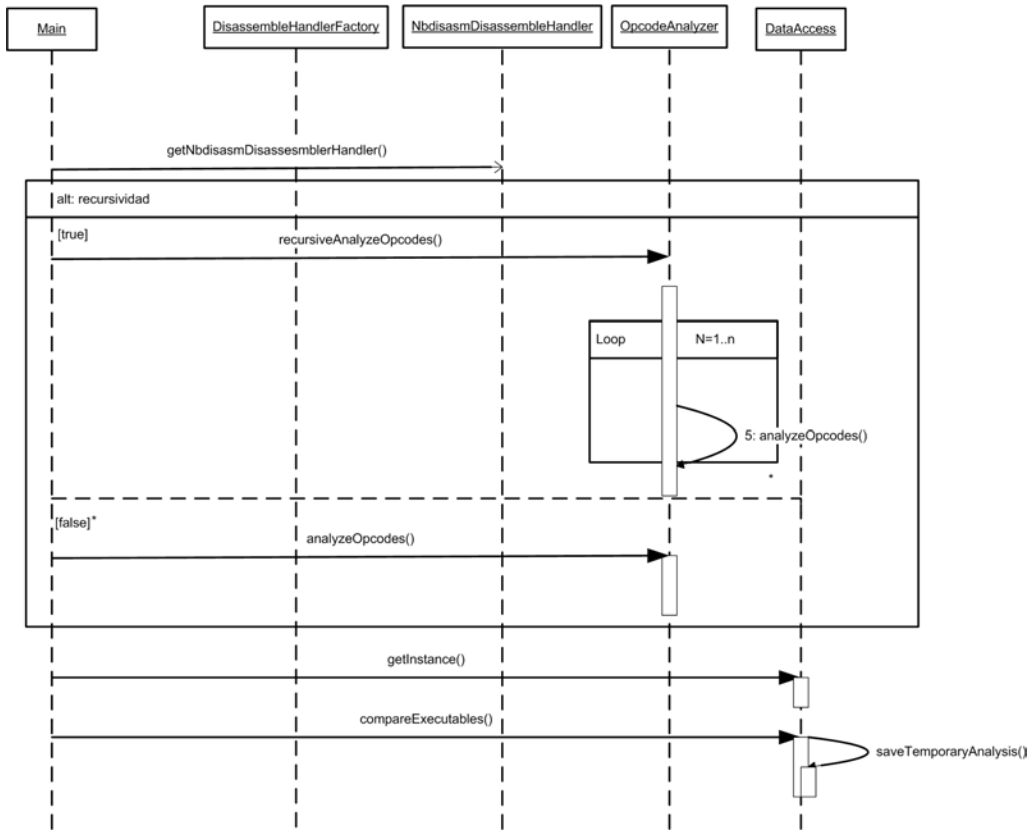

Figura A.8: Diagrama de secuencia para el análisis de un fichero ejecutable mediante el método de detección de variantes.

La Figura A.7 muestra el diagrama de clases asociado al proceso de análisis. El proyecto *DisassembleHandler* proporciona una clase factoría y una clase abstracta para encapsular el uso de diferentes desensambladores. En nuestro caso, solamente implementamos una clase que hace uso del desensamblador New Basic (*NbdisasmDissassembleHandler*). El método *getOpcodesInFile* proporciona una lista ordenada con los identificadores asociados a cada opcode.

La clase *OpcodeAnalyzer* genera una estructura de datos *Dictionary* que contiene cada secuencia con su frecuencia de aparición ponderada asociada. El método *analyzeOpcodes* genera el vector para una longitud N, mien-

tras que el método *recursiveAnalyzeOpcodes* lo hace recursivamente, llamando a su vez al método *analyzeOpcodes*.

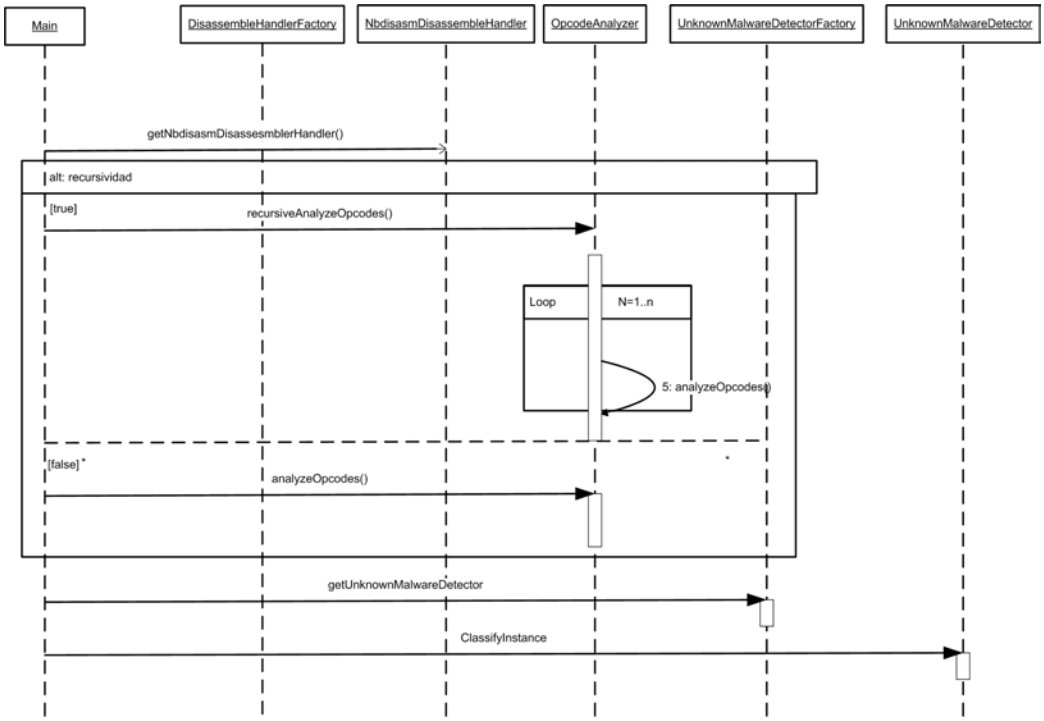

Figura A.9: Diagrama de secuencia para el análisis de un fichero ejecutable mediante el método de detección de malware desconocido..

El proyecto *UnknownMalwareDetector* encapsula el uso del clasificador generado mediante WEKA. Proporciona una clase Factoría para crear instancias de la clase *UnknownMalwareDetector*, que dispone de métodos para entrenar un modelo (*train*) según diferentes algoritmos (*ClassifierType*), guardar el modelo (*saveModel*), y después utilizarlo para clasificar ejecutables a partir de su representación (*classifyInstance*).

El diagrama de secuencia de la Figura A.8 muestra el proceso de análisis de variantes mientras que el diagrama de secuencia de la Figura A.9 muestra el proceso de análisis de malware desconocido.

La interfaz consiste en una única ventana basada en pestañas. Además, la clase que la implementa tiene una serie de métodos para llevar a cabo los procesos de análisis. *countFiles()* y *generateFileList()* calculan recursivamente la lista de ficheros a analizar dentro de un directorio. Posteriormente, en cualquiera de los tres procesos que lleva a cabo la interfaz (*analyzeVariantsDirectory*, *updateDatabase*, *analyzeProactiveDirectory*), se comprobará si el

fichero es o no un ejecutable de tipo PE. Para ello, basta con comprobar que los dos primeros bytes del fichero son 'M' y 'Z', o 'Z' y 'M'.

A.4 Interfaz gráfica

La interfaz de la aplicación ha sido implementada mediante *Windows Forms*. Se ha elegido la navegación basada en pestañas y un tamaño de ventana fijo, para que el aspecto final sea similar a un anti-virus comercial cualquiera del mercado.

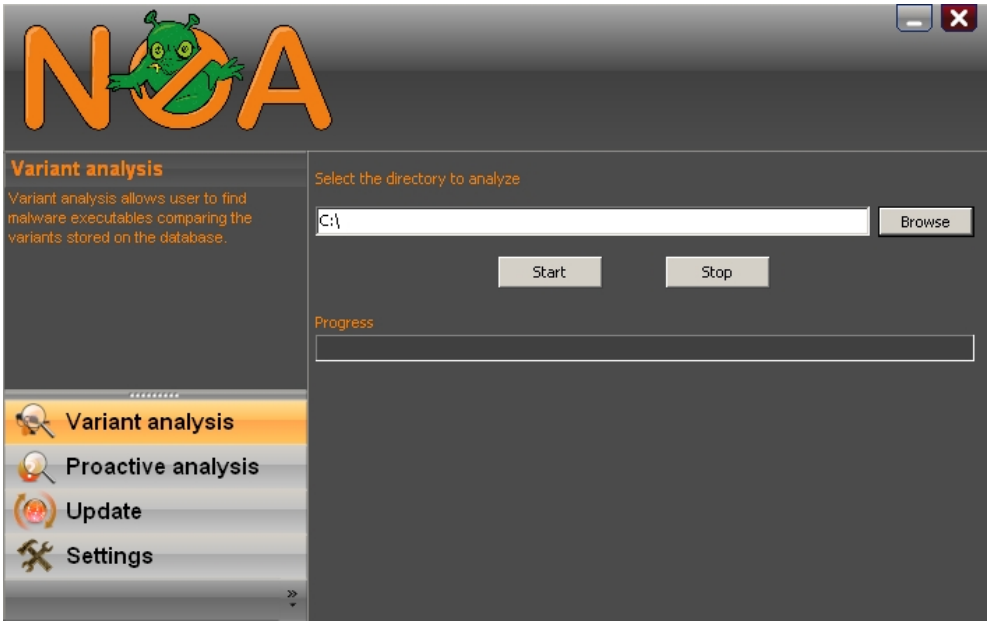

Figura A.10: Interfaz gráfica de NOA. Pestaña de análisis de variantes.

En la Figura A.10 podemos observar el aspecto final del analizador de variantes. En la parte izquierda de la ventana tenemos un menú que permite navegar entre las diferentes pestañas de la aplicación. Un panel en la parte superior da una breve explicación sobre la pestaña que está abierta en ese momento. El menú da acceso a 4 pestañas: análisis de variantes, análisis proactivo (analizador de malware desconocido), actualización (carga de la base de datos para el análisis de variantes) y configuración.

Las 3 primeras pestañas presentan una estructura similar. En la parte superior encontramos una caja de texto que contiene la raíz del directorio sobre el que se desea realizar el proceso de análisis o actualización. El botón *Browse*, despliega un menú de selección de directorios, para facilitar la

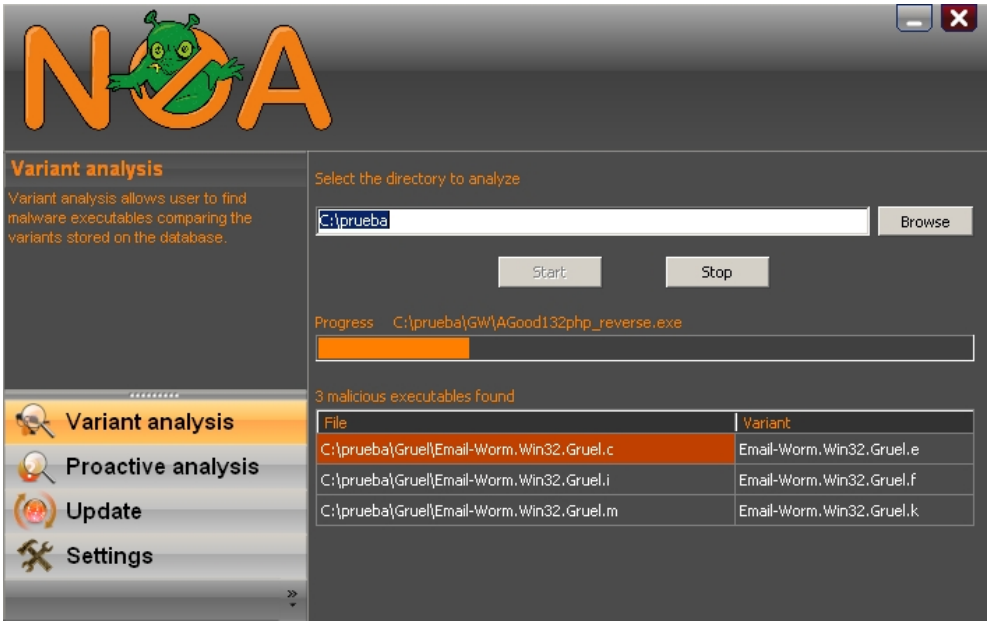

Figura A.11: Interfaz gráfica de NOA. Proceso de análisis de variantes.

tarea al usuario. Bajo esta caja de texto, encontramos dos botones que lanzarán o pararán el proceso, *Start* y *Stop*. Una barra de progreso mostrará el progreso del proceso de análisis que se está ejecutando en el momento actual. NOA puede lanzar varios procesos de análisis y/o actualización simultáneamente, siempre que estos se ejecuten en pestañas diferentes de la aplicación. Esto quiere decir que el usuario puede realizar un análisis de variantes y de malware desconocido al mismo tiempo.

Una vez lanzado el proceso de análisis de variantes nos encontramos con la ventana que se muestra en la Figura A.11. La barra de progreso, muestra el avance del proceso. Un listado muestra en la parte inferior, el número de variantes encontradas, así como una relación del fichero analizado y la variante con mayor similitud. Tal como se observa en la Figura A.11, el botón de comienzo se encuentra deshabilitado, encontrándose habilitado el botón de parada. Al hacer *click* se para el proceso.

La Figura A.12 muestra la segunda pestaña, relativa al análisis proactivo. La diferencia principal respecto a la pestaña anterior es el listado de ficheros sospechosos. En este caso, como se trata de malware desconocido, no aporta información acerca de la familia o variante de la que se trata, tan sólo indicando si el fichero el malicioso o no. Este proceso carga un modelo de aprendizaje automático previamente entrenado. En la interfaz gráfica desarrollado no existe la posibilidad de actualizar este clasificador, siendo

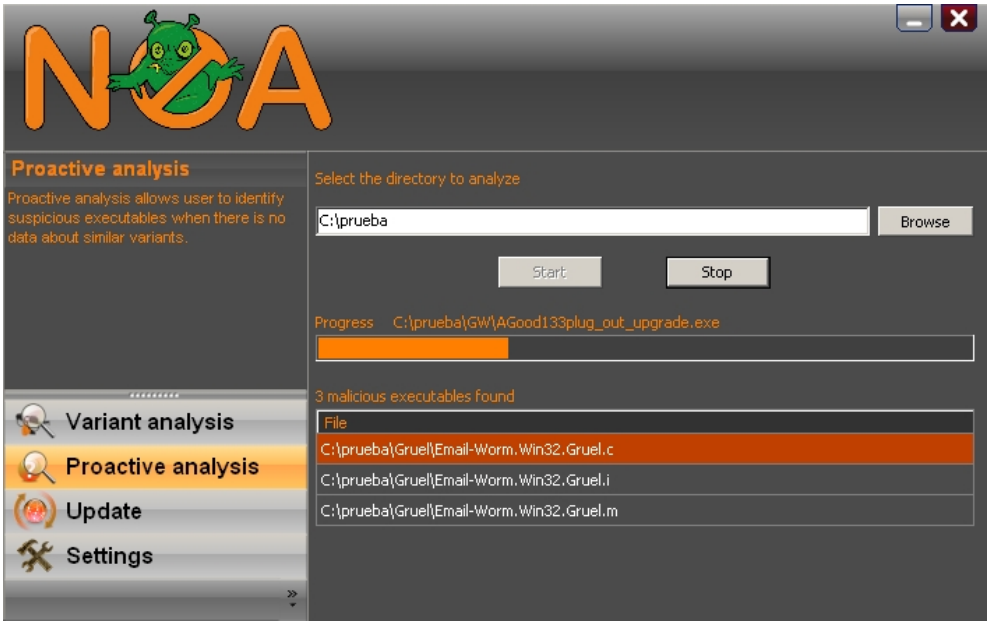

Figura A.12: Interfaz gráfica de NOA. Pestaña de análisis de malware desconocido.

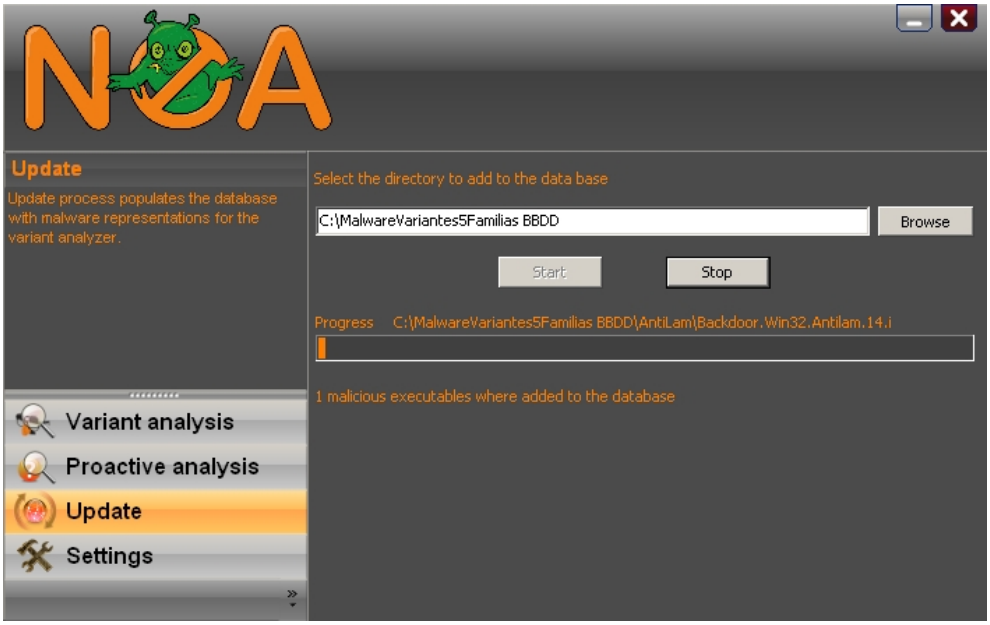

Figura A.13: Interfaz gráfica de NOA. Pestaña de actualización de la base de datos de variantes.

éste un proceso *off-line*.

La Figura A.13 muestra la tercera pestaña, que se encarga de actualizar la base de datos para el análisis de variantes. En este proceso, el usuario selecciona un directorio en el que se encuentran variantes de malware a insertar en la base de datos. El sistema se encarga de insertar éstas en la base de datos. Para el nombre de la familia y el nombre del espécimen, se utiliza el nombre del fichero.

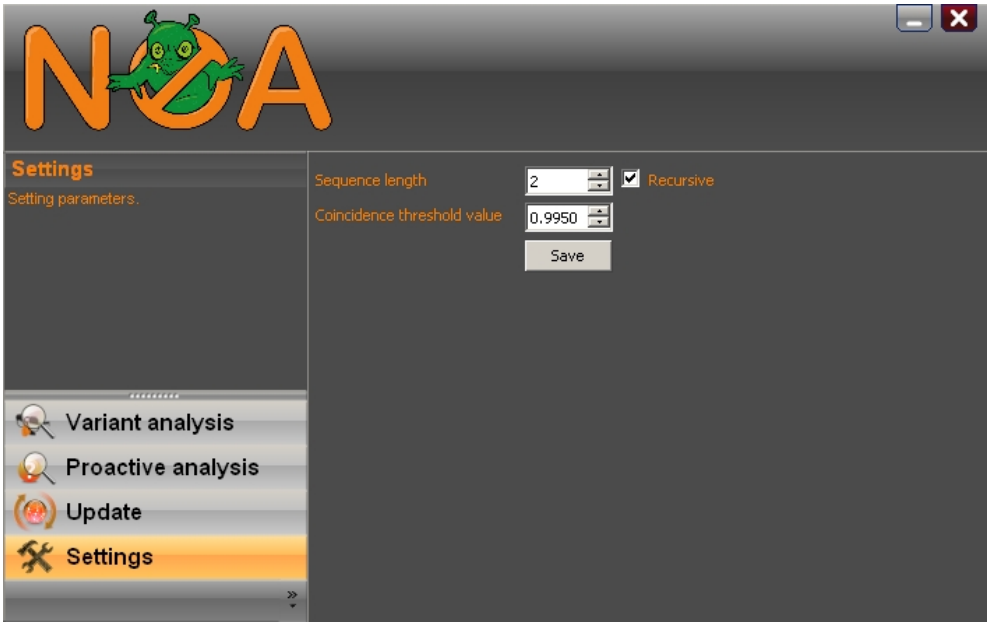

Figura A.14: Interfaz gráfica de NOA. Pestaña de configuración de la aplicación.

Finalmente, el proceso de configuración (véase la Figura A.14) se realiza mediante la cuarta pestaña de la aplicación. Utilizando la interfaz de NOA, se pueden configurar: el tamaño de la secuencia de *opcodes*, si la representación es recursiva o no, y umbral de confianza en la similitud con las variantes de la base de datos.

Apéndice

B

Fichero de relevancias de códigos operacionales

```
<?xml version=``1.0'' encoding=``utf-8'' ?>
<configuracion>
  <weightsTable>
    <retf>100</retf>
    <subss>100</subss>
    <subpd>100</subpd>
    <sqrtss>100</sqrtss>
    <sqrtpd>100</sqrtpd>
    <shufpd>100</shufpd>
    <punpcklqdq>100</punpcklqdq>
    <pshufd>100</pshufd>
    <mulss>100</mulss>
    <mulpd>100</mulpd>
    <movupd>100</movupd>
    <movntdq>100</movntdq>
    <movmskpd>100</movmskpd>
    <movhpd>100</movhpd>
    <movdq2q>100</movdq2q>
    <minss>100</minss>
    <maxss>100</maxss>
```

```
<maxsd>100</maxsd>
<maxpd>100</maxpd>
<divpd>100</divpd>
<cvttpd2pi>100</cvttpd2pi>
<cvttpd2dq>100</cvttpd2dq>
<cvtsi2ss>100</cvtsi2ss>
<cvtsd2si>100</cvtsd2si>
<cvtpi2pd>100</cvtpi2pd>
<cvtpd2dq>100</cvtpd2dq>
<comisd>100</comisd>
<psubq>99,70282318</psubq>
<psraw>99,48717949</psraw>
<movss>99,38271605</movss>
<setnp>99,34640523</setnp>
<fyl2xp1>99,33035714</fyl2xp1>
<cmovge>99,29824561</cmovge>
<movdqu>99,29328622</movdqu>
<setno>99,18699187</setno>
<fnop>99,09613804</fnop>
<paddsb>98,92307692</paddsb>
<ucomiss>98,6013986</ucomiss>
<bsf>98,59872611</bsf>
<psubusb>98,58490566</psubusb>
<fcmovnbe>98,53333333</fcmovnbe>
<comiss>98,49056604</comiss>
<fdecstp>98,47715736</fdecstp>
<pmaxub>98,44559585</pmaxub>
<fcmove>98,36400818</fcmove>
<sysexit>98,21428571</sysexit>
<rsm>98,21428571</rsm>
<fcomi>97,99897383</fcomi>
<psubusw>97,98657718</psubusw>
<paddq>97,97979798</paddq>
<setp>97,84946237</setp>
<xadd>97,76871916</xadd>
<cvtps2pi>97,71428571</cvtps2pi>
<fbstp>97,70143976</fbstp>
<cmovnp>97,69736842</cmovnp>
<fcmovne>97,59384023</fcmovne>
```

```
<pavgw>97,56944444</pavgw>
<cmovle>97,46835443</cmovle>
<psadbw>97,36842105</psadbw>
<pmaxsw>97,35099338</pmaxsw>
<movnti>97,35099338</movnti>
<pminub>97,30639731</pminub>
<cmovp>97,2972973</cmovp>
<fcmovu>97,27685325</fcmovu>
<fucomip>97,09962169</fucomip>
<fcmovnb>97,04861111</fcmovnb>
<cvtdq2ps>97,02970297</cvtdq2ps>
<pcmpeqd>97,00787402</pcmpeqd>
<fcmovbe>96,98009609</fcmovbe>
<cmovns>96,85714286</cmovns>
<maskmovq>96,80851064</maskmovq>
<psrad>96,74796748</psrad>
<rsqrtps>96,71641791</rsqrtps>
<fcmovb>96,67119565</fcmovb>
<ffree>96,66182874</ffree>
<fidivr>96,53834676</fidivr>
<fucom>96,39519359</fucom>
<pminsw>96,35036496</pminsw>
<fucomp>96,26485569</fucomp>
<psrlw>96,1038961</psrlw>
<psrld>96,08938547</psrld>
<cmpxchg8b>95,90062112</cmpxchg8b>
<cvtpi2ps>95,89041096</cvtpi2ps>
<pcmpgtd>95,85062241</pcmpgtd>
<pmulhuw>95,81395349</pmulhuw>
<cmovg>95,63318777</cmovg>
<paddusw>95,52238806</paddusw>
<divps>95,47511312</divps>
<movdqa>95,46875</movdqa>
<fcomip>95,45772187</fcomip>
<wrmsr>95,20153551</wrmsr>
<fisubr>95,16612829</fisubr>
<packsswb>95,13513514</packsswb>
<psubsb>95,1048951</psubsb>
<rcpps>95,09803922</rcpps>
```

```
<pinsrw>94,81481481</pinsrw>
<iret>94,49004953</iret>
<btc>94,4646098</btc>
<aad>94,29776726</aad>
<fldl2t>94,21965318</fldl2t>
<fstenv>94,03286979</fstenv>
<cmovs>94,02173913</cmovs>
<sqrtps>93,98907104</sqrtps>
<loope>93,92485885</loope>
<lds>93,81135707</lds>
<bsr>93,60580092</bsr>
<movntps>93,52941176</movntps>
<minps>93,23671498</minps>
<xlat>93,19593284</xlat>
<sti>93,17740683</sti>
<fincstp>93,17269076</fincstp>
<stc>93,02238026</stc>
<cmc>93,01987397</cmc>
<pcmpgtw>92,95774648</pcmpgtw>
<cmovbe>92,7756654</cmovbe>
<popf>92,6904717</popf>
<lahf>92,65367316</lahf>
<rdmsr>92,63803681</rdmsr>
<ficomp>92,57641921</ficomp>
<into>92,57606491</into>
<maxps>92,2705314</maxps>
<pmuludq>92,1875</pmuludq>
<loop>92,09858644</loop>
<cli>92,0920641</cli>
<unpckhps>91,98113208</unpckhps>
<cvttps2pi>91,42857143</cvttps2pi>
<out>91,38079507</out>
<jnp>91,28724879</jnp>
<fistp>91,12643401</fistp>
<psubd>90,90909091</psubd>
<movq>89,7905065</movq>
<das>89,76499854</das>
<bswap>89,59261092</bswap>
<jp>89,47351739</jp>
```

```
<rdtsc>89,27315358</rdtsc>
<aaa>88,91231306</aaa>
<fisub>88,88121547</fisub>
<cmovl>88,51851852</cmovl>
<fldz>88,49033063</fldz>
<rdpmc>88,44884488</rdpmc>
<cmovo>88,44086022</cmovo>
<fchs>88,43099378</fchs>
<seto>88,26086957</seto>
<ud2>88,21656051</ud2>
<andnps>88</andnps>
<fsincos>87,66233766</fsincos>
<ftst>87,60958504</ftst>
<aam>87,40712661</aam>
<fldenv>87,33223452</fldenv>
<hlt>87,18049796</hlt>
<fist>87,1119654</fist>
<xchg>87,00103544</xchg>
<fxch>86,93866031</fxch>
<fimul>86,84459587</fimul>
<lods>86,73359324</lods>
<pushf>86,6642367</pushf>
<punpckhbw>86,5625</punpckhbw>
<jno>86,45804201</jno>
<cmovae>86,26760563</cmovae>
<fbld>86,03782396</fbld>
<pcmpgtb>85,91549296</pcmpgtb>
<fxtract>85,71428571</fxtract>
<clc>85,66508514</clc>
<int>85,36622596</int>
<jcxz>85,17774594</jcxz>
<fucompp>85,04016064</fucompp>
<lgs>84,86486486</lgs>
<pandn>84,84848485</pandn>
<sysenter>84,58333333</sysenter>
<aas>84,56732216</aas>
<setb>83,37438424</setb>
<les>83,17326366</les>
<fucomi>82,95605859</fucomi>
```

```
<pextrw>82,67716535</pextrw>
<sgdt>82,63852243</sgdt>
<daa>82,08799403</daa>
<punpckhdq>81,98614319</punpckhdq>
<subsd>81,81818182</subsd>
<in>80,81183796</in>
<punpckldq>80,74534161</punpckldq>
<fsave>80,35800677</fsave>
<paddusb>80,23952096</paddusb>
<xorpd>80</xorpd>
<fprem1>80</fprem1>
<pcmpeqw>79,89690722</pcmpeqw>
<fldln2>79,59183673</fldln2>
<movmskps>78,98832685</movmskps>
<unpcklps>78,96825397</unpcklps>
<rol>78,89352492</rol>
<pavgb>78,74396135</pavgb>
<rcl>78,7243831</rcl>
<ficom>78,60782951</ficom>
<enter>78,39758806</enter>
<ror>78,20782005</ror>
<movntq>78,16733068</movntq>
<fld1>78,12974203</fld1>
<fxam>77,71520515</fxam>
<cbw>77,66387419</cbw>
<db>77,34950635</db>
<fidiv>77,00528867</fidiv>
<fsqrt>76,91309987</fsqrt>
<setns>76,8707483</setns>
<loopne>76,20017553</loopne>
<cmovb>76,12903226</cmovb>
<frstor>75,72888114</frstor>
<orps>75,71428571</orps>
<clts>75,57251908</clts>
<cmove>75</cmove>
<fsubrp>74,78873239</fsubrp>
<addps>74,617737</addps>
<std>74,18123563</std>
<cmovne>73,68421053</cmovne>
```

```
<fdivr>73,15620346</fdivr>
<pcmpeqb>72,44897959</pcmpeqb>
<rcr>71,7446313</rcr>
<cmpps>71,18644068</cmpps>
<psubb>70,80745342</psubb>
<paddb>70,32520325</paddb>
<addsd>70</addsd>
<lar>69,34460888</lar>
<cmova>69,17293233</cmova>
<invd>69,14893617</invd>
<arpl>68,3423567</arpl>
<cvtps2pd>67,03539823</cvtps2pd>
<int3>66,70805768</int3>
<frndint>66,66666667</frndint>
<bound>66,65292937</bound>
<sahf>66,45495495</sahf>
<faddp>66,21404324</faddp>
<ret>66,15264847</ret>
<setl>65,96618357</setl>
<ins>65,86680085</ins>
<sldt>65,76806879</sldt>
<andps>65,66523605</andps>
<pusha>65,5399718</pusha>
<outs>65,44711738</outs>
<pxor>64,9122807</pxor>
<mulsd>64,28571429</mulsd>
<xorps>64,1025641</xorps>
<f2xm1>63,82978723</f2xm1>
<popa>63,01976871</popa>
<bt>63,01204819</bt>
<fscale>62,8458498</fscale>
<wait>62,78162081</wait>
<jns>62,63971953</jns>
<test>62,44614005</test>
<pmovmskb>62,42424242</pmovmskb>
<pshufw>62,17948718</pshufw>
<fabs>61,9669612</fabs>
<mul>61,4649107</mul>
<movzx>61,31044675</movzx>
```

```
<jo>60,61441746</jo>
<je>60,49451204</je>
<fcom>60,40782292</fcom>
<wbinvd>60,11450382</wbinvd>
<setbe>59,88372093</setbe>
<cmpxchg>59,38461538</cmpxchg>
<cmps>58,50625211</cmps>
<lfs>58,44155844</lfs>
<pslld>57,4204947</pslld>
<pmaddwd>57,27109515</pmaddwd>
<movapd>57,14285714</movapd>
<nop>56,75179641</nop>
<mulps>56,27705628</mulps>
<pmullw>56,25</pmullw>
<sets>56,21621622</sets>
<fstp>56,09716484</fstp>
<js>55,75258859</js>
<subps>55,48387097</subps>
<cmovno>55,16224189</cmovno>
<sar>54,87642104</sar>
<fsubr>54,47577945</fsubr>
<jne>54,45840164</jne>
<packssdw>54,20168067</packssdw>
<psubsw>54,06360424</psubsw>
<fld>53,85520769</fld>
<psubw>53,50553506</psubw>
<neg>53,39009098</neg>
<fcmovnu>53,29181495</fcmovnu>
<setge>52,93670886</setge>
<setle>51,55159038</setle>
<shrd>51,38529738</shrd>
<movlps>51,18790497</movlps>
<cld>50,80080037</cld>
<cmp>50,74478572</cmp>
<fldlg2>50,42194093</fldlg2>
<cpuid>50,26455026</cpuid>
<rsqrtss>50</rsqrtss>
<movsd>50</movsd>
<andpd>50</andpd>
```

```
<movsx>49,6882352</movsx>
<mov>49,54911918</mov>
<fiadd>49,12820513</fiadd>
<ja>48,82717195</ja>
<fcos>48,74623872</fcos>
<fmul>48,20925969</fmul>
<shl>47,95260345</shl>
<fpatan>47,22222222</fpatan>
<pop>47,12944381</pop>
<punpckhwd>47,02970297</punpckhwd>
<cmppd>44,44444444</cmppd>
<fyl2x>44,37299035</fyl2x>
<jmp>44,2098602</jmp>
<fptan>43,82716049</fptan>
<fdiv>43,11028828</fdiv>
<not>42,99982957</not>
<paddw>42,65232975</paddw>
<paddsw>42,56410256</paddsw>
<setne>42,32366286</setne>
<imul>41,78422163</imul>
<btr>41,57303371</btr>
<prefetcht2>41,26679463</prefetcht2>
<jl>40,88373671</jl>
<fcompp>40,65074135</fcompp>
<lsl>40,35532995</lsl>
<setg>40,22177419</setg>
<finit>40</finit>
<jbe>38,70349896</jbe>
<fstsw>38,11797018</fstsw>
<lss>37,88395904</lss>
<psllq>37,5</psllq>
<shr>37,32392344</shr>
<sete>36,93399737</sete>
<fclex>36,67412636</fclex>
<xor>34,51408578</xor>
<psrlq>34,39716312</psrlq>
<jle>34,26514482</jle>
<sub>33,9244064</sub>
<movlpd>33,33333333</movlpd>
```

```
<punpcklbw>32,8358209</punpcklbw>
<fsin>32,22416813</fsin>
<fldcw>32,13127513</fldcw>
<add>32,06928502</add>
<fprem>31,92090395</fprem>
<adc>31,53257485</adc>
<fadd>30,81687755</fadd>
<push>30,54649586</push>
<movhps>29,92248062</movhps>
<fldpi>29,72972973</fldpi>
<call>29,26642272</call>
<fcomp>28,75146542</fcomp>
<fst>27,86961367</fst>
<psllw>27,57009346</psllw>
<fdivrp>27,45367193</fdivrp>
<bts>27,00329308</bts>
<fsub>26,5447708</fsub>
<div>25,5518102</div>
<packuswb>24,67532468</packuswb>
<fdivp>24,56140351</fdivp>
<sbb>23,61877372</sbb>
<fild>22,43255197</fild>
<fmulp>22,42907801</fmulp>
<fsubp>22,37330037</fsubp>
<setae>22,22222222</setae>
<pand>22</pand>
<and>20,5575516</and>
<punpcklwd>19,07894737</punpcklwd>
<leave>18,55602421</leave>
<lfence>17,1314741</lfence>
<movups>17,00975081</movups>
<dec>16,48726132</dec>
<seta>16,39344262</seta>
<movs>14,5863316</movs>
<jae>14,34672861</jae>
<jg>13,74873989</jg>
<idiv>13,59728855</idiv>
<shufps>12,27272727</shufps>
<paddd>12,16931217</paddd>
```

```
    <shld>11,80274859</shld>
    <ucomisd>11,11111111</ucomisd>
    <lea>9,910670492</lea>
    <inc>9,490482858</inc>
    <scas>9,220048668</scas>
    <fldl2e>8,415841584</fldl2e>
    <pmulhw>7,975460123</pmulhw>
    <movaps>7,629427793</movaps>
    <movd>6,875</movd>
    <jb>6,659128724</jb>
    <or>6,180889801</or>
    <jge>6,139306847</jge>
    <fstcw>4,877606528</fstcw>
    <stos>3,357017144</stos>
    <cwd>3,104999289</cwd>
    <por>1,136925358</por>
    <emms>0,520833333</emms>
    <unpcklpd>0</unpcklpd>
    <orpd>0</orpd>
    <divss>0</divss>
    <cvtdq2pd>0</cvtdq2pd>
    <cmpsd>0</cmpsd>
    <addss>0</addss>
    <addpd>0</addpd>
  </weightsTable>
</configuracion>
```

Apéndice

C

Muestras empleadas para la validación del detector de variantes

En este apéndice, detallamos las variantes de familias de malware que hemos utilizado para validar nuestro detector de variantes, enumerando cuales de ellas se han utilizado como base de conocimiento y cuales se han utilizado para analizar y validar el detector.

Tabla C.1: Muestras de la familia *Antilam* utilizadas en la validación.

Ejecutables en la BBDD	**Ejecutables para analizar**
Backdoor.Win32.Antilam.13.b	Backdoor.Win32.Antilam.20.q
Backdoor.Win32.Antilam.14.i	
Backdoor.Win32.Antilam.20.n	
Backdoor.Win32.Antilam.20.p	
Backdoor.Win32.Antilam.20.s	
Backdoor.Win32.Antilam.gen	
6	1

Tabla C.2: Muestras de la familia *Bancodor* utilizadas en la validación.

Ejecutables en la BBDD	**Ejecutables para analizar**
Backdoor.Win32.Bancodor.b	Backdoor.Win32.Bancodor.c
Backdoor.Win32.Bancodor.d	Backdoor.Win32.Bancodor.r
Backdoor.Win32.Bancodor.e	
Backdoor.Win32.Bancodor.f	
Backdoor.Win32.Bancodor.j	
Backdoor.Win32.Bancodor.m	
Backdoor.Win32.Bancodor.t	
Backdoor.Win32.Bancodor.u	
Backdoor.Win32.Bancodor.x	
9	2

Tabla C.3: Muestras de la familia *Gruel* utilizadas en la validación.

Ejecutables en la BBDD	**Ejecutables para analizar**
Email-Worm.Win32.Gruel.a	Email-Worm.Win32.Gruel.c
Email-Worm.Win32.Gruel.b	Email-Worm.Win32.Gruel.i
Email-Worm.Win32.Gruel.e	Email-Worm.Win32.Gruel.m
Email-Worm.Win32.Gruel.f	
Email-Worm.Win32.Gruel.g	
Email-Worm.Win32.Gruel.h	
Email-Worm.Win32.Gruel.j	
Email-Worm.Win32.Gruel.k	
Email-Worm.Win32.Gruel.l	
Email-Worm.Win32.Gruel.n	
10	3

Tabla C.4: Muestras de la familia *Hupigon* utilizadas en la validación.

Ejecutables en la BBDD	**Ejecutables para analizar**
Backdoor.Win32.Hupigon.abc	Backdoor.Win32.Hupigon.ajt
Backdoor.Win32.Hupigon.abu	Backdoor.Win32.Hupigon.axn
Backdoor.Win32.Hupigon.aj	Backdoor.Win32.Hupigon.blb
Backdoor.Win32.Hupigon.akx	Backdoor.Win32.Hupigon.dl
Backdoor.Win32.Hupigon.amc	Backdoor.Win32.Hupigon.gb
Backdoor.Win32.Hupigon.are	Backdoor.Win32.Hupigon.up
Backdoor.Win32.Hupigon.awl	Backdoor.Win32.Hupigon.vw
Backdoor.Win32.Hupigon.awn	
Backdoor.Win32.Hupigon.bbk	
Backdoor.Win32.Hupigon.blh	
Backdoor.Win32.Hupigon.bqz	
Backdoor.Win32.Hupigon.bto	
Backdoor.Win32.Hupigon.ccf	
Backdoor.Win32.Hupigon.dh	
Backdoor.Win32.Hupigon.eg	
Backdoor.Win32.Hupigon.ej	
Backdoor.Win32.Hupigon.fo	
Backdoor.Win32.Hupigon.g	
Backdoor.Win32.Hupigon.j	
Backdoor.Win32.Hupigon.lp	
Backdoor.Win32.Hupigon.qk	
Backdoor.Win32.Hupigon.r	
Backdoor.Win32.Hupigon.rc	
Backdoor.Win32.Hupigon.ri	
Backdoor.Win32.Hupigon.sc	
Backdoor.Win32.Hupigon.ub	
Backdoor.Win32.Hupigon.vh	
Backdoor.Win32.Hupigon.xr	
Backdoor.Win32.Hupigon.ze	
29	7

Tabla C.5: Muestras de la familia *PcClient* utilizadas en la validación.

Ejecutables en la BBDD	Ejecutables para analizar
Backdoor.Win32.PcClient.ah	Backdoor.Win32.PcClient.bn
Backdoor.Win32.PcClient.au	Backdoor.Win32.PcClient.du
Backdoor.Win32.PcClient.b	Backdoor.Win32.PcClient.hw
Backdoor.Win32.PcClient.bg	Backdoor.Win32.PcClient.pj
Backdoor.Win32.PcClient.bm	
Backdoor.Win32.PcClient.dq	
Backdoor.Win32.PcClient.fb	
Backdoor.Win32.PcClient.h	
Backdoor.Win32.PcClient.ik	
Backdoor.Win32.PcClient.jl	
Backdoor.Win32.PcClient.m	
Backdoor.Win32.PcClient.nu	
Backdoor.Win32.PcClient.qe	
Backdoor.Win32.PcClient.sd	
Backdoor.Win32.PcClient.wj	
Backdoor.Win32.PcClient.xc	
16	4

Tabla C.6: Ejecutables benignos para la validación del sistema de detección de variantes.

Ejecutables en la BBDD	Ejecutables para analizar
0	1000

Índice alfabético

árbol de decisión, 46, 123–124

agrupamiento, *véase* clustering
algoritmo basado en reglas, 46
análisis de componentes principales, 43
análisis del flujo de control, 36–38
análisis dinámico del flujo de información, 45
aplicaciones, 160
aprendizaje automático, 33, 45, 121–153
  de clase única, 121, 138–141
  supervisado, 121–137
ASL, 43
autómata auto-replicante, 4

base de datos de malware, 82–84
bases de datos léxicas, 163
boosting, 46
bosque aleatorio, *véase* random forest
Bulgarian Virus Producing Factory, 6

C4.5, 124–126
certificación de software, 166
checksumming, *véase* resumen criptográfico
Chernobyl, 29
CIH, *véase* Chernobyl
clasificador bayesiano ingenuo, 46, *véase* Naïve Bayes
clustering, 87
CLUTO, 87
Code Surfer, 37
Common N-gram, 46
comparación de patrones, 34–35
Computation Tree Predicate Logic (CTPL), 37
conclusiones, 155
contribuciones, 157
Control Flow Graph Analysis (CFG), *véase* análisis del flujo de control
cross validation, 143
CTL, 37
CWSandbox, 39

decisión tree, *véase* árbol de decisión
dendrograma, 87, 111, 112
desempaquetado de ejecutables, 49–52
detección de malware
  basada en el comportamiento, 28
  basada en firmas, 33
  clasificación, 28, 31–34
    clasificación de Idika et ál., 31

clasificación en base a objetivo, 32
cobertura de variantes, 80
detección basada en anomalías, 32, 34, 41–45
detección basada en especificaciones, 32
detección basada en firmas, 32
detección de malware desconocido, 8, 33, 41–48
detección basada en anomalías, 34
detección basada en aprendizaje automático, 33
detección de variantes basada en el conocimiento, 7, 33–41
detección dinámica, 38–40
detección estática, 34–38
detección híbrida, 40
comparación de patrones, 34
llamadas a la API, *véase* syscalls
detección dinámica, 32, 43, 47
detección estática, 32, 34, 41
detección híbrida, 32, 45
detección sintáctica, 28
detector de malware, 28
eficiencia, 9
etiquetado específico, 80
método clásico de firmas, 3, 17, 18, 28–31, 46, 52
método propuesto
detección de malware desconocido, 121
detección de variantes, 119
retos a superar, 7–9
detección de malware , 27
distancia de Hamming, 44
distancia euclidiana, 132
distancia Mahalanobis, 44
Dynamic Information Flow Analysis (DIFA), *véase* análisis dinámico del flujo de información

empaquetador, 26, 49
entorno controlado, 38
entry point, 50
Ether, 39
Eureka, 51
exploit, 45

f-measure, 142
False Positive Ratio (FPR), *véase* tasa de falsos positivos
falso negativo, 8
falso positivo, 8, 38, 49, 81, 91
filogenética, 85
formalización teórica del malware, 48

generación automática de firmas, 52–53
generador de consultas, 82, 84
generador de modelos filogenéticos, 82, 85–87
gusano de Morris, 4

hiperplano, 127

IDA Pro, 37
indecidibilidad, 48
Intel VT, 39

k-nearest neighbour (KNN), 46, 47, 132
K2, 135–136

lógica de predicados de arboles de computación, *véase* Computation Tree Predicate Logic (CTPL)

lenguaje máquina, 56
limitaciones, 162
llamadas a la API, *véase* syscalls
llamadas al sistema, *véase* syscalls

método propuesto
  detección de malware desconocido, 153
  detección de variantes, 79
  representación de ejecutables, 55–78
machine learning, *véase* aprendizaje automático
malware, 3, 18
  cambios en la motivación, 6
  definición, 18
  metamorfismo, 26
  polimorfismo, 26
  tipos de malware, 18–21
    backdoor, 20
    bomba lógica, 21
    bot, 21
    botnet, 21
    caballo de Troya, 19
    gusano, 19
    puerta trasera, 20
    spyware, 20
    troyano, 19
    virus, 19, 48
McBoost, 40
metamorfismo, 5, 26, 31
metodología de investigación, 12
  investigación-acción, 13
Microsoft Authenticode, 43
modelo filogenético, 80, 82, 85–88
modelos filogenéticos, 86, 112

n-gramas, 42, 46, 47
Naïve Bayes, *véase* clasificador bayesiano ingenuo, 136–137

ofuscación de software, 3, 21–25
  conceptos básicos, 22
  coste, 23
  métodos, 24–25
    de agregación, 24
    de almacenamiento y codificación, 24
    de capa, 24
    de control, 25
    de orden, 25
  potencia, 22, 23
  predicados opacos, 25
  resistencia, 23
  sigilo, 24
OllyBonE, 50
OmniUnpack, 51
opcodes, 56, 57, 59–63, 67–71, 76–78
  capacidades de los opcodes, 56–63

packer, *véase* empaquetador
PAYL, 44
PEiD, 84, 118
polimorfismo, 5, 26, 31, 49
PolyUnpack, 50
precisión, 142
Principal Component Analysis (PCA), *véase* análisis de componentes principales

QEMU, 39, 47

random forest, 126–127
recuperación de información, 15, 63
  definición, 64
recuperación de variantes de malware, 82, 85
Redes bayesianas, 133–137

redes sociales online, 165
Renovo, 51
representación de ejecutables, 9, 157
    evaluación de la eficiencia, 71–77
    modelo basado en recuperación de información, 63–70, 81
resumen criptográfico, 42
resumen de los resultados, 157
Ripper, 46
Roc-SVM, 138–141

sandbox, *véase* entorno controlado
selección jerárquica de características, 46
SIGC, 44
SIGE, 44
SIGT, 44
smart scanning, 35
spam, 164
Static Analyser for Executables (SAFE), 37
Static Analysis for Vicious Executables (SAVE), 35
stop word removal, 163
string scanning, 34
Support Vector Machines (SVM), 127–131
    funciones de kernel, 130–131
syscalls, 35–36

tasa de falsos positivos, 142
tasa de positivos verdaderos, 142
tendencia evolutiva, 86
teorema de Bayes, 46
Topic Detection and Tracking, TDT, 164
trabajo futuro, 162
    ampliación del sistema de detección, 164
    defender los medios actuales de propagación de malware, 164
    representación de ejecutables, 162
Tree Augmented Naïve (TAN), 137
True Positive Ratio (TPR), *véase* tasa de positivos verdaderos
TTAnalyze, 39

Universal PE Unpacker, 50
Unweighted Pair Group Method with Arithmetic mean (UPGMA), 87

verificación de software, 43

wildcards, 35
WINE, 47

Esta tesis doctoral se terminó de escribir
en Bilbao el 1 de Junio de 2011.