Universidad Zaragoza

Trabajo Fin de Grado en Ingeniería Informática

Evaluación de algoritmos de fuzzy hashing para similitud entre procesos

Iñaki Abadía Osta

Director: Ricardo J. Rodríguez

Ponente: José Merseguer Hernáiz

Departamento de Informática e Ingeniería de Sistemas
Escuela de Ingeniería y Arquitectura
Universidad de Zaragoza

Septiembre de 2017
Curso 2016/2017

Agradecimientos

Dedicar este trabajo de fin de grado
a mis padres por escucharme siempre que lo he necesitado,
a Inés por acompañarme a lo largo de todo el camino,
a Ricardo por su dedicación a este trabajo y todo el apoyo.

Evaluación de algoritmos de fuzzy hashing para similitud entre procesos

RESUMEN

Una función de *hash* criptográfico recibe una entrada de datos y genera una cadena de caracteres de tamaño fijo. Este tipo de funciones son idóneas para sistemas que confían en la criptografía para satisfacer aspectos de seguridad como la integridad de los datos. Las funciones de hash criptográfico se apoyan en el efecto cascada, que provoca que un pequeño cambio en la entrada genere un desajuste en la salida. Por tanto, estas funciones sirven para detectar alteraciones indeseadas en la entrada de datos.

Este trabajo se centra en un subconjunto de funciones de hash conocidas como funciones o algoritmos de *fuzzy hash*, que tratan de evitar este efecto cascada. En su lugar, satisfacen que un cambio (no demasiado grande) en la entrada se verá reflejado en la salida, aunque guardará cierto porcentaje de similitud con el hash de la entrada original. Este tipo de funciones son comúnmente utilizadas en el ámbito de Internet para identificar plagios o como sistema de detección de spam. Se van a considerar 4 familias o tipos distintos de algoritmos de fuzzy hashing: Context Triggered Piecewise Hashing, Statistically-Improbable Features, Block-Based Rebuilding, Block-Based Hashing y Locality-Sensitive Hashing.

En este trabajo se va a desarrollar una herramienta integrada en el framework de análisis de memoria forense Volatility con el fin de evaluar la utilidad de estas funciones en la detección de similitudes entre procesos. Este área de estudio es muy importante a la hora de discernir si procesos en ejecución de una máquina supuestamente comprometida son legítimos o no.

Se van a evaluar procesos de sistema, software de usuario y malware en máquinas Windows 7 y Windows 10 de 32 y 64 bits. Los experimentos realizados muestran que el algoritmo de la familia Block-Based Hashing considerado (`dcfldd`) obtiene los mejores resultados a la hora de localizar similitudes entre procesos. Por último, conviene destacar que los resultados de la muestra de malware analizada no son tan buenos como en el resto de software evaluado.

Índice

Índice de Figuras

Capítulo 1

Introducción

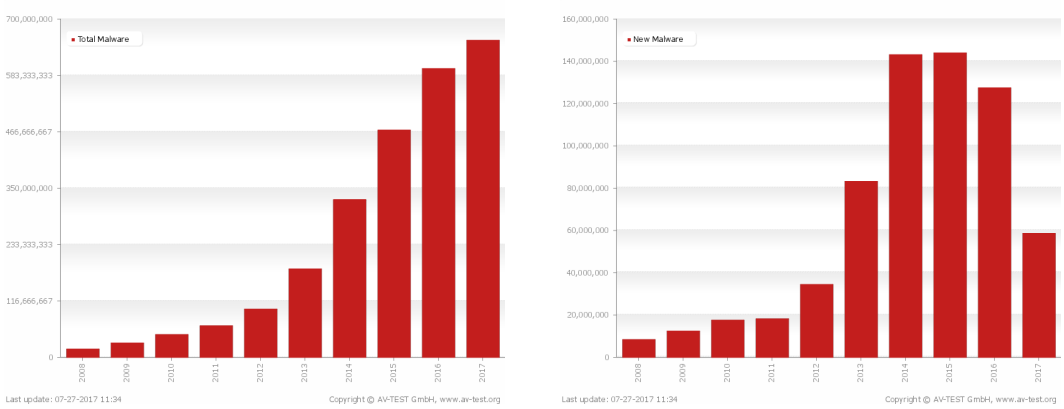
El análisis y detección de software malicioso, o *malware*, son tareas de vital importancia como medidas de contingencia frente a la gran cantidad de malware que circula diariamente. En los últimos años, estas cifras se han disparado hasta llegar a un crecimiento exponencial y preocupante. Sin embargo, durante el último año, según datos de AV-TEST [AT17a], este crecimiento se ha ralentizado: ya no aparecen tantas nuevas familias de malware, sino que se generan gran cantidad de variaciones de las familias ya existentes. Estos datos pueden observarse en la Figura 1.1.

Esto plantea una oportunidad para mejorar la detección de malware, ya que aquellas diferentes piezas de malware de la misma familia compartirán gran cantidad de código y/o datos característicos. Para evitar este hecho, los desarrolladores de malware utilizan técnicas de ofuscación, empaquetado y polimorfismo, entre otras [BM08, YZA08, CJ03]. No obstante, estas prácticas no solo están destinadas a este tipo de software, ya que también son usadas como protección contra la piratería por software benigno [CT02]. Mediante estas técnicas se logra que incluso dos malware idénticos sean completamente distintos, al menos, en lo que se refiere a su código binario.

Las funciones de hash son utilizadas como primera medida de detección en el análisis de malware. Se emplean, por ejemplo, para calcular un identificador del contenido binario del malware junto con el resto de sus indicadores de compromiso [Mar14]. Estas funciones o algoritmos de hash criptográficos (e.g., `MD5`, `SHA-1` o `SHA-256`, entre otros) reciben una entrada de datos y generan una cadena de caracteres de tamaño fijo. Este tipo de funciones de hash son idóneas para aplicaciones que confían en la criptografía para asegurar ciertos aspectos de seguridad como la integridad de los datos. Se apoyan en el efecto cascada, una propiedad que asegura que un cambio en la entrada del algoritmo genera un desajuste en la salida.

La problemática de utilizar este tipo de algoritmos de hash para identificar malware, es que este efecto cascada provoca que dos malware idénticos, salvo por una leve diferencia, generan un hash completamente diferente. Este hecho puede ilustrarse con un ejemplo: dado el primer capítulo del Quijote de Cervantes [dCS99] como entrada, su hash `MD5` (hash criptográfico) es:

```
401c7eaecc51b3c425d99431cb494865
```



(a) Malware total en los últimos 10 años. Imagen de [AT17c]

(b) Malware nuevo en los últimos 10 años. Imagen de [AT17b]

Figura 1.1: Estadísticas sobre malware durante los últimos 10 años.

Sustituido el primer carácter de dicho capítulo por una X, se obtiene el siguiente hash `MD5`:

```
8002ae5ea34a733ebcd14c4173e7fb55
```

Puede observarse a simple vista que los dos hashes no guardan ningún parecido.

Sabiendo esto, es posible deducir que aquellos malware que utilicen técnicas de ofuscación o empaquetado serán imposibles de identificar correctamente mediante un hash criptográfico. Además, una de las primeras fases de la respuesta a incidentes [SS13] es obtener un volcado de memoria de la máquina comprometida, para después ser analizado con herramientas forenses como Volatility. Este trabajo se va a centrar en Windows dado que este sistema operativo es el objetivo de la mayor parte de malware que circula: hasta un 85 % del malware total detectado en 2015, según [Pur16]. Debido al modo de funcionamiento de Windows [RSI12], un proceso en ejecución no será idéntico al fichero ejecutable que se encuentra en disco y por tanto, de nuevo, sus hashes serán diferentes (en el Capítulo 2 se detalla por qué sucede esto).

Existe otra familia de algoritmos de hash que evitan el efecto cascada, llamadas funciones de *fuzzy hash*. Estas funciones, al contrario que las anteriores, satisfacen que un cambio (no demasiado grande) en la entrada se refleja en la salida, aunque sigue guardando cierto porcentaje de similitud con el hash original. Este hecho puede observarse de nuevo con el ejemplo anterior del primer capítulo del Quijote de Cervantes [dCS99]: si se calcula su hash `ssdeep` (una opción tipo fuzzy hash), se obtiene

```
96:24DPjpT6IJv20LW9C4KPGh+ElX4unmT5vC1iO+PMtgLjEibzH8VhCJ:ZjdvZLWVKPG+Elokg5UinPRLYibzIhCJ
```

Alterado de la misma forma la entrada que con el hash `MD5`, se obtiene

```
96:CqVDPjpT6IJv20LW9C4KPGh+ElX4unmT5vC1iO+PMtgLjEibzH8VhCJ:NjdvZLWVKPG+Elokg5UinPRLYibzIhCJ
```

Puede observarse que mientas que los hashes `MD5` difieren completamente, los hashes `ssdeep` guardan un porcentaje de similitud de un 97 %. Por tanto, es de suponer que haciendo uso de estas funciones, el fuzzy hash de un proceso obtenido de la memoria y de su correspondiente ejecutable en disco guardarán un gran parecido, así como dos procesos de diferentes ejecuciones del mismo ejecutable. Esta es la hipótesis de trabajo que se pretende contrastar en este trabajo.

1.1. Objetivo

Los objetivos de este TFG son: a) Diseñar e implementar una herramienta llamada ProcessFuzzyHash en forma de plugin de Volatility, que permita calcular y comparar fuzzy hashes de procesos extraídos de un volcado de memoria; b) Realizar un estudio y evaluación de diferentes algoritmos de fuzzy hashing para detección de similitudes entre procesos.

El proceso llevado a cabo para lograr estos objetivos será:

- Estudio del estado del arte en el ámbito de algoritmos para fuzzy hashing, así como estudio de otras soluciones ya propuestas que satisfagan parcial o completamente los objetivos anteriormente propuestos.
- Estudio del funcionamiento del framework de análisis forense de memoria Volatility.
- Diseño e implementación de la herramienta, en forma de plugin de Volatility para detección de similitudes entre procesos de un mismo ejecutable.
- Diseño de experimentos, definiendo un exhaustivo banco de pruebas que incluirá distintas ejecuciones de un mismo fichero ejecutable, sea software malicioso o benigno, en distintas versiones del sistema operativo Windows.
- Evaluación de los algoritmos implementados en la herramienta mediante el banco de pruebas diseñado.

1.2. Motivación

La gran cantidad de malware que circula diariamente acentúa la necesidad de acelerar los procesos actuales de análisis y contingencia. Una de las primeras fases de la respuesta a incidentes cuando una máquina es comprometida, consiste en obtener un volcado de memoria para analizarlo. En dicho volcado se encuentran, entre otras cosas, todos los procesos que en el momento de la captura se encontraban en ejecución. La posibilidad de comparar los fuzzy hashes de tales procesos con, por ejemplo, una base de datos de hashes conocidos, tanto de malware como de procesos legítimos, permite acelerar un primer diagnóstico sobre el estado de la máquina.

Mediante el desarrollo de esta herramienta para evaluar diferentes algoritmos de fuzzy hash se pretende comprobar si el uso de este tipo de hashes es una buena técnica para

detectar similitudes entre procesos. Por otro lado, se espera que la herramienta desarrollada sea de ayuda a la comunidad de analistas forenses. Además, como producto secundario de este trabajo se ha realizado una guía para el desarrollo de plugins de Volatility, inexistente hasta el momento, que puede resultar útil a otros que quieran aprender a desarrollar plugins para Volatility [Aba17].

1.3. Organización

La estructura de este documento está pensada para facilitar una lectura continua que permita al lector avanzar sin necesidad de retroceder o consultar otras secciones para entender lo que se está explicando. Se encuentra dividido en seis capítulos y tres enexos, cuya lectura no es necesaria pero sí recomendada.

El Capítulo 2 introduce algunos conceptos útiles para facilitar la comprensión del resto del documento: descripción de los algoritmos de fuzzy hashing considerados inicialmente, estructura de ejecutables Windows, procesos de Windows y el framework Volatility. A continuación, el Capítulo 3, recoge los apartados de diseño, implementación y uso de la herramienta desarrollada. El Capítulo 4 describe la experimentación realizada y los resultados obtenidos. El Capítulo 5 recoge una lista de propuestas o alternativas ya existentes a la aquí planteada. Por último, el Capítulo 6 contiene las conclusiones y líneas futuras de este trabajo.

Al final del documento se encuentran los anexos, que amplían la información de este trabajo:

- **Anexo A:** En este anexo se encuentra un desglose detallado de en qué se ha invertido el tiempo de trabajo.
- **Anexo B:** Contiene un esqueleto de código para desarrollar un plugin de Volatility. Se trata de un ejemplo sencillo con los componentes mínimos necesarios para un plugin.

Capítulo 2

Conocimientos previos

En este capítulo se va realizar un listado y descripción de los algoritmos de fuzzy hash inicialmente barajados para su implementación en la herramienta. Además, se repasará la estructura de los ejecutables y procesos en Windows, así como del framework Volatility.

2.1. Algoritmos Fuzzy Hash

Antes de empezar se debe conocer la diferencia entre un fuzzy hash y un hash criptográfico, habitualmente referidos como hash a secas. Como ya se ha comentado anteriormente, un algoritmo de fuzzy hash, al contrario que un algoritmo de hash criptográfico, no sufre el llamado efecto cascada que provoca que un pequeño cambio en la entrada del algoritmo produzca un desajuste en la salida. En su lugar, busca garantizar que un pequeño cambio en la entrada del algoritmo genere un hash distinto pero manteniendo cierto porcentaje de similitud con el hash de la entrada original. De esta forma, en el fuzzy hash de varias entradas con pequeñas diferencias entre ellas se podrá observar pequeños cambios en la salida pero a su vez, también identificar las partes idénticas como se ha ilustrado con el ejemplo del Capítulo 1.

Existen distintos tipos o formas de generar fuzzy hashes. En este trabajo se van a tratar cinco vertientes extraídas de [MAE14]: *Context Triggered Piecewise Hashing* (CTPH), *Statistically-Improbable Features* (SIF), *Block-Based Rebuilding* (BBR), *Block-Based Hashing* (BBH) y *Locality-Sensitive Hashing* (LSH).

Tras un estudio del estado del arte en algoritmos de fuzzy hashing se eligieron aquellos que mejor encajan en el objetivo perseguido, se clasificaron en cada una de sus correspondientes familias y se eliminaron aquellos de carácter teórico o que carecían de implementación. A continuación, se describe brevemente cada tipo y los algoritmos de cada uno considerados para su posterior inclusión en la herramienta.

2.1.1. Context Triggered Piecewise Hashing

Aquellos algoritmos pertenecientes a esta familia generalmente utilizan una técnica basada en el cálculo y combinación de dos hashes distintos: uno tradicional (e.g., MD5,

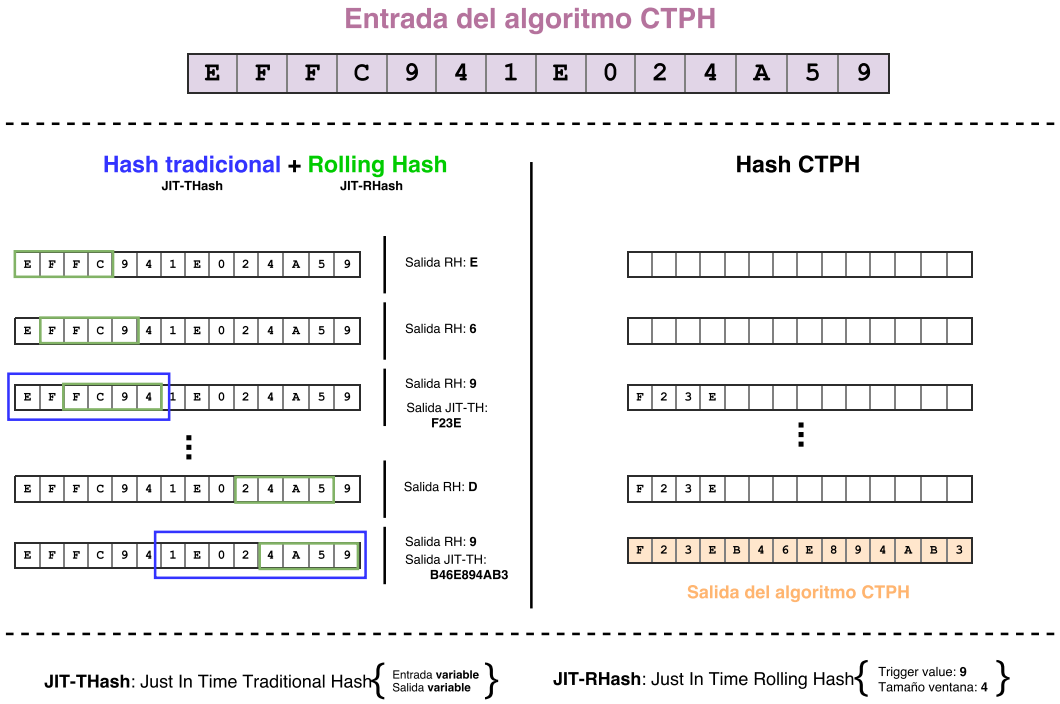


Figura 2.1: Funcionamiento de un algoritmo CTPH.

SHA1) y uno denominado *rolling hash*. En la Figura 2.1 puede observarse el funcionamiento de un algoritmo CTPH. La salida del *rolling hash* depende de la entrada del mismo, dada por una ventana deslizante sobre la entrada del hash CTPH, y un estado compuesto por unos pocos bytes de anteriores entradas. La forma de combinar estos dos algoritmos es la siguiente: cada vez que el rolling hash genera una salida específica (llamada *trigger value*), se genera y se añade al hash final el calculado por el algoritmo tradicional. La mayoría de algoritmos CTPH realizan la comparación entre hashes calculando la distancia Hamming, que calcula el número de posiciones en las que difieren dos cadenas de caracteres. En otras palabras, es el mínimo número de sustituciones requeridas para transformar una cadena en la otra.

Para este trabajo se estudiaron los algoritmos `ssdeep`, `FKSum` y `MRSHv1/2`. A continuación, se describen brevemente:

`ssdeep`. El algoritmo `ssdeep` fue propuesto en 2006 [Kor06]. Está basado en *spamsum* [Tri02], una de las primeras soluciones CTPH para detección de spam. El autor compara las prestaciones de `ssdeep` con algoritmos criptográficos como MD5 y SHA256, obteniendo peores resultados de rendimiento pero destacando que el uso de algoritmos CTPH en la informática forense es algo nuevo y que podría ayudar a analistas a localizar ficheros que, de otro modo, se habrían perdido entre grandes cantidades de datos. Los hashes generados por este algoritmo varían de tamaño según la entrada.

`MRSH v1/2`. La primera versión fue publicada en 2007 [RRM07]. Este algoritmo se basa en una versión de `ssdeep` con varios cambios que afectan al algoritmo tradicional utilizado, el tamaño del hash generado y la adición de un filtro Bloom para generar el hash final. (Un filtro Bloom es una estructura de datos probabilística diseñada para ser eficiente en tamaño y usada para comprobar si un elemento pertenece a un conjunto.) La segunda versión, propuesta en [BB13], trata de mejorar sus prestaciones. El autor no revela información sobre el tamaño exacto de los hashes generados ni la forma en la que representa la ratio de comparación entre hashes.

`FKSum`. Este algoritmo fue propuesto en 2008 [CW08]. Como `ssdeep`, `FKSum` está basado en `spamsum`, aunque dista un poco más de la técnica que éste utiliza. Utiliza dos funciones de hash tradicional en lugar de combinar un hash tradicional con un rolling hash. El principal aporte de `FKSum` a la familia CTPH es que además de generar un hash tradicional con cada trigger value, calcula varios hashes intermedios con el segundo algoritmo de hash tradicional, reduciendo así el número de iteraciones sobre la entrada. Por tanto, resulta más eficiente que otros algoritmos CTPH como `spamsum`. En [CW08] se encuentra una descripción más detallada de todo el proceso y tratamiento de los datos obtenidos en cada iteración. Cabe mencionar que los autores realizan una comparación en prestaciones con `spamsum`, pero no con `ssdeep`, y una comparación de resultados de similitud con `ssdeep`, pero no con `spamsum`. El autor no menciona las características de los hashes generados para el estudio pero, al estar basado en `spamsum`, es bastante acertado asumir que son de longitud variable. Por último, conviene destacar que el código fuente de `FKSum` no está disponible.

2.1.2. Statistically-Improbable Features

Este tipo de funciones o algoritmos tratan de localizar un conjunto de características compuestas por secuencias de bits poco habituales. De esta forma, la comparación entre hashes consiste en hallar similitudes entre estos conjuntos de características poco habituales. En la Figura 2.2 puede observarse de manera muy simplificada el funcionamiento de un algoritmo SIF muy simplificado.

A continuación, se describen brevemente los algoritmos estudiados para este trabajo: `SDHash` y `SimHash`.

`SimHash`. Propuesto inicialmente en 2002 [Cha02] y más tarde revisado en 2007 [MJDS07], este algoritmo es algo diferente del resto ya comentados, puesto que precisa como entrada una lista de características ya extraídas con unos pesos dados. El hash resultante se calcula utilizando los hashes de cada una de las características tomadas en la entrada y un vector de 64 posiciones inicializado a cero de tamaño igual al número de bits de estos hashes. Por cada bit en el hash de una característica se incrementará, si es 1, o decrementará, si es 0, la correspondiente posición en el vector. El valor a incrementar o decrementar será el peso asignado a la característica. El signo de cada posición del vector resultante determinará el valor del hash final.

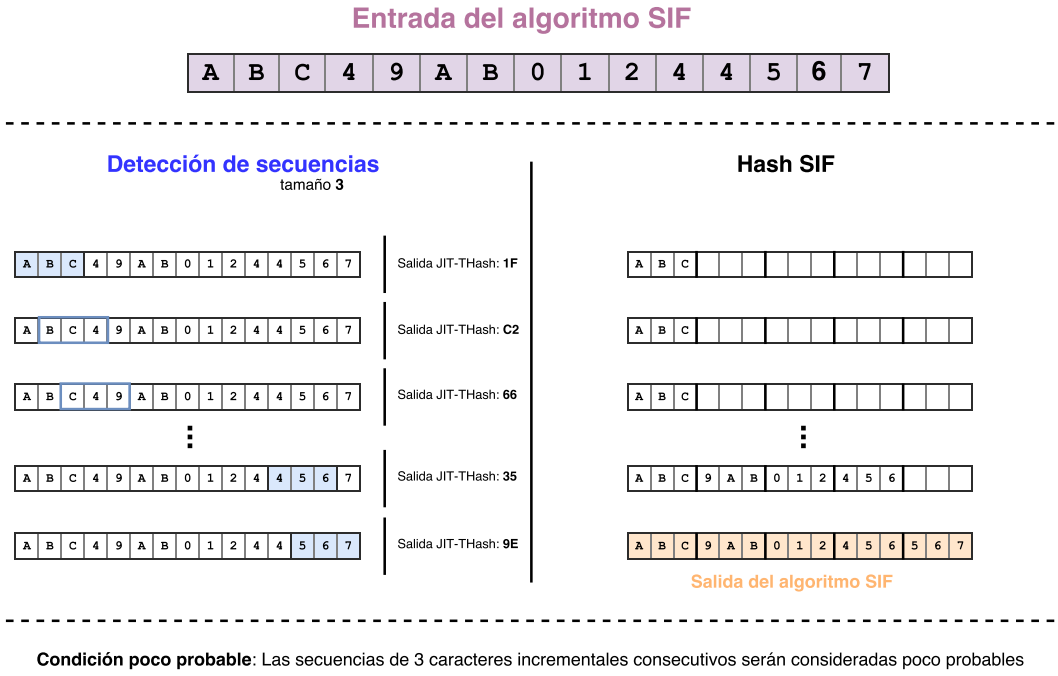


Figura 2.2: Funcionamiento de un algoritmo SIF.

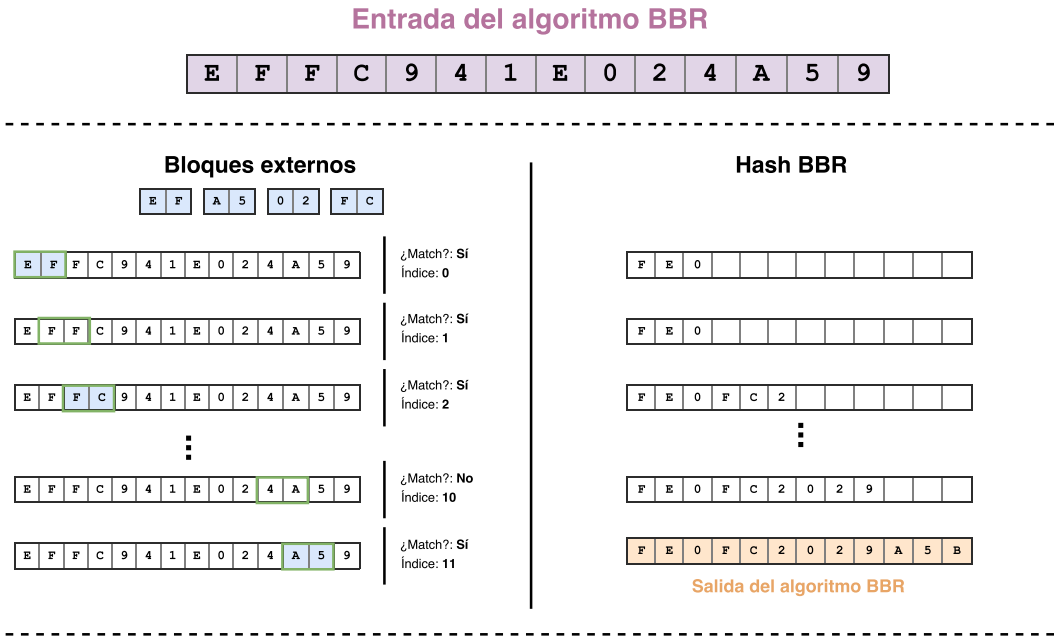
`SDHash`. Propuesto en 2009 [Rou09], este algoritmo localiza características poco probables realizando un cálculo de la entropía de la entrada. Estas características son posteriormente compuestas en una cadena de 64 bytes que es tratada con la función hash criptográfica SHA-1 para después ser pasada por un filtro Bloom. Por tanto, las entradas se considerarán similares si comparten estas características. Este algoritmo genera hashes de longitud variable, entre un 2.6 % y 3.3 % del tamaño de la entrada.

2.1.3. Block-Based Rebuilding

Esta familia de algoritmos o funciones hace uso de datos externos a la propia entrada del algoritmo. Estos datos pueden obtenerse aleatoriamente o ser siempre los mismos. En la Figura 2.3 puede observarse el funcionamiento de un algoritmo tipo BBR. La forma de comparar hashes en esta familia depende de cada algoritmo, pero en general se realiza mediante el cálculo de distancia Hamming.

Para este trabajo se estudiaron los algoritmos `SimiHash` y `bbHash`, descritos a continuación.

`SimiHash`. Publicado en 2007 [SL07] como SimHash, más tarde vuelto a publicar rebautizado como `SimiHash` en un informe técnico [SL11] de 2011. Esta solución preselecciona 16 bloques de 1 byte cada uno, busca estos mismos bloques en la entrada del algoritmo y cuando encuentra alguno lo almacena en una tabla. Al acabar, se procesa esta tabla y se obtiene la salida del algoritmo. `SimiHash` es habitualmente confundido, debido a



Condición de Match: Cuando el bloque tratado coincida con un bloque externo, dicho bloque y su índice serán añadidos al hash resultante.
Generación de Bloques externos: Los bloques externos son calculados aleatoriamente.

Figura 2.3: Funcionamiento de un algoritmo BBR.

su antiguo nombre, con el algoritmo `SimHash`, de tipo SIF explicado previamente. Los autores de `SimiHash` decidieron desambiguar y fue por ello que lo rebautizaron en 2011. El autor no menciona explícitamente el tamaño de los hashes generados, pero sí que apunta que depende de la entrada y de la configuración del algoritmo.

`bbHash`. Ideado en 2012 [BB12], utiliza un conjunto fijo de un número arbitrario de secuencias de 128 bytes aleatorios llamados *building blocks*. El proceso consiste en recorrer byte a byte la entrada del algoritmo y calcular la distancia Hamming de todos los building blocks con la secuencia de bytes tratada en cada paso. Si el building block que menos distancia Hamming ha logrado en cada paso es inferior a un límite determinado, el índice de este building block contribuye a la salida del algoritmo. Uno de los mayores inconvenientes de esta solución es la gran cantidad de recursos y tiempo que consume. La longitud de los hashes generados se encuentra en torno a un 0.5 % del tamaño de la entrada.

2.1.4. Block-Based Hashing

Esta familia de algoritmos realiza un proceso más sencillo. En concreto, generan hashes criptográficos de bloques de tamaño fijo y realizan la comparación entre los hashes de cada bloque. En la Figura 2.4 puede observarse de manera simplificada el funcionamiento de un algoritmo BBH. El porcentaje de similitud se basará, por tanto, en cuántos hashes de bloques idénticos comparten.

Para este trabajo se ha estudiado sólo el algoritmo `dcfldd`, descrito a continuación.

`dcfldd`. Esta herramienta fue desarrollada en 2002 [Def09]. Se trata de una versión mejorada de la herramienta de GNU *dd*, que permite convertir y realizar copias de datos byte a byte. `Dcfldd` divide la entrada en bloques de tamaño fijo y les aplica un hash criptográfico individualmente. Es muy eficiente en tiempo, pero muy sensible a pequeñas modificaciones: un diminuto cambio al principio de la entrada puede afectar a todos los hashes generados. El tamaño del hash generado depende directamente del tamaño de la entrada y de los bloques en que se divide. La evaluación de similitud entre hashes se realiza comparando uno a uno los hashes de cada bloque.

2.1.5. Locality-Sensitive Hashing

Esta familia de algoritmos trata de reducir el inmenso espacio de posibles entradas a una cantidad discreta de posibilidades, maximizando la probabilidad de que dos entradas similares generen una salida casi idéntica. Las técnicas utilizadas en estos algoritmos están relacionados con el problema de la búsqueda del vecino más cercano. En la Figura 2.5 puede observarse el funcionamiento simplificado de un algoritmo LSH. El cálculo de similitud se realiza habitualmente mediante el cálculo de la distancia Hamming.

En esta sección se van a describir los algoritmos elegidos para este trabajo: `Nilsimsa` y `TLSH`.

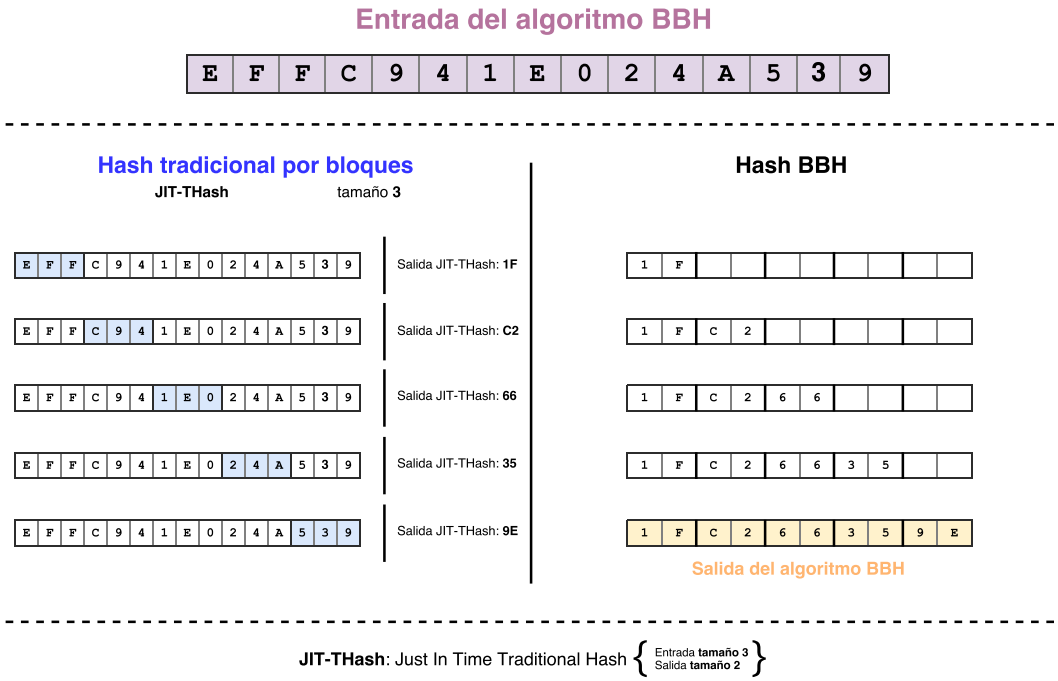


Figura 2.4: Funcionamiento de un algoritmo BBH.

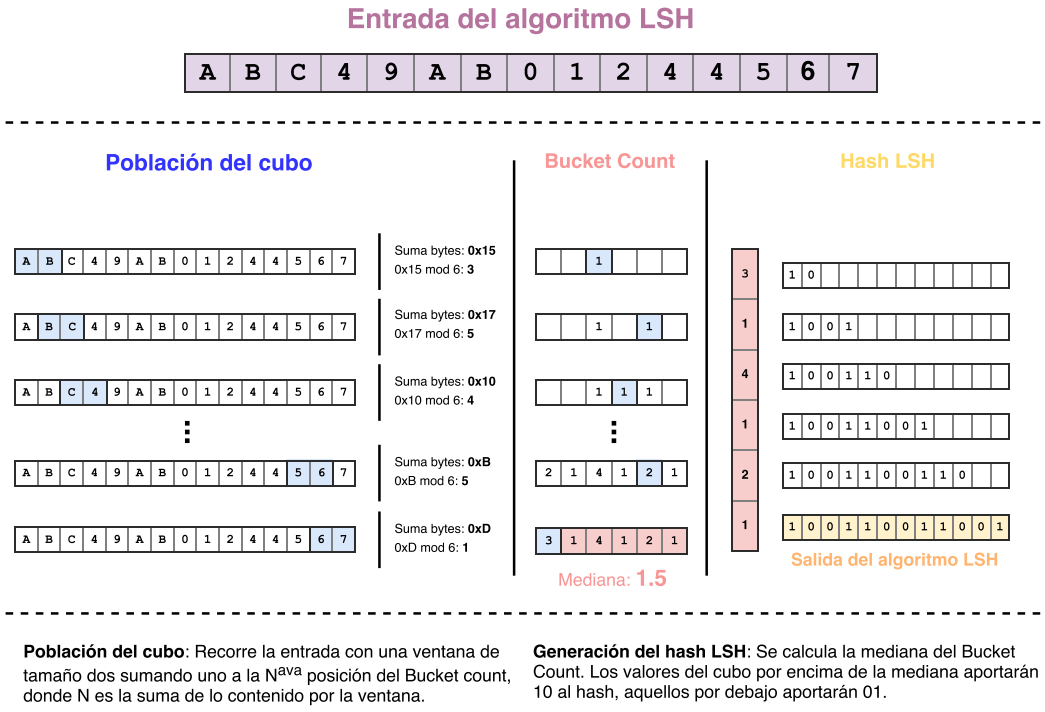


Figura 2.5: Funcionamiento de un algoritmo LSH.

`Nilsimsa.` Se trata de una solución anti-spam del 2001, revisada posteriormente en 2004 [CDD+04]. Este algoritmo está bastante extendido dado que fue una de las primeras soluciones LSH para encontrar similitudes. También es conocido por generar una gran cantidad de falsos positivos. El hash generado por `Nilsimsa` es una cadena de 64 caracteres hexadecimales.

`TLSH.` Desarrollado en 2015 [Mic15] y diseñado para uso en el ámbito de la seguridad e informática forense. Requiere una entrada de al menos 256 bytes y con un mínimo de aleatoriedad, ya que una entrada de bytes idénticos no generaría ningún hash. Genera una salida de tamaño fijo 35 bytes, representado como una cadena de 70 caracteres hexadecimales. Estos 35 bytes se componen de 3 bytes dedicados a almacenar la naturaleza de la entrada (longitud, etc.) y los 32 bytes restantes componen el hash calculado de la entrada. La ratio de similitud comprende un rango indeterminado de valores positivos: 0 indica equivalencia y valores más altos indican diferencia. El autor remarca que aunque los resultados muestran que la mayoría de hashes no pasan del valor 300, algunos superan los 1000.

2.1.6. Otros tipos de algoritmos

También merece la pena destacar otras soluciones interesantes no consideradas en este trabajo.

`peHash.` Publicado en 2008 [Wic08], este algoritmo está enfocado al análisis de ficheros ejecutables del sistema operativo Windows. Genera una salida basándose en la estructura de la cabecera del ejecutable, así como en lo contenido en otras partes de su código binario.

`Imphash.` Una solución propuesta en 2014 [MAN14]. Fue desarrollado basándose en el hecho de que aunque un malware sea polimórfico, tendrá las mismas dependencias de las bibliotecas de enlace dinámico (DLLs) [RSI12]. (Una DLL es un archivo que contiene código ejecutable, datos u otros recursos que son utilizados por diferentes programas.) Realiza un tratamiento del listado de estas bibliotecas convirtiendo nombres de DLLs a minúsculas o quitando extensiones de archivo, entre otros tratamientos. Finalmente, genera un hash MD5 de la lista resultante.

2.2. Ejecutables Windows

En Windows, los ficheros ejecutables (extensión `.EXE`) se denominan ficheros PE (Portable Executable)[Mica]. El formato PE es un estándar para empaquetar código ejecutable y, por tanto, también se utiliza en otros tipos de fichero como DLLs, ficheros OBJ (contienen código no específico de Windows), ficheros SCR (salvapantallas) o ficheros FON (contienen fuentes tipográficas), entre otros. La estructura de un fichero PE se compone de varias cabeceras, una tabla de secciones y un espacio donde se almacenan

Figura 2.6: Estructura básica de un fichero PE (Figura no escalada).

los datos contenidos en las secciones apuntadas por dicha tabla. En la Figura 2.6 se encuentra un diagrama con la estructura básica de un fichero PE de Windows (figura no escalada).

A continuación, se describe brevemente la utilidad de cada sección:

- `MS-DOS HEADER`. Esta cabecera se compone de una parte obligatoria llamada `IMAGE_DOS_HEADER` que ocupa los primeros 64 bytes, y de otra opcional llamada `DOS_STUB`. Se trata de una reminiscencia de `MS-DOS`, utilizada para que si un sistema operativo `MS-DOS` trataba de ejecutar un programa de Windows, pudiera al menos saber que no era capaz de hacerlo e incluso mostrar un mensaje indicándolo (contenido en el `DOS_STUB`, es un mensaje similar a *"This program cannot be run in DOS mode"*.).

- `IMAGE_NT_HEADERS`. Se compone de dos estructuras. La primera es `IMAGE_FILE_HEADER`, que contiene información sobre el propio fichero PE. La segunda estructura, a pesar de su nombre nombre, `IMAGE_OPTIONAL_HEADERS`, contiene datos necesarios para el correcto funcionamiento del ejecutable.

- `SECTION TABLE`. Contiene estructuras de tipo `IMAGE_SECTION_HEADER`, que almacenan información sobre todas las secciones en `SECTION_DATA` como el nombre, localización o tamaño.

- `SECTION DATA`. Es el espacio destinado a almacenar el contenido de las secciones apuntadas por la `SECTION TABLE`. Cada sección de un fichero ejecutable tiene una función diferente: almacenar el código binario, los datos de sólo lectura, datos de lectura y escritura, recursos del ejecutable, etc.

2.3. Procesos Windows

Un proceso en Windows, en esencia, es la representación de un fichero ejecutable (o programa) en ejecución. Debe destacarse la diferencia fundamental entre programa y un proceso: mientas que un programa es una secuencia estática de instrucciones, un proceso es un contenedor para todos los recursos usados durante la ejecución de una instancia de un programa. Haciendo una analogía con los conceptos de programación orientada a objetos, un programa sería una "clase", mientras que un proceso es un "objeto", es decir, una instancia de dicha clase.

A un nivel de abstracción muy alto, un proceso en Windows se compone de:

- **Espacio privado de direcciones virtuales**. Un conjunto de direcciones virtuales de memoria que el proceso puede usar libremente.
- **Ejecutable de un programa**. En él se encuentran definidos el código y datos iniciales que son mapeados en el espacio de direcciones virtuales del proceso.
- **Una lista de recursos de sistema**, usados por el proceso. Pueden ser semáforos, puertos, ficheros, etc.
- **Tokens de acceso**. Se trata del contexto de seguridad de Windows, almacenan toda la información de usuario propietario, grupos, privilegios, estado de la sesión, etc. asociados al proceso.
- **Un identificador único de proceso (process ID)**, es único a nivel de todo el sistema.
- **Al menos un hilo de ejecución**. Un proceso, en Windows, es un contenedor de threads (o hilos). Por tanto, puede contener varios hilos de ejecución. También se puede dar el caso de un proceso "vacío", aunque no presenta ninguna utilidad [RSI12].

Previamente a la ejecución de un proceso se realizan una serie de tareas fundamentales que pueden observarse en la Figura 2.7. Estos pasos se describen a continuación:

Fase 1. Validación de parámetros. Cuando se lanza la ejecución de un proceso pueden incluirse parámetros que, o bien le indiquen cómo comportarse, o le indiquen al sistema operativo cómo inicializarlo. Durante esta fase se realiza una comprobación y validación de dichos parámetros.

Fase 2. Carga el fichero ejecutable (PE) en memoria.

Fase 3. Crea la estructura de datos de sistema que representará el proceso.

Fase 4. Crea la estructura de datos de sistema que representará el hilo principal del proceso.

Fase 5. Realiza una inicialización, específica para cada sistema Windows, de las estructuras internas de proceso e hilo.

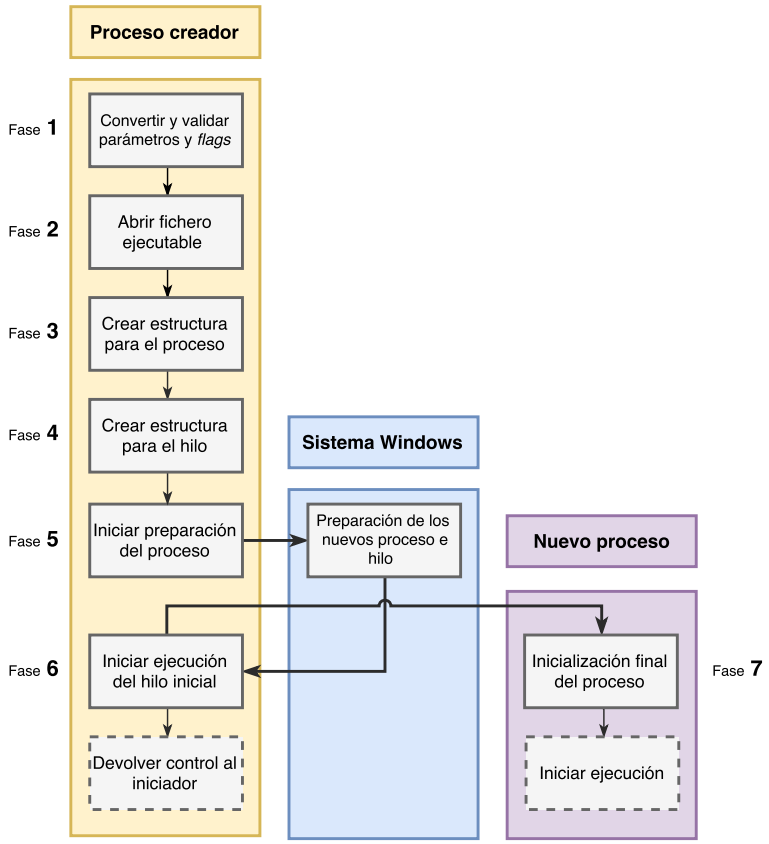


Figura 2.7: Proceso de creación de un nuevo proceso (figura adaptada de [RSI12]).

Fase 6. Inicia la ejecución del hilo principal del nuevo proceso.

Fase 7. Desde el espacio de direcciones del propio proceso, termina la inicialización. En esta fase se realiza la carga de bibliotecas de enlace dinámico (DLLs), entre otras.

La estructura de un proceso en memoria se compone de distintas secciones en que se encuentra dividido el espacio de direcciones del proceso [LCLW14]. En la Figura 2.8 puede observarse un diagrama de alto nivel de la distribución de la memoria de un proceso. Las más importantes son:

- **Bibliotecas de enlace dinámico (DLLs) [RSI12]**. Contiene las bibliotecas o librerías compartidas cargadas en el espacio de direcciones del proceso.
- **Variables de entorno**. Contiene variables de entorno como rutas completas, directorios temporales, carpetas locales, etc.
- **Bloque de entorno del proceso (PEB)**. Se trata de una estructura que contiene gran cantidad de información sobre el propio proceso, como dónde se encuentra

cada sección del espacio de direcciones, los argumentos con los que se ejecutó, o el directorio actual.

- **Heap del proceso**. Aquí se almacenan variables que usen memoria dinámica, como cadenas de texto de tamaño variable o búffers de datos recibidos por sockets, entre otros.
- **Pilas de los hilos del proceso**. Cada una corresponde a un hilo de ejecución del proceso. En arquitecturas de 32 bits, y en especial Intel x86, almacena argumentos de funciones, direcciones de retorno y variables locales de procedimientos o funciones, entre otros.
- **Ficheros y datos de aplicación**. Esta sección no tiene un fin tan concreto como el resto, su uso depende de cada proceso. Podrá contener ficheros de disco que necesite o archivos de configuración. Se consideran datos de aplicación a cualquier tipo de dato o estructura que el proceso precise para realizar su función.
- **Ejecutable (PE)**. Es la representación del ejecutable en disco, pero en memoria. Contiene la sección de código (normalmente denominada `.text`), así como variables de lectura/escritura (secciones `.data` o `.rdata`), y cualquier otra sección que tenga el fichero PE.

Es importante remarcar que dos procesos de un mismo ejecutable presentaran diferencias sustanciales. Esto se debe principalmente a que cuando Windows carga el ejecutable para comenzar su ejecución, es posible que no siempre use las mismas direcciones relativas y por tanto, el contenido del fichero, de los ficheros y datos asociados, o de las propias librerías cargadas junto con el binario se encuentren en posiciones diferentes. Este cambio en la posición de los bytes influirá en el cálculo de hash correspondiente. Una de las causas que influyen en esta diferencia es la protección Address Space Layout Randomization [BDS03, PaX], que provoca que las posiciones de memoria en las que se mapea cada librería cambie con cada inicio del sistema operativo Windows. Por último, también afecta el estado de ejecución en que se encuentra el proceso en el momento de la captura, ya que el contenido de la pila puede diferir.

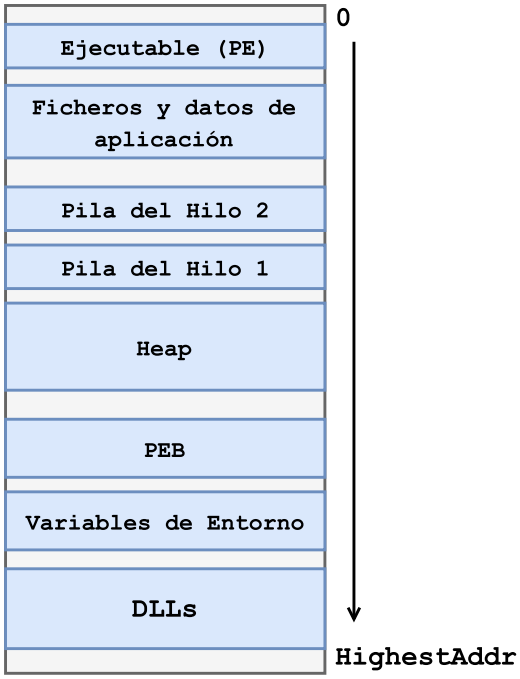


Figura 2.8: Diagrama de alto nivel del contenido habitual de un proceso (adaptado de [LCLW14], figura no escalada).

2.4. El framework Volatility

La primera versión de Volatility fue publicada en 2007 en la Black Hat DC[WP07] bajo el nombre de Volatools. Hasta ese momento, la tendencia en la informática forense era el análisis de memoria estática. Volatility introdujo a la comunidad de analistas el poder analizar el estado de la memoria en tiempo de ejecución, ya que permite el análisis de un volcado de memoria obtenido de una máquina en ejecución. Este framework ha ido creciendo y mejorando con muchos aportes de la comunidad hasta llegar a ser uno de los más importantes en el ámbito de la informática forense. Actualmente, *The Volatility Foundation*, una organización independiente sin ánimo de lucro, es la que se encarga del desarrollo de este software gratuito y de código abierto. Volatility está escrito en Python, lo que le da una versatilidad insuperable a través de todo tipo de plataformas.

Volatility se organiza como un framework que aglomera un conjunto de plugins, desarrollados tanto por The Volatility Foundation como por la comunidad. Cada comando de Volatility equivale a un plugin y muchos se apoyan en otros para realizar tareas comunes como listar procesos o recorrer estructuras del kernel de Windows. Por ejemplo, la herramienta desarrollada en este trabajo se apoya en *procdump*, otro plugin que extrae el fichero ejecutable de los procesos contenidos en un volcado de memoria (se explica en más detalle en el Capítulo 3).

La mayor barrera a superar a la hora de implementar un plugin para Volatility es la

ausencia de fuentes sobre cómo hacerlo. Sin embargo, al ser de código abierto es posible navegar por el código de Volatility y sus plugins, pudiendo observar aspectos comunes y deduciendo el mínimo común necesario. Entre esto y una pequeña guía [bri16] ha sido posible deducir este mínimo común necesario para implementar la herramienta ProcessFuzzyHash. Como resultado adicional a este trabajo, también se ha publicado una guía para la comunidad acerca del desarrollo de plugins con Volatility [Aba17].

En el Anexo B se encuentra el esqueleto de código de un plugin para Volatility que itera sobre la lista de procesos del volcado de memoria y los imprime por pantalla, junto a su PID y fecha de creación. También incluye una breve descripción de las partes más importantes incluidas en dicho esqueleto.

Capítulo 3

Herramienta ProcessFuzzyHash: diseño e implementación

Uno de los objetivos de este trabajo es desarrollar una herramienta que permita evaluar distintos algoritmos de fuzzy hashing a la hora de identificar similitudes entre procesos. Durante la fase previa de análisis se tuvieron en cuenta ciertos requisitos no funcionales a satisfacerse:

- **Se debe hacer uso del framework Volatility** dada la naturaleza de la herramienta: manejar procesos extraídos del volcado de memoria de una máquina. Existen otras soluciones que permitirían realizar el mismo trabajo, pero ninguna de ellas es tan especializada ni facilita tanto la tarea de recorrer un volcado de memoria.
- **La herramienta debe ser portable**, para así ejecutarse en el mayor número de sistemas posibles.

Dadas estas dos restricciones, se ha decidido utilizar el lenguaje de programación Python para la implementación.

En el Capítulo 2 se realiza una descripción de los algoritmos barajados para su inclusión en ProcessFuzzyHash. Dado que la implementación va a realizarse en Python y el objetivo de este trabajo no es implementar sino evaluar algoritmos, se ha seleccionado un algoritmo de cada tipo, considerando aquellos que (1) aporten la mayor diversidad y (2) preferiblemente cuenten con una implementación en Python. Los algoritmos seleccionados finalmente son los siguientes:

- `ssdeep`. Un algoritmo de la familia CTPH (véase Sección 2.1.1) que genera hashes de tamaño variable pero reducido.
- `SDHash`. De la familia SIF (véase Sección 2.1.2), genera hashes variables de considerable longitud.
- `Dcfldd`. Cuenta con la operativa más simple, de la familia BBH (véase Sección 2.1.4). Este algoritmo ha sido implementado en Python durante este trabajo dado

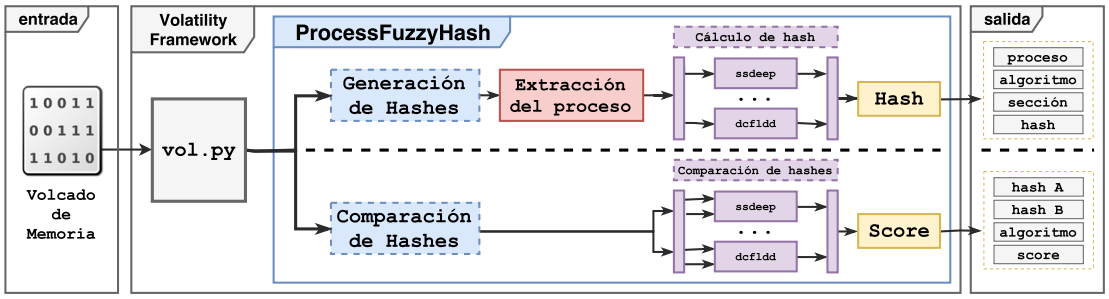


Figura 3.1: Diagrama de alto nivel de la operativa de ProcessFuzzyHash.

que aporta una nueva familia a las comparaciones y su complejidad es relativamente sencilla.

- `TLSH`. El único algoritmo contemplado de la familia LSH (véase Sección 2.1.5) que obtiene buenos resultados con ficheros binarios. Genera hashes de longitud fija, pero no es capaz de generar un hash si la entrada no cuenta con una entropía mínima.

Por otro lado, se han barajado dos soluciones para decidir de qué forma implementar la herramienta: un programa independiente que haga uso del módulo de Python Volatility, o en forma de plugin para dicho framework. Ambas cuentan con ventajas y desventajas pero, finalmente, se ha elegido implementarlo en forma de plugin dado que esto permitirá ejecutar la herramienta a través de Volatility y contribuir a la comunidad de usuarios y desarrolladores de Volatility.

3.1. Diseño e implementación

La Figura 3.1 contiene un diagrama de sistema de alto nivel de la operativa de la herramienta desarrollada. Se ha decidido separar la operativa de la herramienta en dos acciones independientes: la *generación de hashes* y la *comparación de hashes*.

Para la generación de hashes se espera un volcado de memoria, proporcionado por el usuario, que contiene los procesos de los que se desea obtener su hash. Del volcado de memoria se extraen los procesos indicados y se evalúa el resto de condiciones, también indicadas por el usuario. La parametrización de este modo de funcionamiento permite elegir de qué parte del proceso se desea obtener el hash. Se contemplan los siguientes casos:

- **Proceso completo.** Se obtiene un hash de todas las páginas de memoria usadas del espacio de direcciones del proceso. Esa opción es la más lenta de todas, dado que la cantidad de información a tratar puede superar en varios órdenes de magnitud a las demás.

- **PE.** En este caso se genera un hash del fichero PE cargado en memoria. Esta

opción resulta interesante ya que el PE almacena el contenido más característico de cada programa: su código binario.

- **Secciones del PE.** También se contempla la opción de generar un hash de tan sólo una sección del PE [Micb]. Con esta opción es posible extraer hashes de sólo la sección del código ejecutable o de la sección de datos de sólo lectura.

Para la comparación de hashes se precisa de, como mínimo, uno o más hashes y un algoritmo. El modo de comparación contempla dos casos:

- **Comparación con un fichero de hashes.** Esta opción compara los hashes proporcionados por el usuario con un fichero, también indicado por el usuario, que contiene un hash por línea. El resultado obtenido es la comparación uno a uno de los hashes proporcionados con cada uno de los incluidos en el fichero dado.

- **Comparación con hashes obtenidos de un volcado de memoria.** De esta forma, en lugar de proporcionar uno o más hashes y un fichero de hashes para comparar, se proporciona: un listado de hashes, un volcado de memoria y una configuración para extraer hashes de dicho volcado. El resultado obtenido es la comparación uno a uno del listado de hashes con los generados de los procesos extraídos del volcado de memoria. Esta supone una combinación de los modos extracción y comparación.

La Figura 3.2 muestra mediante un diagrama de clases los elementos implicados en el sistema, su utilidad y relación. Téngase en cuenta que, al tratarse de un plugin de Volatility, se han obviado todas aquellas funciones propias de un plugin dado que no aportan nada a la comprensión de la operativa de ProcessFuzzyHash.

Para realizar un buen diseño del sistema que cumpla con las características de modularidad e independencia, se ha utilizado el patrón de diseño *Facade* en `HashAlgorithm`. Esta clase abstracta define las funciones abstractas `hash` y `compare` implementadas por sus subclases. Dado que la herramienta utiliza una serie de módulos para manejar distintos algoritmos, este patrón permite reducir la complejidad del diseño ofreciendo una interfaz común. La utilidad de estos algoritmos puede dividirse en dos tipos, igual que la operativa de la herramienta: generación de hashes y comparación de estos.

`ProcessFuzzyHash` es la clase instanciada por Volatility cuando se ejecuta el plugin. Volatility requiere que esta clase esté contenida en un fichero con el mismo nombre en minúsculas, `processfuzzyhash`, que también será el nombre del comando que ejecute el plugin. Además, debe heredar de una clase que, a su vez, herede de la clase `common.Command`. Sólo bajo estas condiciones, Volatility lo identificará como un plugin. Se ha decidido que la clase heredada sea `common.AbstractWindowsCommand` ya que `ProcessFuzzyHash` está orientado a volcados de memoria de sistemas Windows. En esta clase se realizan las tareas de interpretar los parámetros, extraer los procesos del volcado de memoria e invocar a `HashEngine`, la clase encargada de generar hashes o compararlos. La extracción de procesos se ha delegado a otros plugins de Volatility: `procdump` es invocado para extraer el PE de los procesos cuando se requiere un hash del PE completo o de

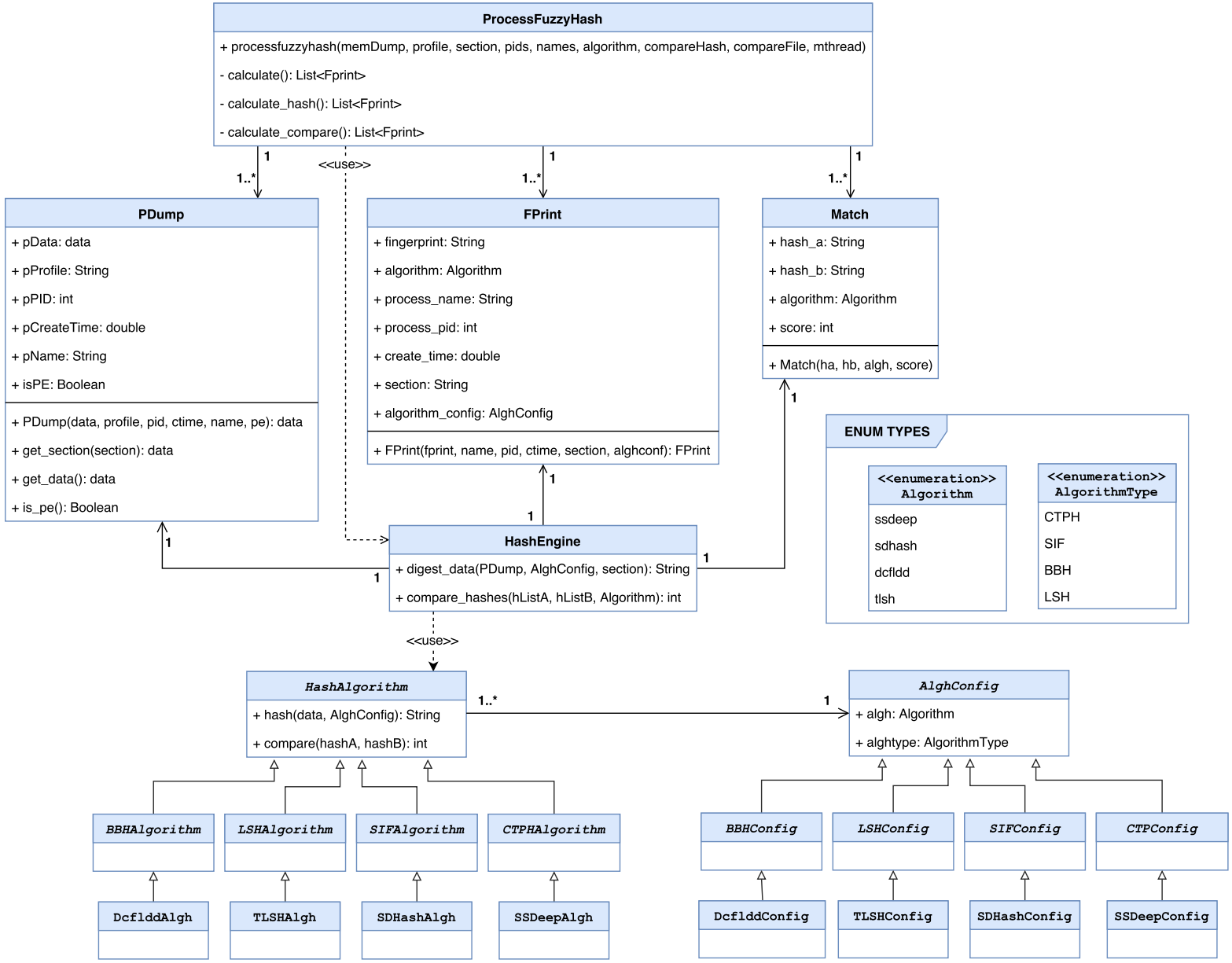


Figura 3.2: Diagrama de clases de ProcessFuzzyHash.

alguna de sus secciones; mientras que la extracción de todas las páginas de memoria usadas del proceso se realiza mediante `memdump`. Ambos reciben como parámetros una lista de PIDs y un directorio donde guardar la información extraída. Esta solución, aunque facilita la implementación, impide realizar una paralelización completa de la extracción de hashes.

`HashEngine` es la clase en la que se centralizan las tareas de generar y comparar hashes. Cada invocación de las funciones de generación o comparación de `HashEngine` produce un único resultado. Esta clase es la encargada de invocar el algoritmo adecuado o de informar mediante una excepción en caso de indicar un algoritmo no soportado.

`HashAlgorithm` es la clase abstracta que implementa el patrón *Facade*. Esta clase define las funciones abstractas `hash` y `compare`, implementadas por cada una de las subclases. De esta clase heredan otras clases abstractas `BBHAlgorithm`, `SIFAlgorithm`, `LSHAlgorithm` y `CTPHAlgorithm`, que representan las diferentes familias de algoritmos de fuzzy hashing consideradas. A su vez, de estas últimas heredan las clases que implementan los algoritmos de fuzzy hashing finalmente escogidos para la herramienta:

- `SDHashAlgorithm`: Implementa el algoritmo `sdhash` (véase la Sección 2.1.2), aprovecha las funciones proporcionadas por el módulo `fuzzyhashlib` [Ton].
- `TLSHAlgorithm`: Esta clase implementa el algoritmo `TLSH` (véase la Sección 2.1.5) haciendo uso del módulo `tlsh` [Mic15].
- `SSDeepAlgorithm`: Implementa las funciones de hash y comparación del algoritmo `ssdeep` (véase la Sección 2.1.1). Curiosamente, a pesar de que el módulo `fuzzyhashlib` ofrece estas funciones, se ha observado que las implementadas por el módulo `ssdeep` [Phi] obtienen mejores prestaciones.
- `DcflddAlgorithm`: Esta clase implementa el algoritmo `dcfldd`(véase la Sección 2.1.4). Las funciones de hash y comparación aprovechan las funciones expuestas por el módulo `dcfldd` desarrollado para este trabajo.

`AlghConfig` es otra clase abstracta que, de la misma forma que `HashAlgorithm`, implementa el patrón *Facade*. Las clases que heredan de esta clase almacenan configuraciones de algoritmos. Pueden observarse a simple vista las similitudes con la jerarquía de clases de `HashAlgorithm`. Esto se debe a que para cada clase de algoritmo existe una clase de configuración. En el planteamiento inicial de la implementación de ProcessFuzzyHash se planeó soportar distintas configuraciones para cada algoritmo, pero debido a que se ha comprobado experimentalmente que configuraciones alternativas no aportaban grandes cambios a los resultados obtenidos, se decidió utilizar siempre la configuración por defecto. No obstante, la jerarquía se ha mantenido de cara a trabajo futuro.

Por último, `Fprint`, `PDump` y `Match` son tres clases que actúan como objetos para pasar información entre las clases que realizan trabajo útil. `PDump` almacena la información asociada al volcado de un proceso, ya sean todas las páginas de memoria del proceso o el PE. Es utilizado por `ProcessFuzzyHash` para pasar a `HashEngine` toda la información

referente al proceso del que se va a generar el hash. Del mismo modo, `FPrint` es usado por `HashEngine` como respuesta a `ProcessFuzzyHash` cuando es invocado para calcular un hash. Contiene la información asociada a dicho hash: el propio hash calculado, algoritmo utilizado, nombre del proceso, etc. `Match` es también utilizado por `HashEngine` para comparar dos hashes. Contiene ambos hashes comparados, el algoritmo utilizado y el resultado de la comparación.

Finalmente, se han implementado dos tipos enumerados, `Algorithm` y `AlgorithmTypes`; ambos destinados a facilitar la identificación e instanciación de los algoritmos.

3.2. Ejecución

Al tratarse de un plugin de Volatility, ProcessFuzzyHash ha de ejecutarse a través del propio framework. Volatility se ejecuta con un script Python, `vol.py`, al que se le indica como mínimo un volcado de memoria y el comando a ejecutar. También será necesario indicar la localización de ProcessFuzzyHash, ya que se trata de un plugin externo al framework. La parametrización en Volatility se realiza a través de argumentos incluidos detrás del comando, ProcessFuzzyHash contempla los siguientes parámetros:

- **-P <pids>**: Recibe una lista de los PIDs, separados por comas, de los procesos de los que obtener su hash.
- **-N <nombres>**: Una lista de nombres de procesos, separados por comas, de los que obtener su hash.
- **-A <algoritmos>**: Este parámetro recibe la lista de algoritmos, separados por comas, que emplear en el cálculo o comparación de hashes.
- **-S <seccion>**: Recibe el nombre de la parte del proceso que utilizar para obtener el hash. La cadena indicada con este parámetro será utilizada para buscar la sección el PE en la tabla de secciones del PE. Hay dos excepciones: con la cadena `full` se calculará el hash del proceso completo, mientras que con la cadena `pe` se calculará el hash del PE completo.
- **-c <hash>**: Con este parámetro se indica uno o más hashes a utilizar para la comparación.
- **-C <fichero-hash>**: En este caso, se podrá indicar uno o más ficheros de hashes que utilizar para la comparación.
- **-M**: Este parámetro activa la generación de hashes en multi-hilo (pendiente de implementar actualmente).

```
$ python vol.py --plugins=ProcessFuzzyHash/ -f vmcore.elf \
> --profile=Win10x86_15063 processfuzzyhash -A ssdeep -S pe\
> -N VBoxService,winlogon,services

Volatility Foundation Volatility Framework 2.6
Name           PID  Create Time  Section Algorithm Hash
winlogon.exe   500  131483892000 pe      SSDeep    6144:pzP/qv...8ciJQdsyJqj
services.exe   544  131483892003 pe      SSDeep    6144:Q/6kXE...jXd5
VBoxService    1060 131483892039 pe      SSDeep    12288:K/oDR...CxuexSQ
```

(a) Generación de hashes mediante el algoritmo `ssdeep`

```
$ python vol.py --plugins=ProcessFuzzyHash/ -f vmcore.elf \
> --profile=Win10x86_15063 processfuzzyhash -A ssdeep -S pe -N svchost \
> -c '768:9n3SsSfvrOtOHW4CO5LTiMRMxVKPhPDjRWWm:d3BGrOtO2NO5LTiqUVKP5/zm'

Volatility Foundation Volatility Framework 2.6
Hash A                Hash B                    Algorithm Score
768:9n3Ss...qUVKP5/zm 768:9n3SsS...qDVKP5/0m    ssdeep    94
768:9n3Ss...qUVKP5/zm 768:9n3SsS...q5VKP5/0m    ssdeep    94
768:9n3Ss...qUVKP5/zm 768:9n3SsS...qUVKP5/zm    ssdeep    100
768:9n3Ss...qUVKP5/zm 768:9n3SsS...qFVKP5/zm    ssdeep    97
768:9n3Ss...qUVKP5/zm 768:9n3SsS...qMVKP5/zm    ssdeep    100
768:9n3Ss...qUVKP5/zm 768:9n3SsS...qAVKP5/zm    ssdeep    97
768:9n3Ss...qUVKP5/zm 768:9n3SsS...q9VKP5/zm    ssdeep    97
768:9n3Ss...qUVKP5/zm 768:9n3SsS...q+VKP5/zm    ssdeep    97
...
```

(b) Comparación de hashes del proceso `svchost` con el algoritmo `ssdeep`

Figura 3.3: Ejemplo de salida de ProcessFuzzyHash.

Salida del algoritmo

La salida generada por ProcessFuzzyHash depende de si genera hashes o los compara. En el caso de la generación, la salida está compuesta por una línea dedicada a cada hash generado con la siguiente información: nombre del proceso, PID del proceso, estampilla de tiempo del momento de creación del proceso, sección de la que se ha generado el hash, algoritmo con el que se ha calculado y el propio hash. Por otro lado, en el caso de la comparación, la salida se compone de una línea por cada comparación con la siguiente información: hash A, hash B, algoritmo utilizado y resultado de la comparación. La Figura 3.3 ilustra el resultado de una ejecución de generación y de comparación.

Capítulo 4

Experimentación

En este capítulo se recoge la metodología para la evaluación de ProcessFuzzyHash y los resultados obtenidos.

4.1. Entorno de pruebas

Como entorno de pruebas, se van a emplear varias máquinas virtuales alojadas en el hipervisor VirtualBox de Oracle, versión `5.1.26`. Estas máquinas se componen de las versiones de 32 (denominada x86) y 64 bits (denominada x64) de `Windows 7 Service Pack 1 Professional build 7601` y `Windows 10 Professional versión 1703 build 15063.0`.

Respecto al software elegido para la evaluación, se ha utilizado un conjunto de programas comúnmente objetivo de ataques que abarcan procesos de sistema, programas comerciales, así como malware que se describen brevemente a continuación:

- `winlogon.exe`: Realiza una serie de tareas críticas relacionadas con la autenticación de usuario, con lo que siempre hay un proceso de `winlogon` ejecutándose en Windows. Se han analizado las versiones `10.0.15063.0` para Windows 10 y `6.1.7601.18540` para Windows 7.

- `svchost.exe`: `ServiceHost` es un proceso de sistema que se encarga de manejar diversos servicios compartidos dentro de Windows. Es habitual encontrar numerosas instancias de este mismo proceso en ejecución al mismo tiempo, cada una de ellas controlando un servicio distinto. Se han estudiado las versiones `10.0.15063.0` para Windows 10 y `6.1.7600.16385` para Windows 7.

- `explorer.exe`: Este proceso es conocido como Explorador de Windows y es de vital importancia ya que es el gestor de ventanas de Windows. Por ello es comúnmente objetivo de cantidad de ataques, dado que es capaz de registrar las acciones del usuario como movimientos de ratón o pulsaciones de teclas. Se han analizado las versiones `10.0.15063.0` para Windows 10 y `6.1.7601.17514` para Windows 7.

- `iexplore.exe`: Se trata del proceso del explorador web por defecto de Windows 7 (y versiones anteriores). Se evaluará la última versión disponible y soportada: Internet Explorer 11, en su versión `11.0.9600.17843`.
- `MicrosoftEdge.exe`: Este explorador web viene con el sistema operativo Windows 10. Se trata del sustituto de Internet Explorer y, de la misma forma que este último, es el explorador web por defecto. Se ha evaluado la versión `40.15063.0.0`.
- `chrome.exe`: Es el proceso del famoso explorador web Google Chrome. Se han usado las versiones `stable-60.0.3112.01` y `beta-61.0.3163.49` para la experimentación.
- `firefox.exe`: Se trata del proceso de Mozilla Firefox, otro explorador web muy utilizado. Se han considerado las versiones `stable-55.0.2` y `beta-56.0b5`.
- `AcroRd32.exe`: Es el proceso de Acrobat Reader, el lector de ficheros PDF gratuito de Adobe. Se ha evaluado la versión `DC-2017.012`.
- `ALINA.exe`: Es un malware de tipo POS RAM scrapper [Rod17]. Su operativa consiste en recorrer los espacios de direcciones del resto de procesos buscando secuencias de bytes que correspondan con una tarjeta de crédito para después exfiltrarlas a un servidor remoto. El hash `MD5` de la muestra considerada es `0de9765c9c40c2c2f372bf92e0ce7b68`.

Aunque inicialmente se planeó evaluar resultados con hashes de procesos completos, tras realizar una evaluación preliminar, se decidió excluir este caso dada la falta de relevancia observada en sus resultados. Además, como se ha comentado en el Capítulo 3, el rendimiento durante la obtención de hashes de procesos completos supera en varios órdenes de magnitud al resto de configuraciones. Por tanto, se contemplan los siguientes casos:

- `PE`: Se obtendrán hashes del PE completo de cada proceso.
- **Sección** `.text`: Se trata de la sección del PE que contiene el código binario cargado en memoria.
- **Sección** `.rsrc`: Así como el caso anterior, se trata de una sección del PE. En este caso almacena datos de sólo lectura. Durante la experimentación se ha observado que ciertos ejecutables no cuentan con esta sección en su PE, concretamente el malware ALINA. Por ello, para este malware, se utilizará la sección .rdata, que también contiene datos de sólo lectura (algunos de los otros software considerados no disponen de esta sección `.rdata`).

Por último, se han definido los siguientes escenarios o casos de estudio:

Caso A **Mismo programa y proceso en distintas ejecuciones de la misma máquina.** Se evaluarán similitudes para todos los programas.

Caso B **Mismo programa en distinta máquina, con diferente sistema operativo y misma arquitectura.** En este caso se van a evaluar similitudes para todos los programas excepto para `iexplore.exe` y `MicrosoftEdge.exe`, dado que ambos son propios de tan sólo un sistema operativo de los evaluados.

Caso C **Mismo programa en distinta máquina, con el mismo sistema operativo y diferente arquitectura.** De nuevo, se evaluarán similitudes para todos los programas.

Caso D **Mismo programa con distintas versiones (`stable/beta`) en el mismo sistema operativo con la misma arquitectura.** Se evaluarán las diferencias entre `chrome.exe` y `firefox.exe` en sus versiones `stable` y `beta` en máquinas idénticas. Este caso se desvía de la línea que siguen los tres anteriores puesto que se ha considerado interesante evaluar similitudes en un software tan "vivo" como un navegador web (software que sufre de actualizaciones recurrentes).

4.2. Extracción de resultados

La máquina sobre la que se han realizado las pruebas tiene un sistema operativo `Ubuntu 16.04 GNOME de 64 bits` con un procesador `Intel Core i7-4720HQ @ 2.60GHz`, `12GiB @ 1867MHz` de memoria RAM y un disco de estado sólido `Samsung 850 PRO` con interfaz `SATA3`. Cabe destacar que las máquinas virtuales utilizadas se encuentran almacenadas en un disco duro externo de `2,5"` de `1TB @ 5400 RPM` con interfaz `USB 3.0`.

La extracción de resultados se ha realizado en un entorno de pruebas compuesto por dos *scripts* Python, ocho máquinas virtuales manejadas por el hipervisor Oracle VirtualBox y una base de datos MySQL. La Figura 4.1 contiene un diagrama del entorno dicho entorno de pruebas.

El primer *script* realiza las siguientes tareas: encendido y apagado de las máquinas, extracción del volcado de memoria, generación de hashes mediante ProcessFuzzyHash e inserción de resultados en la base de datos. Tras la extracción de todos los hashes necesarios, el segundo *script* se encarga de recogerlos de la base de datos, realizar las comparaciones usando ProcessFuzzyHash e insertar los resultados en la misma base de datos.

El tiempo total que tardan en ejecutarse estas pruebas varía entre las cuatro y las cinco horas. El primer *script* realiza cinco iteraciones de una extracción completa de resultados. Una iteración conlleva la ejecución y extracción (secuencial) de memoria de las ocho máquinas virtuales, así como el cálculo de los cuatro tipos de hash para cada uno de los procesos. El segundo *script* tarda menos de un minuto en realizar todas las comparaciones. Esta gran diferencia la marca principalmente, el tiempo de encendido y apagado de las máquinas virtuales.

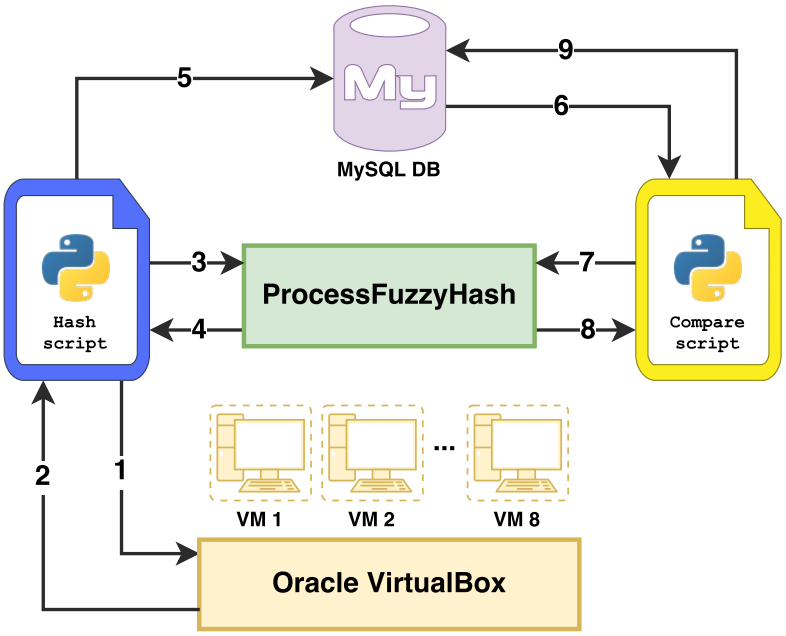


Figura 4.1: Diagrama de alto nivel de la operativa de automatización de la experimentación.

4.3. Resultados

Las Figuras 4.2 a 4.13 muestran los resultados de los experimentos realizados. Cada figura sigue la misma estructura: la parte izquierda muestra los resultados para la arquitectura de 32 bits, mientras que en la parte derecha se muestran los resultados para la arquitectura de 64 bits. Además, se dividen en tres filas, cada una atendiendo a los casos contemplados: análisis del PE completo, de la sección `.text`, o de la sección `.rsrc`, respectivamente. Nótese que en las figuras referentes al algoritmo TLSH, tratado aparte, un menor valor implica una mayor ratio de similitud.

Caso A: Misma máquina, mismo ejecutable, diferentes ejecuciones

La sección `.rsrc` es la que mayor ratio de similitud general obtiene para todos los algoritmos (salvo en ALINA), en torno a un 50 % de similitud (véanse Figuras 4.2 y 4.4 (a) y (b)). Puede observarse una ligera mejora en Windows 10 sobre Windows 7. La sección `.text` obtiene muy buenos resultados en x64 excepto en ALINA, Acrobat Reader e Internet Explorer, mientras que en x86 se observan resultados muy pobres para todos los programas y algoritmos (véanse Figuras 4.2 y 4.4 (c) y (d)). La sección `pe` de x86 obtiene resultados muy similares a `.text`, como puede observarse en las Figuras 4.2 y 4.4 (e). Puede observarse una ligera mejora general, aunque destaca en Internet Explorer donde mejora considerablemente. En x64 los resultados empeoran para todos los casos (véanse Figuras 4.2 y 4.4 (f)). El algoritmo que más porcentaje de similitud ha obtenido

para este caso es `dcfldd`. `Ssdeep` y `sdhash` se encuentran en segundo lugar, aunque `ssdeep` tiene más casos en los que obtiene un 0 % de similitud. En el caso de `TLSH` pude observarse una tendencia similar a la obtenida por `sdhash` en todos los casos, excepto en la sección `pe` de svchost en Windows 7, donde obtiene peores resultados (véanse Figuras 4.3 (e) y (f)).

Caso B: Mismo programa, misma arquitectura, distinto sistema operativo

La sección `.rsrc` obtiene resultados del casi 100 % en Acrobat Reader, Chrome y Firefox. Sin embargo, para los procesos de sistema Explorer y Winlogon los resultados son casi nulos, destacando svchost que guarda en torno a un 70 % de parecido (véanse Figuras 4.6 (a) y (b)). ALINA obtiene resultados muy similares al caso A. En la sección `.text` puede observarse un 0 % de similitud general en todos los procesos de sistema (véanse Figuras 4.6 (c) y (d)). Chrome y Firefox obtienen muy buenos resultados en x64 (entre 90 % y 95 %), pero caen hasta menos de un 40 % en x86. ALINA y Acrobat Reader obtienen resultados nulos con `ssdeep` e inferiores al 40 % con `dcfldd` y `sdhash`, siendo similares los resultados en ambas arquitecturas. Los resultados para la sección `pe` son ligeramente mejores a los de la sección `.text`, salvo para Chrome y Firefox, que empeoran (véanse Figuras 4.6 (e) y (f)). `TLSH` sigue la misma tendencia que `dcfldd`, que de nuevo obtiene los mejores resultados.

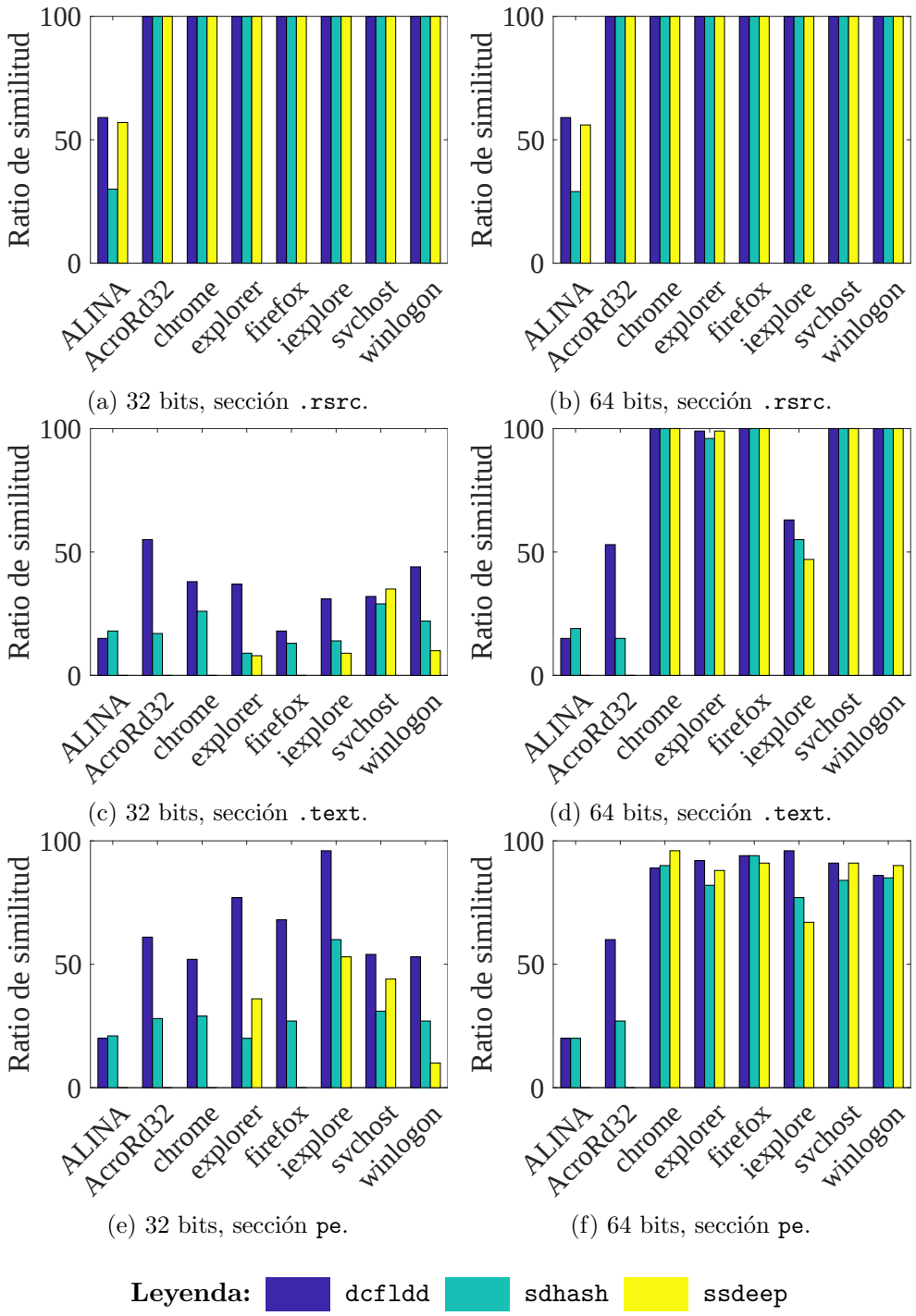
Caso C: Mismo programa, mismo sistema operativo, distinta arquitectura

En general, la sección `.rsrc` obtiene peores resultados que en los casos anteriores, salvo en el caso de ALINA que obtiene idénticos resultados, como puede observarse en las Figuras 4.8 (a) y (b). La sección `.text` obtiene unos resultados casi nulos para todos los programas salvo en ALINA y Acrobat Reader, donde `dcfldd` y `sdhash` obtienen resultados en torno al 30 % (véanse Figuras 4.8 (c) y (d)). La sección `pe` mejora ligeramente los resultados respecto `.text`, sobre todo en Internet Explorer donde `dcfldd` logra casi un 90 % de similitud (véanse Figuras 4.8 (e) y (f)). En este caso, `ssdeep` ha obtenido los peores resultados con diferencia, `sdhash` logra encontrar algún parecido fuera de la sección `.rsrc`, pero nada por encima del 30 %, y `dcfldd` logra mejores resultados que todos los demás.

Caso D: Misma máquina, mismo programa, distinta versión, distintas ejecuciones

En este caso, la sección `.rsrc` obtiene peores resultados comparándolos con los anteriores (véanse Figuras 4.10 y 4.12 (a) y (b)). En todos se obtiene en torno a un 80 % de similitud, salvo en Windows 10 de 64 bits, donde sólo `dcfldd` destaca, aunque no supera el 80 % de similitud (véase Figura 4.12 (b)). La sección `.text` obtiene resultados casi nulos, excepto en el caso del algoritmo `TLSH` en Windows 10 y Windows 7 de 64 bits

donde logra superar, con diferencia, al resto de algoritmos en el caso de Chrome (véanse Figuras 4.11 y 4.13 (d)). La sección `pe` mejora los resultados obtenidos de `.text`, aunque siguen siendo muy pobres y sólo `dcfldd` pasa del 50 % de similitud en x86 (véanse Figuras 4.10 y 4.12 (e)). De nuevo, `dcfldd` obtiene los mejores resultados, exceptuando el caso de Windows 10 de 64 bits, en donde destaca `TLSH`.



(a) 32 bits, sección `.rsrc`.

(b) 64 bits, sección `.rsrc`.

(c) 32 bits, sección `.text`.

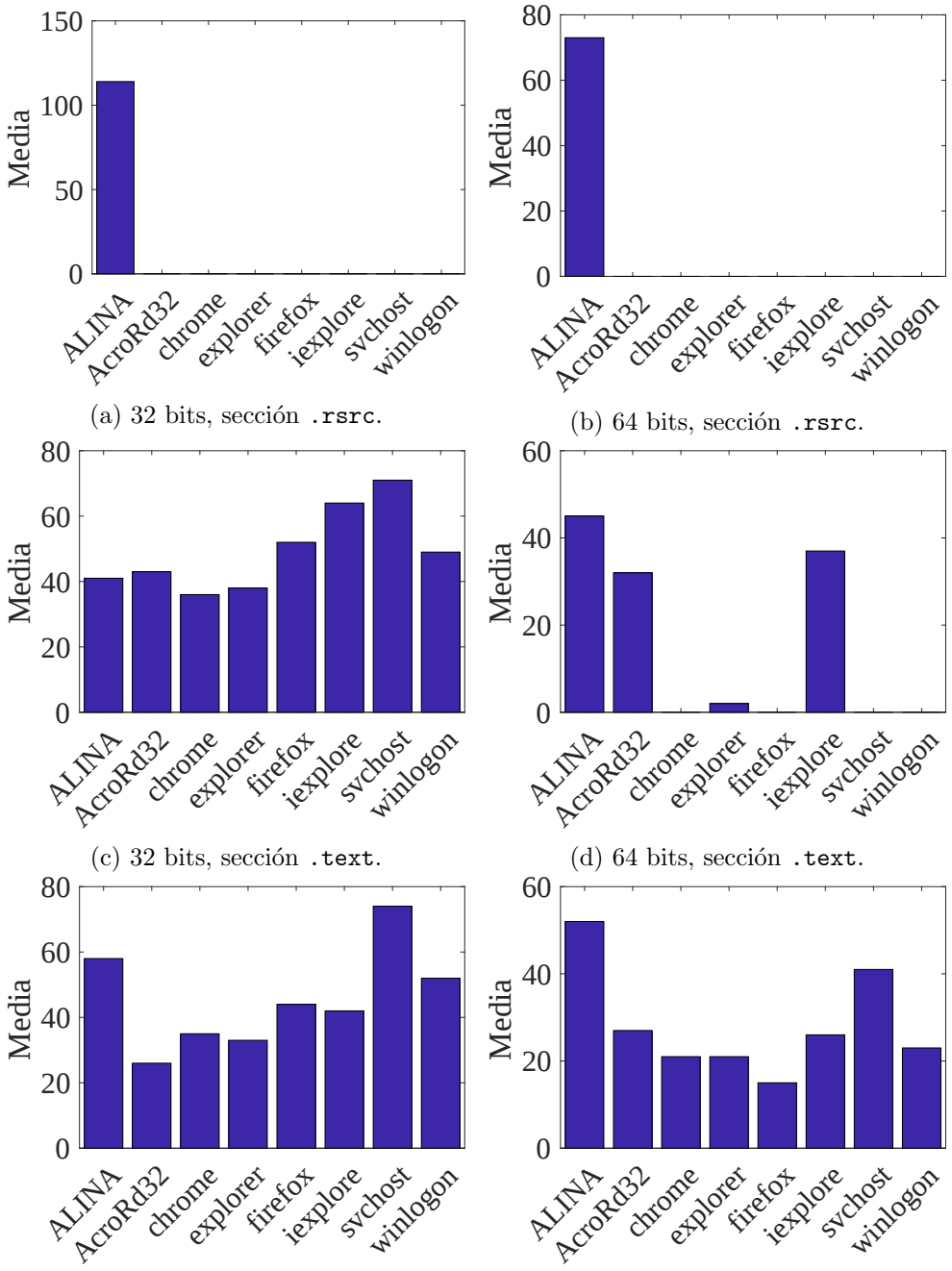
(d) 64 bits, sección `.text`.

(e) 32 bits, sección `pe`.

(f) 64 bits, sección `pe`.

Leyenda: `dcfldd` `sdhash` `ssdeep`

Figura 4.2: Caso A: Windows 7.



(a) 32 bits, sección `.rsrc`.

(b) 64 bits, sección `.rsrc`.

(c) 32 bits, sección `.text`.

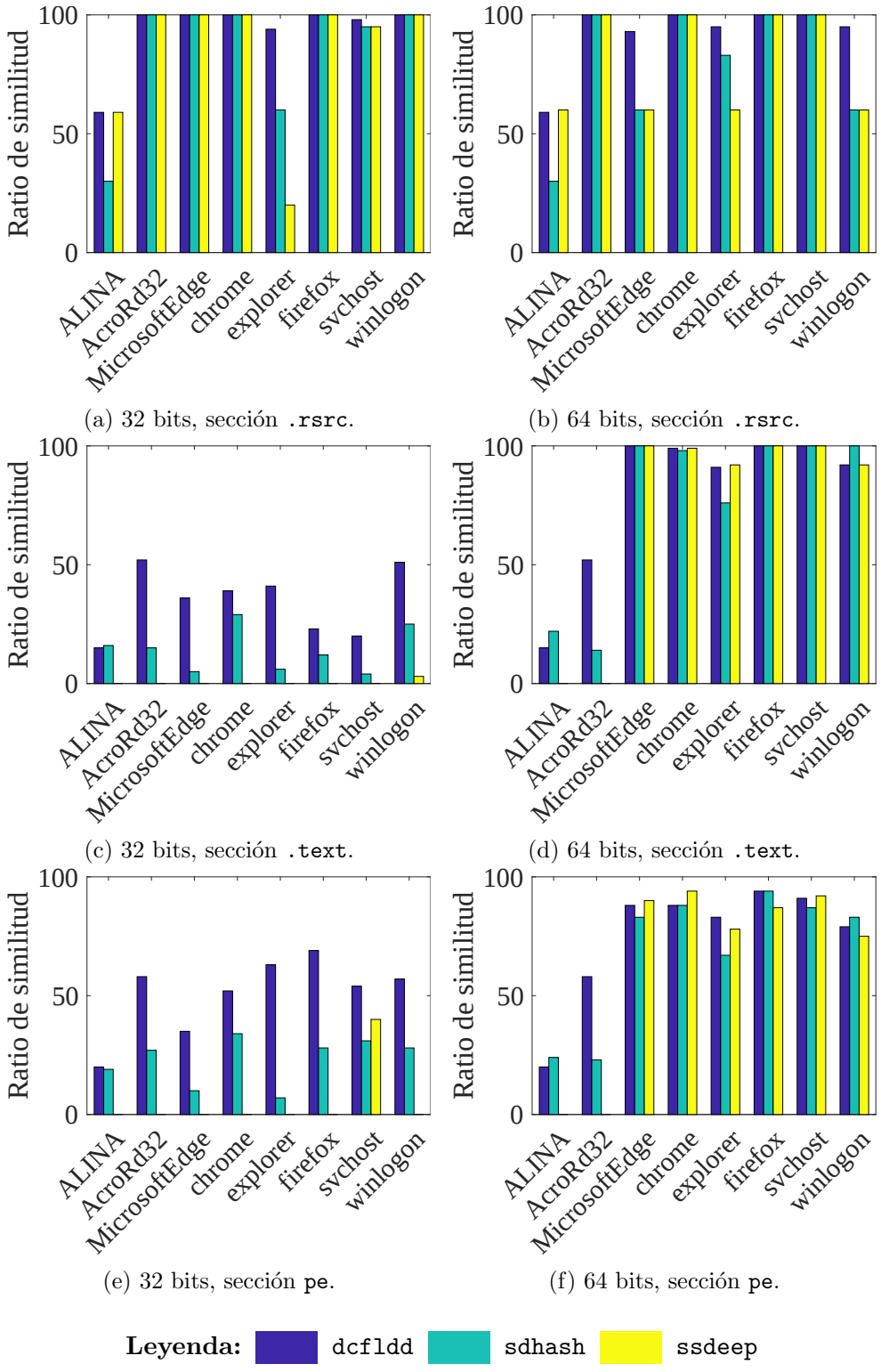
(d) 64 bits, sección `.text`.

(e) 32 bits, sección `pe`.

(f) 64 bits, sección `pe`.

Leyenda: TLSH

Figura 4.3: Caso A: Windows 7 (TLSH).



(a) 32 bits, sección `.rsrc`.

(b) 64 bits, sección `.rsrc`.

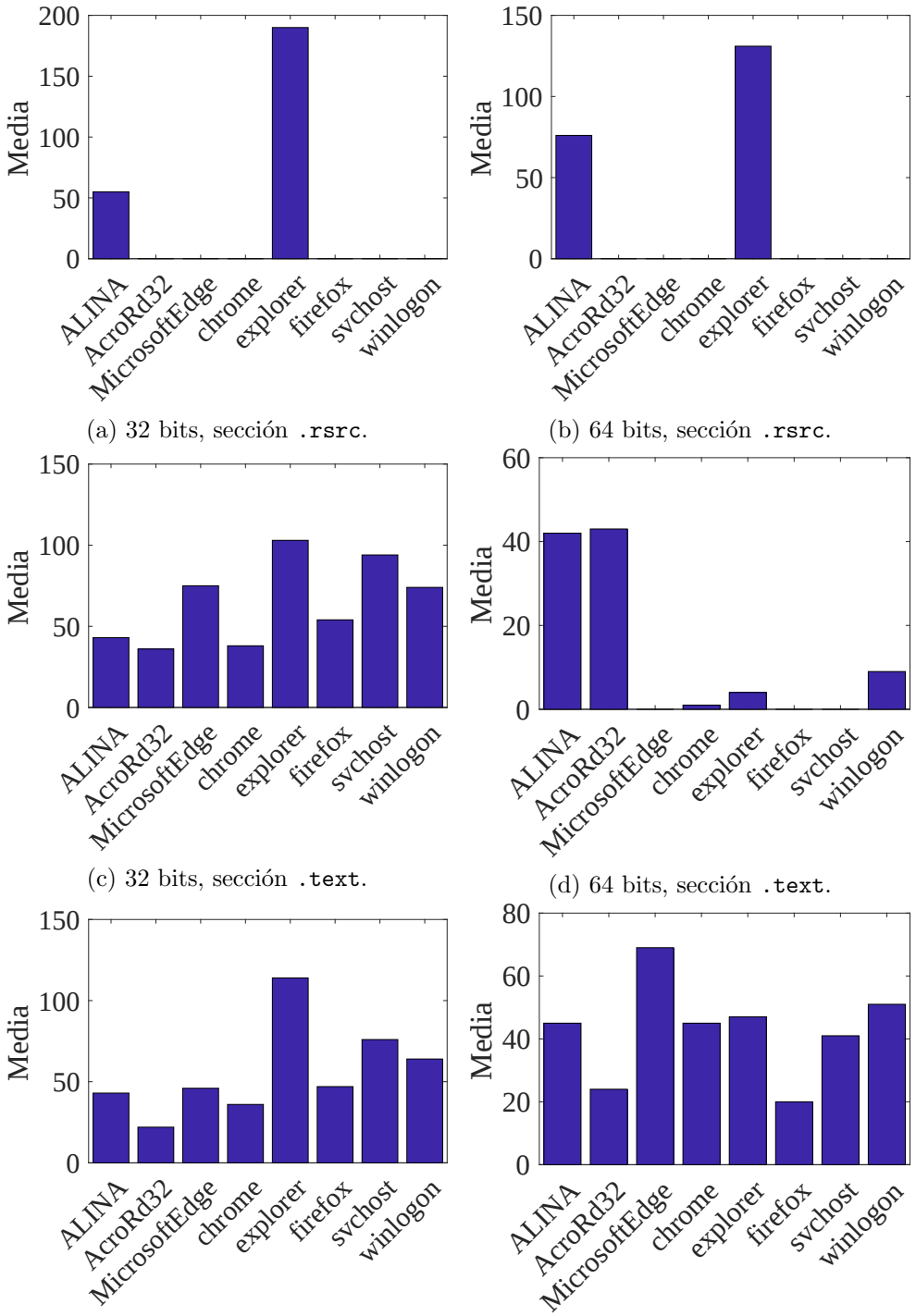
(c) 32 bits, sección `.text`.

(d) 64 bits, sección `.text`.

(e) 32 bits, sección `pe`.

(f) 64 bits, sección `pe`.

Figura 4.4: Caso A: Windows 10.



(a) 32 bits, sección `.rsrc`.

(b) 64 bits, sección `.rsrc`.

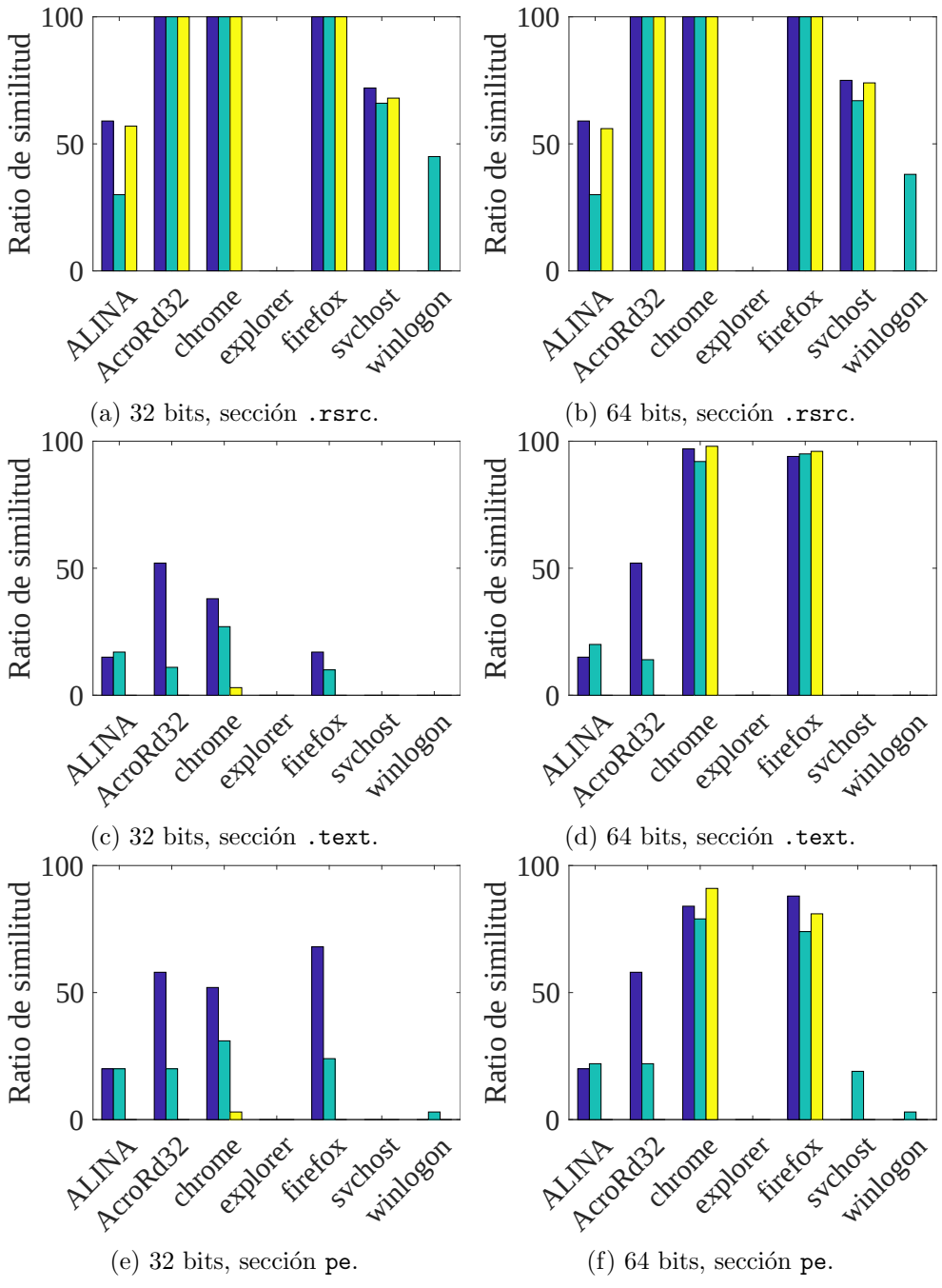
(c) 32 bits, sección `.text`.

(d) 64 bits, sección `.text`.

(e) 32 bits, sección `pe`.

(f) 64 bits, sección `pe`.

Figura 4.5: Caso A: Windows 10 (TLSH).



(a) 32 bits, sección `.rsrc`.

(b) 64 bits, sección `.rsrc`.

(c) 32 bits, sección `.text`.

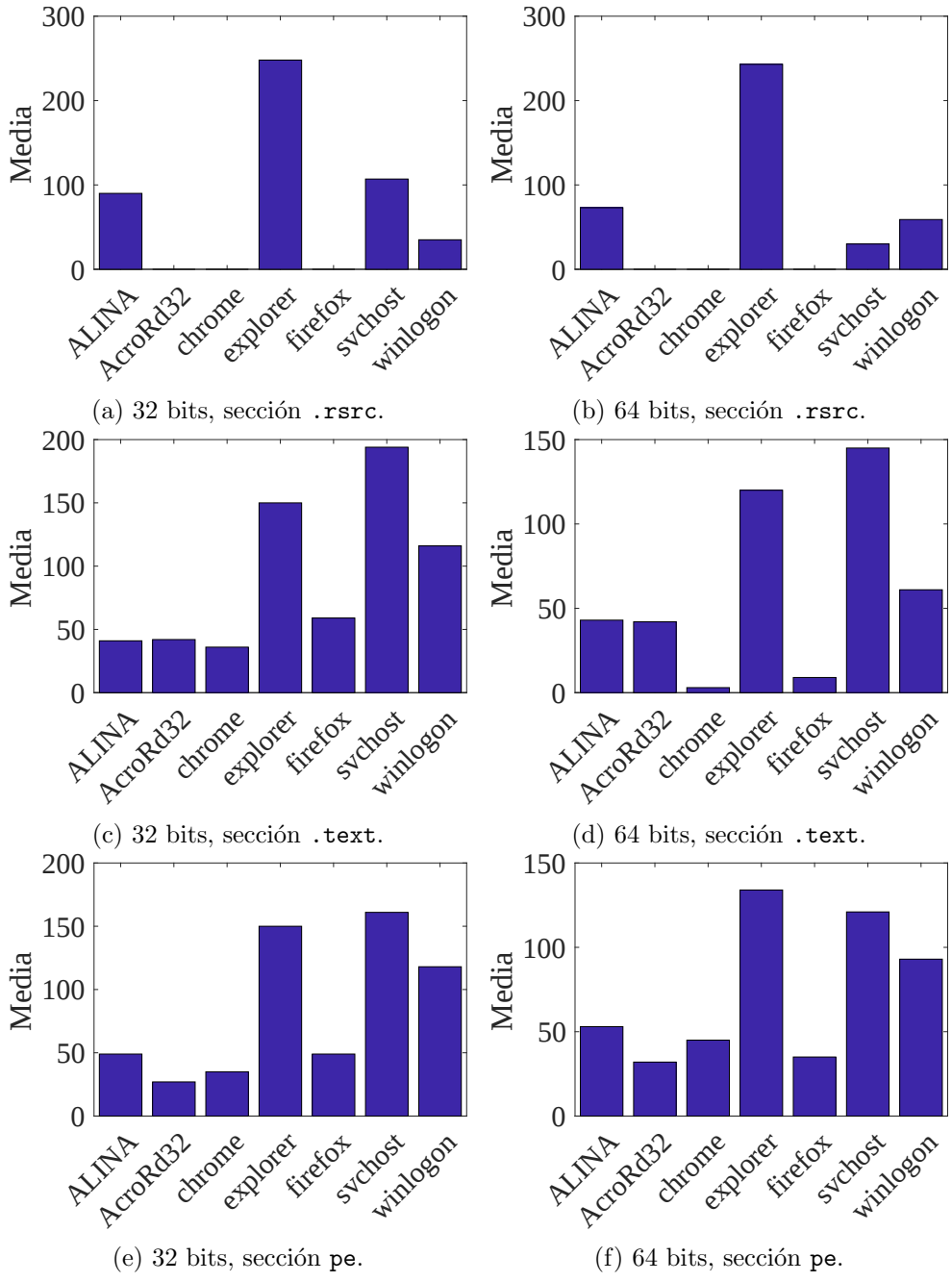
(d) 64 bits, sección `.text`.

(e) 32 bits, sección `pe`.

(f) 64 bits, sección `pe`.

Leyenda: `dcfldd` `sdhash` `ssdeep`

Figura 4.6: Caso B



(a) 32 bits, sección `.rsrc`.

(b) 64 bits, sección `.rsrc`.

(c) 32 bits, sección `.text`.

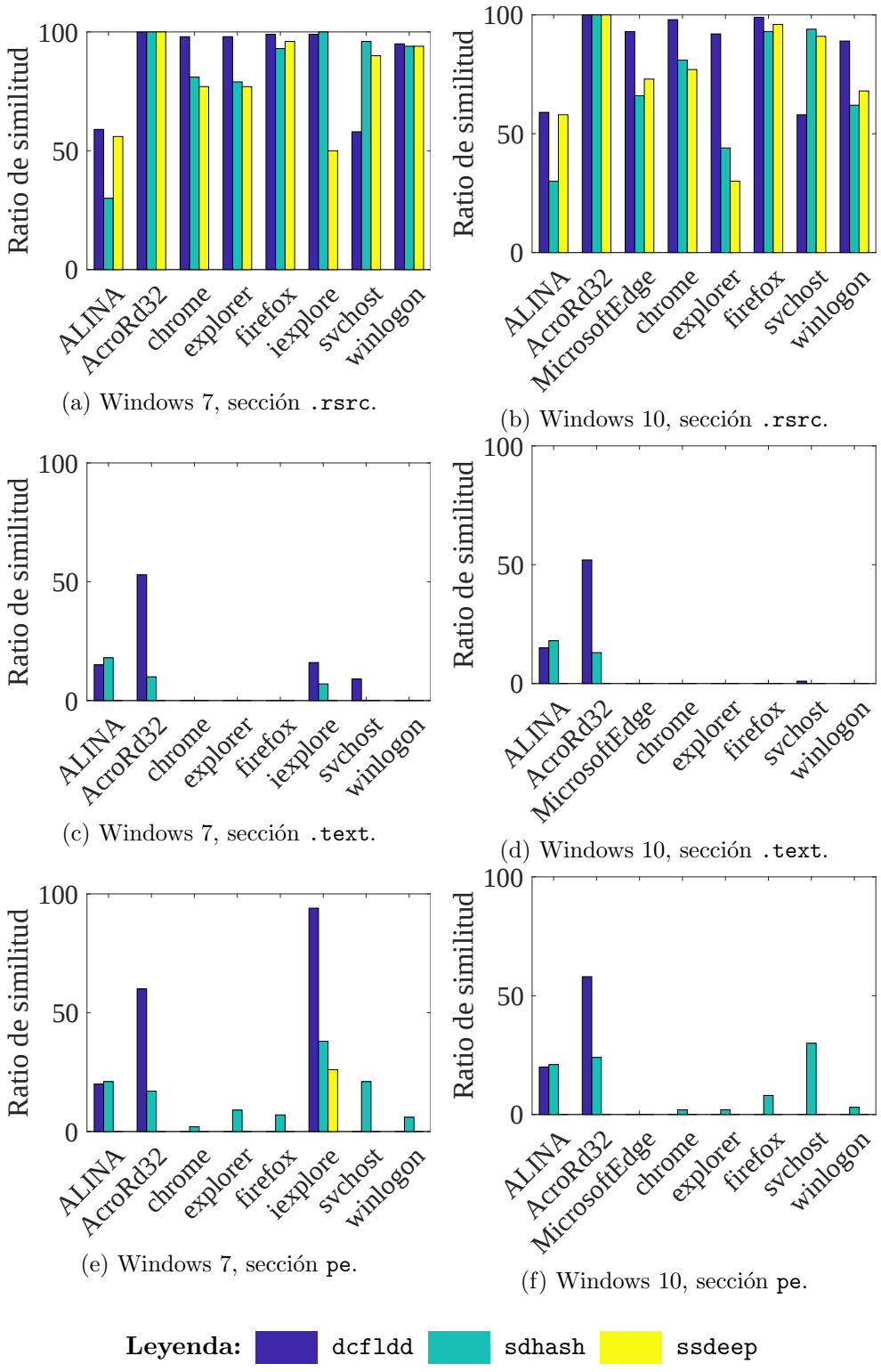
(d) 64 bits, sección `.text`.

(e) 32 bits, sección `pe`.

(f) 64 bits, sección `pe`.

Leyenda: TLSH

Figura 4.7: Caso B (TLSH).



(a) Windows 7, sección `.rsrc`.

(b) Windows 10, sección `.rsrc`.

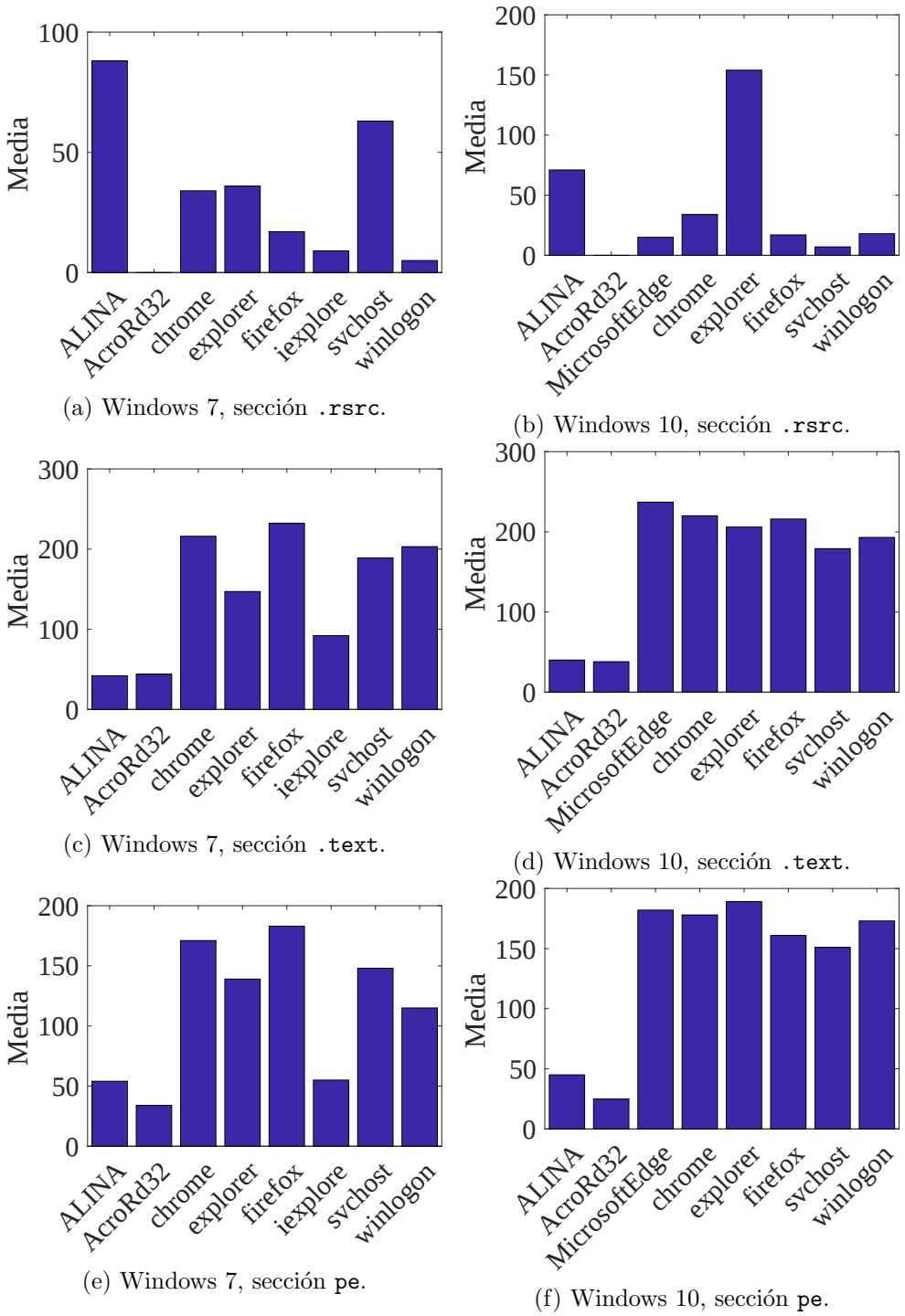
(c) Windows 7, sección `.text`.

(d) Windows 10, sección `.text`.

(e) Windows 7, sección `pe`.

(f) Windows 10, sección `pe`.

Figura 4.8: Caso C



(a) Windows 7, sección `.rsrc`.

(b) Windows 10, sección `.rsrc`.

(c) Windows 7, sección `.text`.

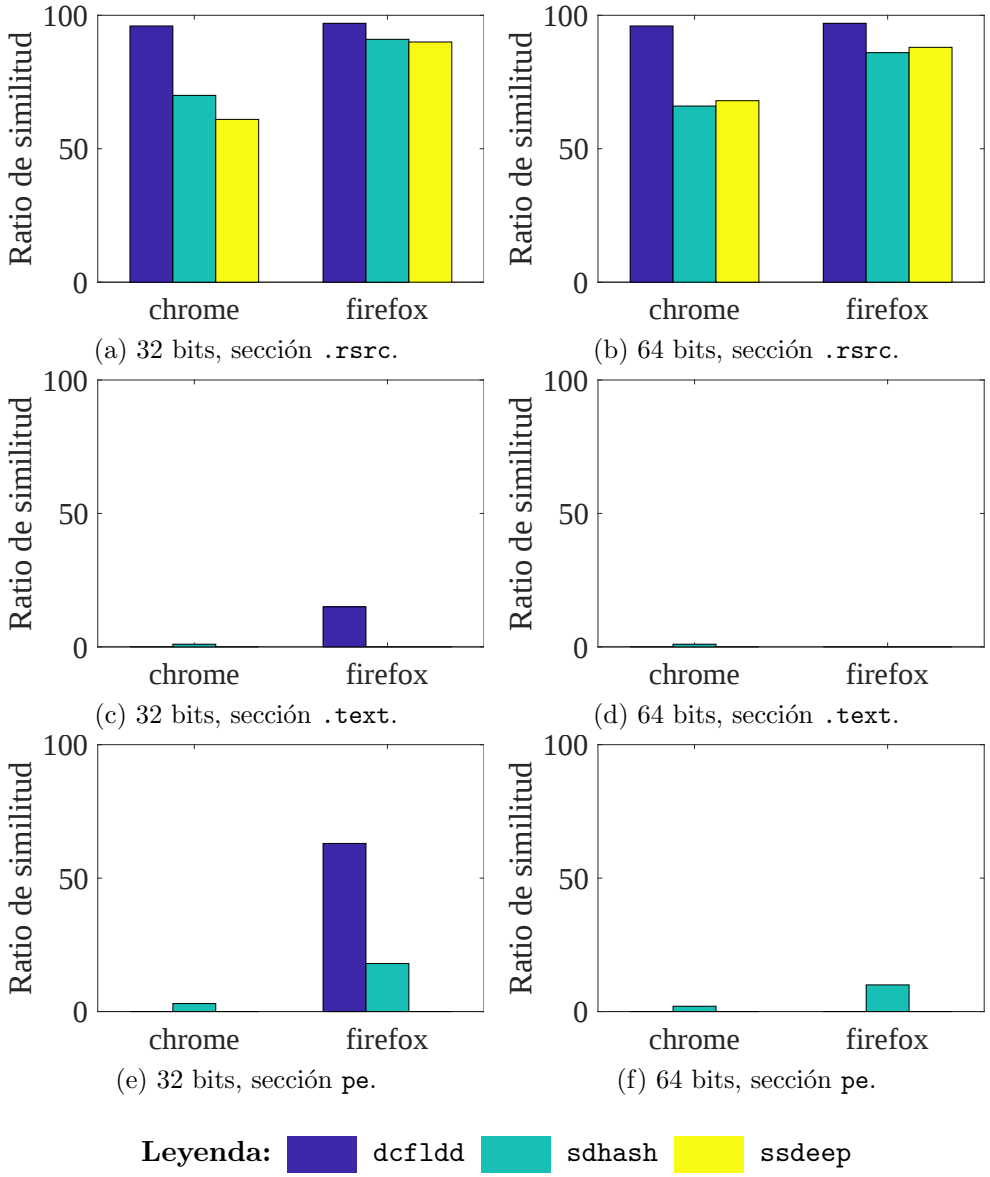
(d) Windows 10, sección `.text`.

(e) Windows 7, sección `pe`.

(f) Windows 10, sección `pe`.

Leyenda: TLSH

Figura 4.9: Caso C (TLSH).



(a) 32 bits, sección `.rsrc`.

(b) 64 bits, sección `.rsrc`.

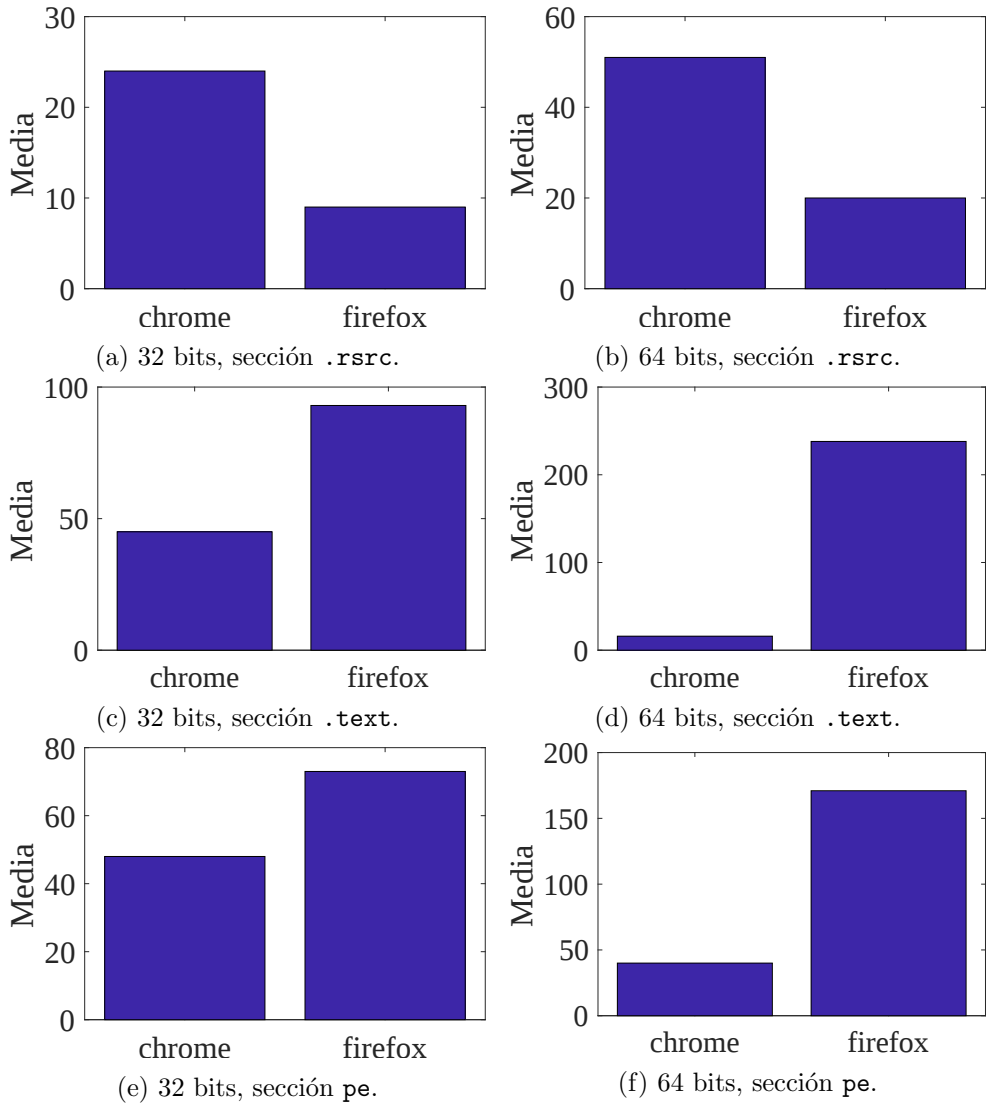
(c) 32 bits, sección `.text`.

(d) 64 bits, sección `.text`.

(e) 32 bits, sección `pe`.

(f) 64 bits, sección `pe`.

Figura 4.10: Caso D: Windows 7.



(a) 32 bits, sección `.rsrc`.

(b) 64 bits, sección `.rsrc`.

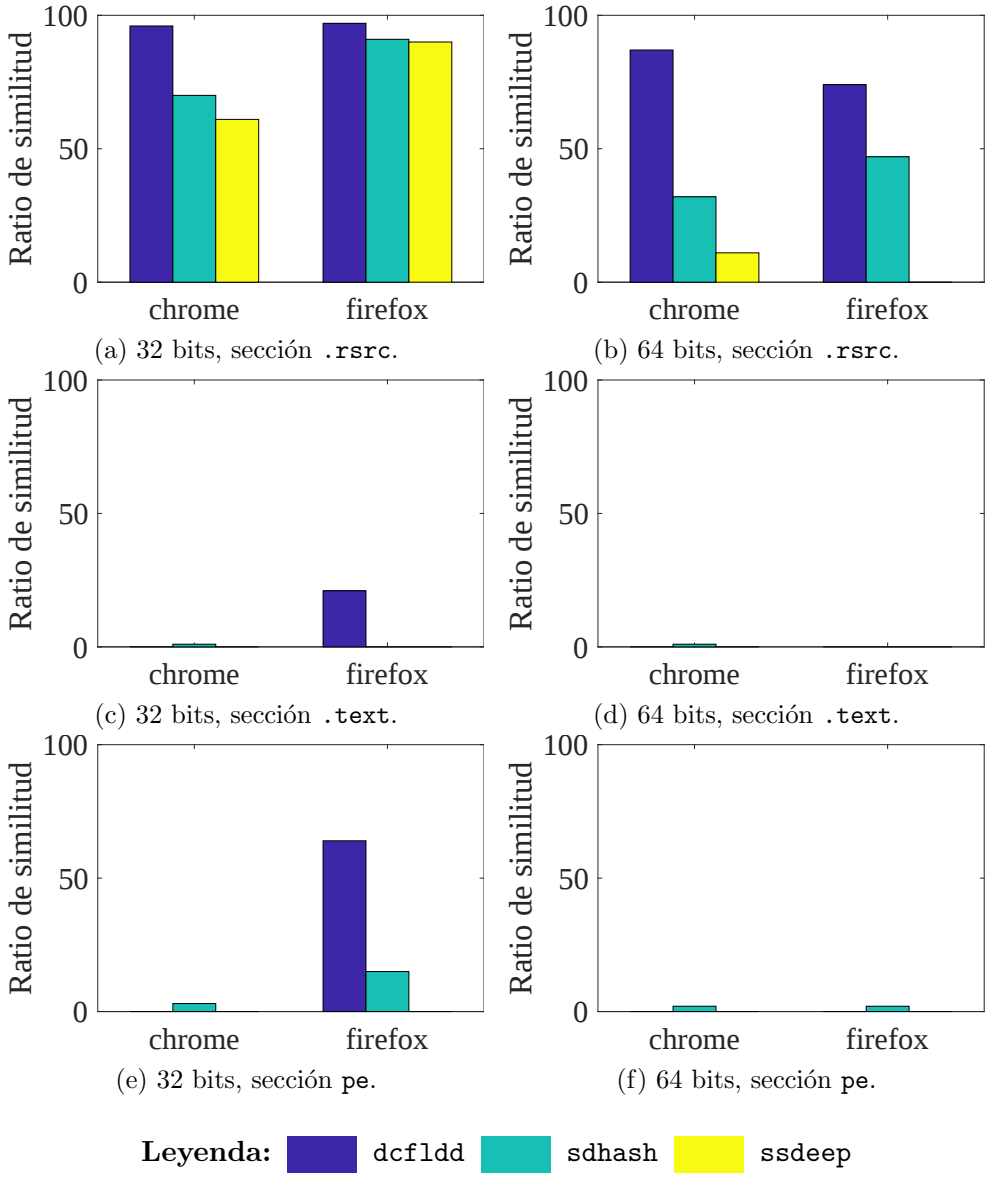
(c) 32 bits, sección `.text`.

(d) 64 bits, sección `.text`.

(e) 32 bits, sección `pe`.

(f) 64 bits, sección `pe`.

Figura 4.11: Caso D: Windows 7 (TLSH).



(a) 32 bits, sección `.rsrc`.

(b) 64 bits, sección `.rsrc`.

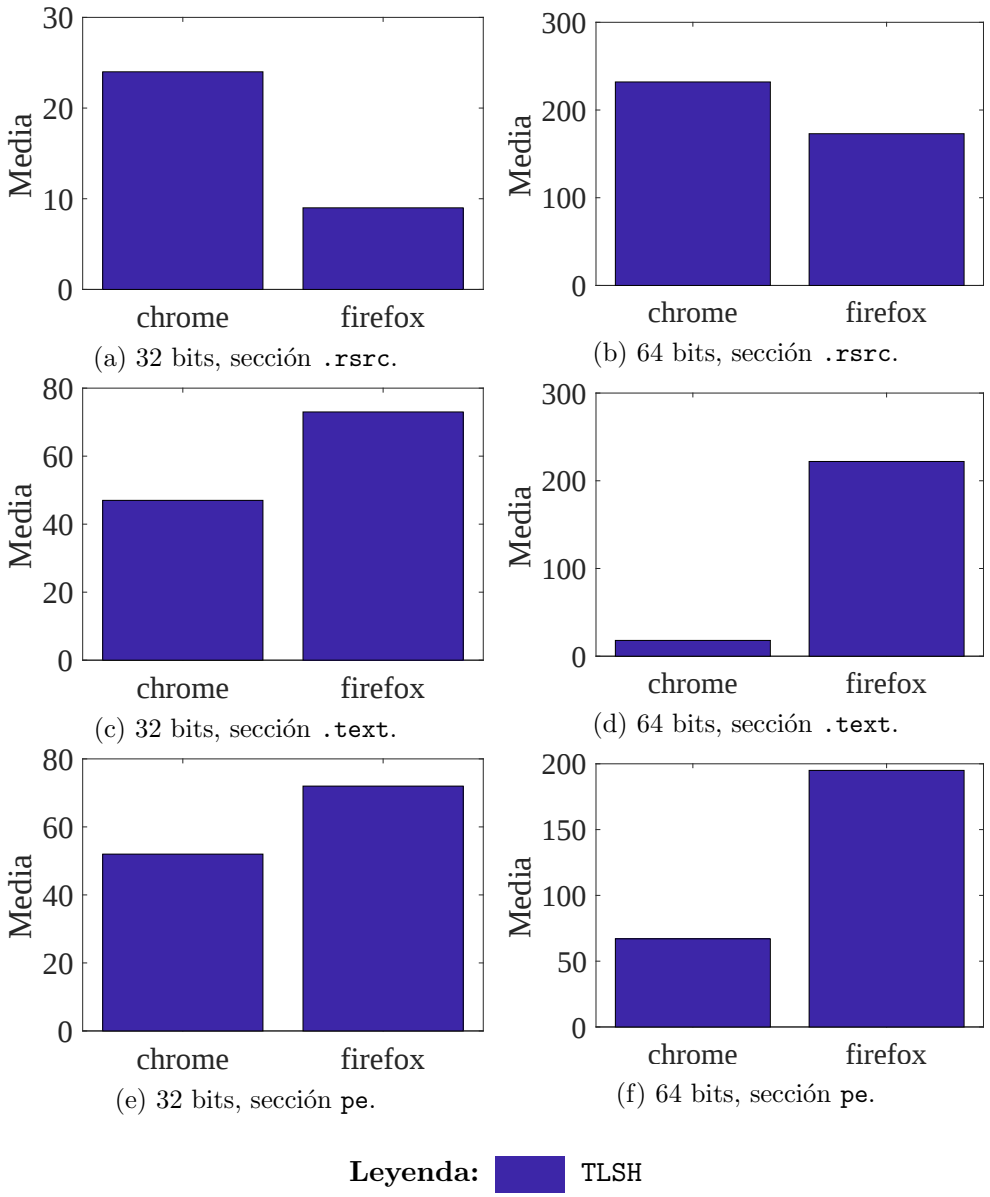
(c) 32 bits, sección `.text`.

(d) 64 bits, sección `.text`.

(e) 32 bits, sección `pe`.

(f) 64 bits, sección `pe`.

Figura 4.12: Caso D: Windows 10.



(a) 32 bits, sección `.rsrc`.

(b) 64 bits, sección `.rsrc`.

(c) 32 bits, sección `.text`.

(d) 64 bits, sección `.text`.

(e) 32 bits, sección `pe`.

(f) 64 bits, sección `pe`.

Leyenda: `TLSH`

Figura 4.13: Caso D: Windows 10 (TLSH).

Capítulo 5

Trabajo relacionado

En este capítulo se describen brevemente una serie de herramientas y publicaciones relacionadas con este trabajo.

Malfunction [Dyn] es una herramienta dedicada al análisis de malware centrada en funciones. Escrita en Python, utiliza `Radare2`[Rad] para buscar funciones contenidas en el código binario de un ejecutable y obtener su fuzzy hash. La operativa de Malfunction consiste en generar una base de datos de hashes de malware para luego comprobar si binarios desconocidos coinciden con alguno de los almacenados. Binwally [Rod] es una herramienta que usa fuzzy hashing para localizar cambios entre distintas versiones de un mismo software, más concretamente para averiguar en qué momento se introdujeron o parchearon *bugs* en el código.

La herramienta desarrollada en este trabajo supone una aportación a este conjunto ya existente de herramientas, dado que: (i) ninguna de ellas contempla diferentes algoritmos; (ii) se centran en un único caso en lugar de permitir obtener hashes de distintos tipos de secciones; y (iii) se aplican sobre contenido del fichero binario estático en lugar de dinámico.

Además de estas herramientas, existen algunos estudios sobre técnicas de fuzzy hashing. En [SBAAN16] se realiza una comparación y evaluación de prestaciones de varios algoritmos de fuzzy hash aplicados a detección de malware. Dicho trabajo concluye que las técnicas de fuzzy hash son una potente vertiente para la detección de malware pero que todavía necesita desarrollo dado que no hay una propuesta específicamente orientada a detección de similitudes en ficheros binarios. [NMAM+16] realiza un estudio centrado en el análisis estático de PEs. En lugar de centrarse en distintos algoritmos, compara los resultados obtenidos al emplear sólo ciertas partes del PE para detectar similitudes. El estudio concluye que los resultados obtenidos son prometedores, pero que todavía necesitan trabajo. También adelanta que la combinación de estas técnicas con heurísticas, que permitan reducir el tamaño de los conjuntos a analizar, podría ser de interés. [AS15] trata sobre la clusterización de malware desde una perspectiva diferente a la habitual, ya que convierten el contenido binario de programas a imágenes JPG a las que aplican distintos algoritmos de fuzzy hash para averiguar a qué familia de malware pertenecen.

El contenido de este trabajo es muy teórico y los autores concluyen que este método, a pesar de lograr menos precisión que los convencionales, logra un rendimiento mucho mejor.

A pesar de que entre todas estas aportaciones se abarcan todos los campos tratados en este trabajo (diferentes algoritmos, diferentes partes de un fichero PE y orientadas a malware), ninguno trata el análisis dinámico del fichero ejecutable ni presenta un banco de resultados tan extenso como el obtenido en este trabajo.

Capítulo 6

Conclusiones y líneas futuras

En este último capítulo se presentan las conclusiones del proyecto, así como sus implicaciones. Además, se plantean varias líneas futuras de trabajo.

A lo largo de este trabajo se han estudiado múltiples algoritmos de fuzzy hashing y se han seleccionado y evaluado cuatro de ellos (concretamente, `ssdeep`, `dcfldd`, `sdhash` y `TLSH`) para similitud entre procesos. Los resultados indican que el algoritmo que mayor ratio de similitud obtiene en casi todos los casos es `dcfldd`, a pesar de contar con la operativa más sencilla. Por otro lado, se ha observado que `TLSH` no es constante y obtiene buenos resultados en casos puntuales. Los otros dos algoritmos considerados, `ssdeep` y `sdhash`, parecen obtener resultados parecidos, aunque `sdhash` logra mejores resultados en general.

Se han evaluado similitudes en el PE de un proceso, así como en sus secciones de código y datos de sólo lectura. Se ha observado que se obtienen los mejores resultados con la sección de datos de sólo lectura, seguida de la sección de código y finalmente, el PE completo. Esto indica que la mejor estrategia es analizar cada sección individualmente. También cabe mencionar que las variaciones en la sección de código son muy dependientes de cada programa, y no se ha observado una tendencia general. Por último, se ha observado que los resultados obtenidos en sistemas operativos con arquitectura de 64 bits son especialmente buenos en la sección de código, casi equiparables a la sección de datos de sólo lectura.

Se ha estudiado procesos de sistema, software de usuario y malware. Los resultados obtenidos para la muestra de malware parecen no superar el 50 % de similitud en ningún caso ni con ningún algoritmo, un comportamiento esperable. No obstante, obtener estos resultados con un software malicioso da una buena perspectiva a la utilidad de las técnicas de fuzzy hashing para localizar similitudes en malware. Cabe destacar que los procesos de sistema obtienen resultados excelentes en comparaciones entre procesos de la misma máquina (igual sistema operativo y arquitectura). Esto supone que si un proceso de sistema resulta comprometido, las probabilidades de identificar este comportamiento fuera de lo habitual son muy altas. Por último, se han estudiado similitudes entre distintas versiones de Chrome y Firefox. Se ha observado que estos dos software varían ampliamente entre versiones en todas sus secciones, salvo en los datos de sólo lectura.

6.1. Líneas de trabajo futuro

Debido a la amplitud y diversidad de temas tratados y aprendidos para llevar a cabo este trabajo, no se ha podido profundizar todo lo deseado en algunos aspectos. En este apartado se listan algunas de las líneas de investigación futuras.

- **Independencia de otros plugins en ProcessFuzzyHash.** Como se expone en el Capítulo 3, la herramienta desarrollada depende de dos plugins de Volatility: `procdump` y `memdump`. Sería interesante prescindir de ellos y evitar que el flujo de información necesaria en una ejecución de ProcessFuzzyHash pase por una o más escrituras a disco. De este modo, se lograría un mayor rendimiento general en la generación de hashes.
- **Paralelización completa de la generación de hashes.** Debido a la dependencia expuesta en el punto anterior, es imposible implementar una paralelización completa. Si se lograra prescindir de tales dependencias sería posible paralelizar el proceso completo de generación de hashes, desde la extracción del proceso hasta el cómputo del hash.
- **Evaluación del rendimiento de los algoritmos.** Sería interesante evaluar el rendimiento de los algoritmos en diferentes escenarios, tanto en generación como comparación de hashes.
- **Implementación de más algoritmos de fuzzy hash.** Para este trabajo se han incluido en ProcessFuzzyHash cuatro algoritmos, tres de ellos que ya contaban con una implementación en Python y otro implementado durante durante el desarrollo de la herramienta. Existen algoritmos de fuzzy hash con implementaciones en otros lenguajes, o sin implementación, que sería interesante evaluar.
- **Estudio de resultado con distintas partes de un proceso.** En este trabajo se han analizado resultados comparando hashes del PE de un proceso y sus secciones de código y datos de sólo lectura: `.text` y `.rsrc/.rdata`. Sería interesante analizar otras partes del proceso, como pueden ser todas las páginas de memoria del espacio de direcciones del proceso marcadas para ejecución pero fuera de la sección de código del PE o la sección de carga de DLLs.
- **Ampliar el estudio a más tipos de malware.** Durante este trabajo se han obtenido resultados principalmente de procesos legítimos. Una vez obtenidos estos resultados, sería interesante aplicar las mismas técnicas a una muestra de malware mayor.

Bibliografía

[Aba17] Iñaki Abadía. Volatility Plugin Tutorial. `https://github.com/iAbadia/VolatilityPluginGuide`, August 2017. [Online; accedido 31 de Agosto de 2017].

[AS15] M. Arefkhani and M. Soryani. Malware clustering using image processing hashes. In *2015 9th Iranian Conference on Machine Vision and Image Processing (MVIP)*, pages 214–218, Nov 2015.

[AT17a] AV-TEST. Malware statistics. `https://www.av-test.org/en/statistics/malware/`, 2017. [Online; accedido 26 de Julio de 2017].

[AT17b] AV-TEST. New malware, last 10 years. `https://www.av-test.org/typo3temp/avtestreports/malware-last-10-years_en.png`, 2017. [Online; accedido 7 de Agosto de 2017].

[AT17c] AV-TEST. Total malware, last 10 years. `https://www.av-test.org/typo3temp/avtestreports/malware-last-10-years_sum_en.png`, 2017. [Online; accedido 7 de Agosto de 2017].

[BB12] Frank Breitinger and Harald Baier. A Fuzzy Hashing Approach Based on Random Sequences and Hamming Distance. In *Proceedings of the Conference on Digital Forensics, Security and Law*, pages 89–100, 2012.

[BB13] Frank Breitinger and Harald Baier. Similarity Preserving Hashing: Eligible Properties and a New Algorithm MRSH-v2. In Marcus Rogers and Kathryn C. Seigfried-Spellar, editors, *Digital Forensics and Cyber Crime: 4th International Conference, ICDF2C 2012, Lafayette, IN, USA, October 25-26, 2012, Revised Selected Papers*, pages 167–182, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[BDS03] Eep Bhatkar, Daniel C. Duvarney, and R. Sekar. Address obfuscation: an efficient approach to combat a broad range of memory error exploits. In *Proceedings of the 12th USENIX Security Symposium*, pages 105–120, 2003.

[BM08] Jean-Marie Borello and Ludovic Mé. Code obfuscation techniques for metamorphic viruses. *Journal in Computer Virology*, 4(3):211–220, Aug 2008.

[bri16] bridgeythegeek. My First Volatility Plugin With Unified Output. https://gist.github.com/bridgeythegeek/bf7284d4469b60b8b9b3c4bfd03d051e, 2016. [Online; accedido 27 de Julio de 2017].

[CDD+04] Damiani De Capitani, E. Damiani, S. De, Capitani Vimercati, S. Paraboschi, and P. Samarati. An Open Digest-based Technique for Spam Detection. In *Proceedings of the 2004 International Workshop on Security in Parallel and Distributed Systems*, pages 15–17, 2004.

[Cha02] Moses S. Charikar. Similarity Estimation Techniques from Rounding Algorithms. In *Proceedings of the Thiry-fourth Annual ACM Symposium on Theory of Computing*, STOC '02, pages 380–388, New York, NY, USA, 2002. ACM.

[CJ03] Mihai Christodorescu and Somesh Jha. Static Analysis of Executables to Detect Malicious Patterns. In *Proceedings of the 12th Conference on USENIX Security Symposium - Volume 12*, SSYM'03, pages 12–12, Berkeley, CA, USA, 2003. USENIX Association.

[CT02] Christian S. Collberg and Clark Thomborson. Watermarking, Tamper-Proofing, and Obfuscation – Tools for Software Protection. *IEEE Trans. Soft. Eng.*, 28(8):735–746, August 2002.

[CW08] L. Chen and G. Wang. An Efficient Piecewise Hashing Method for Computer Forensics. In *First International Workshop on Knowledge Discovery and Data Mining (WKDD 2008)*, pages 635–638, Jan 2008.

[dCS99] M. de Cervantes Saavedra. *Don Quijote de la Mancha*. EDAF, 1999.

[Def09] Defense Computer Forensics Lab (DCFL). dcfldd, 2009.

[Dyn] Dynetics. Malware Analysis Tool using Function Level Fuzzy Hashing. `https://github.com/Dynetics/Malfunction`. [Online; accedido 27 de Agosto de 2017].

[Kor06] Jesse Kornblum. Identifying almost identical files using context triggered piecewise hashing. *Digital Investigation*, 3:91 – 97, 2006.

[LCLW14] Michael Hale Ligh, Andrew Case, Jamie Levy, and Aaron Walters. *The Art of Memory Forensics: Detecting Malware and Threats in Windows, Linux, and Mac Memory*. Wiley Publishing, 1st edition, 2014.

[MAE14] V. Gayoso Martínez, F. Hernández Álvarez, and L. Hernández Encinas. State of the Art in Similarity Preserving Hashing Functions. In *Proceedings of the International Conference on Security and Management (SAM); Athens: 1-7. Athens: The Steering Committee of The World Congress in Computer Science, Computer Engineering and Applied Computing (WorldComp)*, 2014.

[MAN14] MANDIANT. Tracking Malware with Import Hashing. https://www.fireeye.com/blog/threat-research/2014/01/tracking-malware-import-hashing.html, 2014. [Online; accedido 26 de Julio de 2017].

[Mar14] Asier Martínez. La «otra manera» de identificar malware. https://www.certsi.es/blog/indicadores-de-compromiso, 2014. [Online; accedido 7 de Agosto de 2017].

[Mica] Microsoft. MSDN: PE Format. https://msdn.microsoft.com/en-us/library/windows/desktop/ms680547(v=vs.85).aspx. [Online; accedido 20 de Agosto de 2017].

[Micb] Microsoft. MSDN: Peering Inside the PE: A Tour of the Win32 Portable Executable File Format. https://msdn.microsoft.com/en-us/library/ms809762.aspx. [Online; accedido 20 de Agosto de 2017].

[Mic15] Trend Micro. TLSH - Trend Micro Locality Sensitive Hash. https://github.com/trendmicro/tlsh, 2015. [Online; accedido 26 de Julio de 2017].

[MJDS07] Gurmeet Singh Manku, Arvind Jain, and Anish Das Sarma. Detecting near-duplicates for web crawling. In *Proceedings of the 16th International Conference on World Wide Web*, WWW '07, pages 141–150, New York, NY, USA, 2007. ACM.

[NMAM+16] A. P. Namanya, Q. K. A. Mirza, H. Al-Mohannadi, I. U. Awan, and J. F. P. Disso. Detection of Malicious Portable Executables Using Evidence Combinational Theory with Fuzzy Hashing. In *2016 IEEE 4th International Conference on Future Internet of Things and Cloud (FiCloud)*, pages 91–98, Aug 2016.

[PaX] PaX Team. PaX address space layout randomization (ASLR). https://pax.grsecurity.net/docs/aslr.txt.

[Phi] PhiBo - DinoTools. ssdeep: Python wrapper for the ssdeep library. https://pypi.python.org/pypi/ssdeep. [Online; accedido 20 de Agosto de 2017].

[Pur16] Olaf Pursche. AV-TEST Security Report 2015/16. https://www.av-test.org/fileadmin/pdf/security_report/AV-TEST_Security_Report_2015-2016.pdf, 2016. [Online; accedido 7 de Agosto de 2017].

[Rad] Radare: Reverse Engineering Framework with focus on UNIX philosophy and full API bindings. http://rada.re/. [Online; accedido 27 de Agosto de 2017].

[Rod] Bernardo Rodrigues. Binwally: Binary and Directory tree comparison tool using Fuzzy Hashing. `https://github.com/bmaia/binwally`. [Online; accedido 27 de Agosto de 2017].

[Rod17] Ricardo J. Rodríguez. Evolution and Characterization of Point-of-Sale RAM Scraping Malware. *Journal in Computer Virology and Hacking Techniques*, 13(3):179–192, August 2017.

[Rou09] V. Roussev. Building a Better Similarity Trap with Statistically Improbable Features. In *2009 42nd Hawaii International Conference on System Sciences*, pages 1–10, Jan 2009.

[RRM07] Vassil Roussev, Golden G. Richard, and Lodovico Marziale. Multi-resolution similarity hashing. *Digital Investigation*, 4:105 – 113, 2007.

[RSI12] Mark E. Russinovich, David A. Solomon, and Alex Ionescu. *Windows Internals, Part 1: Covering Windows Server 2008 R2 and Windows 7*. Microsoft Press, 6th edition, 2012.

[SBAAN16] N. Sarantinos, C. Benzaïd, O. Arabiat, and A. Al-Nemrat. Forensic Malware Analysis: The Value of Fuzzy Hashing Algorithms in Identifying Similarities. In *2016 IEEE Trustcom/BigDataSE/ISPA*, pages 1782–1787, Aug 2016.

[SL07] Caitlin Sadowski and Greg Levin. SimiHash: Hash-based Similarity Detection. University of California, Santa Cruz, 2007.

[SL11] Caitlin Sadowski and Greg Levin. SimiHash: Hash-based Similarity Detection Technical Report UCSC-SOE-11-07. 2011.

[SS13] Murugiah Souppaya and Karen Scarfone. *Guide to Malware Incident Prevention and Handling for Desktops and Laptops*, chapter 4: Malware Incident Response, pages 17–32. NIST, 2013.

[Ton] Stephen Tonkin. fuzzyhashlib: Hashlib-like wrapper for several fuzzy hash algorithms. `https://pypi.python.org/pypi/fuzzyhashlib`. [Online; accedido 20 de Agosto de 2017].

[Tri02] Andrew Tridgell. spamsum README. `https://www.samba.org/ftp/unpacked/junkcode/spamsum/README`, 2002. [Online; accedido 26 de Julio de 2017].

[Wic08] Georg Wicherski. peHash: A Novel Approach to Fast Malware Clustering. `https://www.usenix.org/legacy/event/leet09/tech/full_papers/wicherski/wicherski_html/`, 2008. [Online; accedido 7 de Agosto de 2017].

[WP07] AAron Walters and Nick Petroni. Volatools: Integrating Volatile Memory Forensics into the Digital Investigation Process. Black Hat DC, 2007.

[YZA08] W. Yan, Z. Zhang, and N. Ansari. Revealing Packed Malware. *IEEE Security Privacy*, 6(5):65–69, Sept 2008.

Anexo A

Horas de trabajo

La Figura A.1 contiene un desglose por tarea de las horas empleadas en este trabajo. Por otro lado, la Figura A.2 contiene une diagrama de Gantt ilustrando la planificación final de este trabajo.

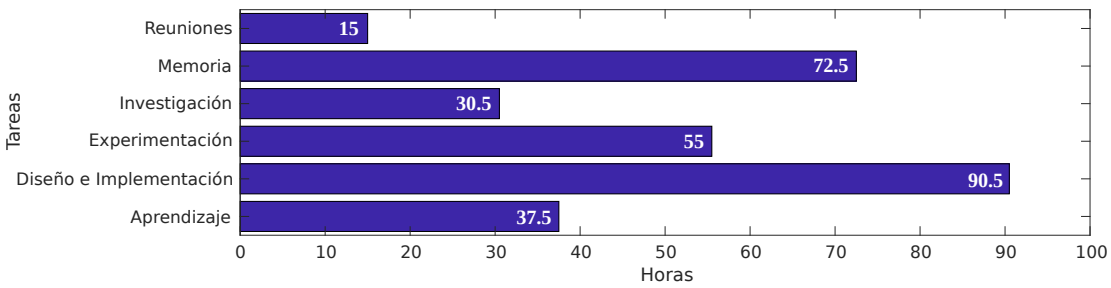


Figura A.1: Desglose de horas empleadas por tarea.

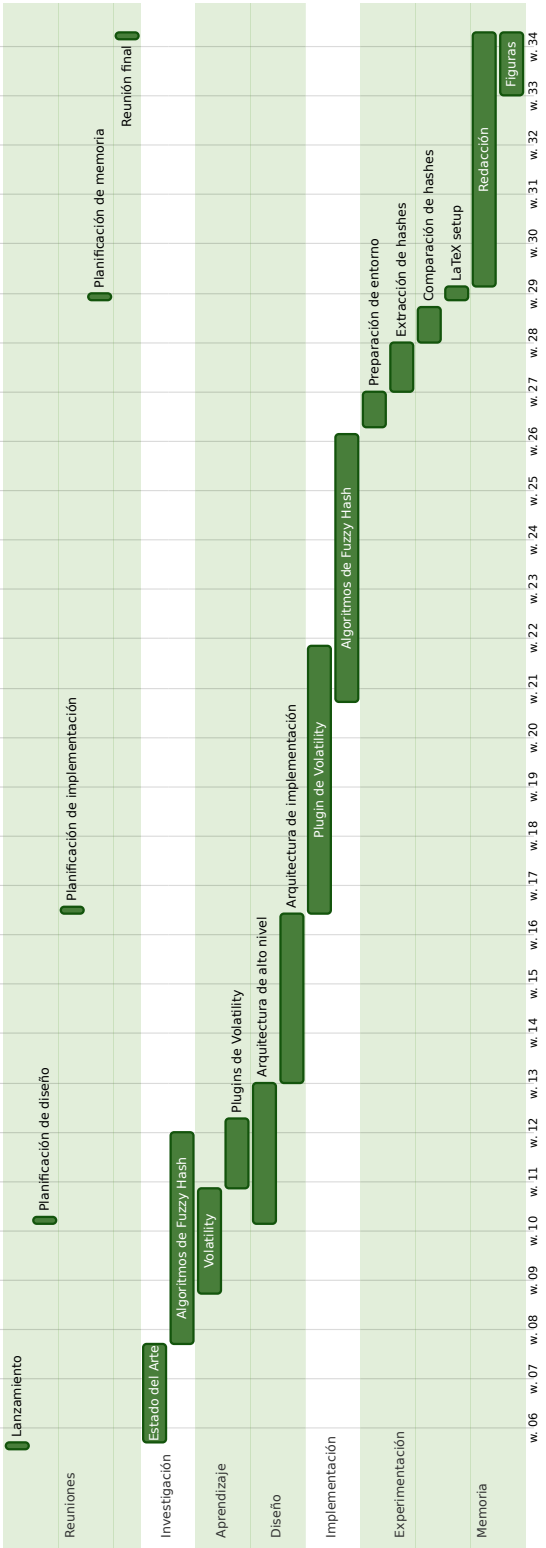


Figura A.2: Diagrama de Gantt del proyecto.

Anexo B

Plugin Volatility

Los aspectos más importantes incluidos en el esqueleto de código se ilustran en la Figura B.1, son los siguientes:

- El plugin debe ser una clase que herede de una de las subclases de *Command* de Volatility. En el caso de ProcessFuzzuHash, se hereda de *AbstractWindowsCommand*.
- Una función `__init__`, que recibirá los parámetros con los que se ha llamado a la instancia de Volatility en ejecución. Si no es necesario procesar los parámetros, esta función puede obviarse.
- Una función `calculate`, que será llamada tras `__init__`, donde ha de realizarse todo el trabajo del plugin (cálculos, llamadas a otros plugins, etc.).
- Una función para representar los resultados. El usuario puede elegir entre texto normal, CSV, JSON, xlsx, dot o incluso un fichero HTML con formato. Existen dos opciones: la función `render_{text,csv,json,html}`, que permitirá presentar los resultados de la forma que se desee según lo indicado por el usuario; y la función `unified_output` que requiere de otra función intermedia, llamada `generator`. Esta última opción fue incluida en las últimas versiones de Volatility y se encuentra recomendada puesto que permite al desarrollador indicar los resultados de la ejecución y delegar a Volatility la tarea de presentarlos correctamente.

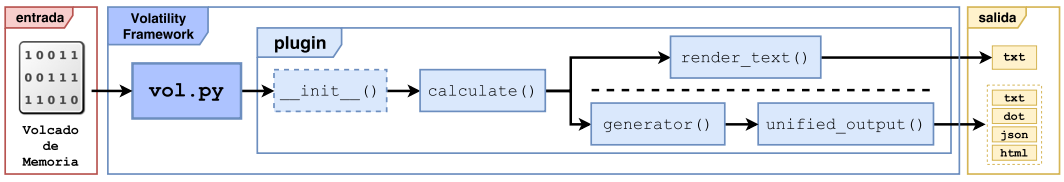


Figura B.1: Funcionamiento básico de un plugin para Volatility.

A continuación, se encuentra el esqueleto de código de un plugin de Volatility:

```
import volatility.plugins.common as common
import volatility.utils as utils
import volatility.win32 as win32

from volatility.renderers import TreeGrid
from volatility.renderers.basic import Address, Hex

class MyPlugin(common.AbstractWindowsCommand):

        def calculate(self):
                addr_space = utils.load_as(self._config)
                tasks = win32.tasks.pslist(addr_space)
                return tasks

        def generator(self, data):
                for task in data:
                        yield (0, [
                                int(task.UniqueProcessId),
                                str(task.CreateTime),
                                str(task.ImageFileName)
                        ])

        def unified_output(self, data):
                return TreeGrid([
                                ("PID", int),
                                ("Created", str),
                                ("Image", str)],
                                self.generator(data))
```