P. Pérez Jiménez, J. A. Onieva, and G. Fernandez, "CCBHash (Compound Code Block Hash) para Análisis de Malware", *XVII Reunión Española sobre Criptología y Seguridad de la Información*, pp. 168-173, 2022.
NICS Lab. Publications: https://www.nics.uma.es/publications

CCBHash (Compound Code Block Hash) para Análisis de Malware

Pablo Pérez Jiménez
NICS Lab, Málaga
ppj@lcc.uma.es

José Antonio Onieva González
NICS Lab, Málaga
onieva@uma.es

Gerardo Fernández
VirusTotal, Málaga
gerardofn@virustotal.com

***Resumen*—En estos últimos años, el análisis de malware ha adquirido una importancia cada vez mayor debido al aumento de ataques informáticos, cada vez más sofisticados. Uno de los objetivos que tiene esta rama de la ciberseguridad es encontrar similitudes entre distintos ficheros, permitiendo así detectar y clasificar malware e incluso, en algunos casos, realizar atribuciones. En este trabajo desarrollaremos un fuzzy hash capaz de caracterizar el malware generando una firma fácilmente comparable y almacenable de sus funciones. Ya que nuestra meta es poder detectar estas similitudes en grandes cantidades de datos en un periodo de tiempo razonable, el tamaño del hash debe ser limitado a la vez que guarde la máxima información posible.**

***Index Terms*—análisis de malware, fuzzy hashing, similitud**

I. Introducción

Encontrar similitudes entre distintos ficheros es un problema común estudiado desde hace más de una década. En los últimos años el problema ha cobrado importancia para la búsqueda de similitudes entre ficheros maliciosos. Si bien es cierto que el objetivo es el mismo, también lo es que la definición del concepto de similitud puede diferir en función del campo en el que esté siendo estudiado. Cuando nos encontramos en el análisis de malware, la palabra similitud puede tomar dos significados. Dos ficheros pueden ser similares cuando el resultado de su ejecución es el mismo, o bien, cuando éste es distinto pero se usan funciones iguales o parecidas en el proceso.

En este trabajo, no olvidaremos ninguno de los dos enfoques anteriormente comentados. El objetivo es poder identificar similitudes entre una gran cantidad de ficheros maliciosos (del orden de petabytes), no solo a nivel de muestra, sino a nivel de segmento de código ensamblador. Al encontrarnos con petabytes de muestras, estas deben estar almacenadas de forma que se permita realizar una rápida comparación entre ellas. Para ello, debemos ser capaces de caracterizar una muestra completa de malware dividiéndola en trozos más pequeños de código ensamblador y generando un hash *compuesto* para cada uno de ellos. Dichos hashes se unirán posteriormente en un solo hash que identificará a la muestra completa, tal y como hacen los fuzzy hashes que ya conocemos [1]. A este fuzzy hash lo llamaremos CCBHash *(Compound Code Block Hash)*. Se debe tener en cuenta que estos hashes son calculados de manera offline, es decir, una vez que se ha obtenido el CCBHash, se almacena para su posterior uso. Esto permite que, aún siendo importante, el tiempo del cálculo del CCBHash no sea un factor totalmente limitante en nuestra herramienta.

Los hash tradicionales tienen como objetivo identificar inequívocamente a una muestra, utilizando un tamaño mucho menor que el original y con las mínimas colisiones posibles [2]. De esta manera, dos ficheros que sean exactamente iguales excepto en un bit darán lugar a dos hashes totalmente diferentes. Las funciones de fuzzy hashing tratan de conseguir lo contrario. Es decir, resumen una muestra en una firma relativamente pequeña pero, si dos ficheros únicamente cambian en un bit, los hashes generados deben ser (casi) idénticos, permitiendo así detectar similitudes a partir del hash.

Para poder encontrar similitudes en bloques de código ensamblador, hay que definir el tamaño de estos bloques o, mejor dicho, el inicio y fin de cada bloque. Dado que nuestro objetivo es almacenar los hashes resultantes en una base de datos, calcular las firmas para todos los bloques posibles sería algo inviable (tanto por espacio como por tiempo de la comparación). Por ello, CCBHash usará como bloques las propias funciones en ensamblador, quedando así acotado el número de bloques posibles.

El problema de la búsqueda de similitudes en malware es tan importante y actual que empresas como VirusTotal ofrecen servicios para ello usando diferentes métricas [3]. Por un lado, existen herramientas que buscan diferencias en segmentos de código ensamblador entre dos ficheros, como BinDiff [4] y Diaphora [5]. Por otra parte, hay numerosos estudios y soluciones que se enfocan en las similitudes entre varias muestras completas de malware. Sin embargo, las alternativas para buscar similitudes a nivel de bloque de código ensamblador, o funciones en nuestro caso, entre un número muy grande de muestras de malware escasean. Algunas de estas opciones, en concreto los fuzzy hashes como SSDEEP [6], consiguen tiempos de ejecución aceptables pero deben encontrar coincidencias exactas. Es decir, dividen una muestra en bloques más pequeños que deben ser iguales para poder detectar similitudes. A diferencia de SSDEEP, en CCBHash el hash de cada bloque de código será a su vez un fuzzy hash compuesto por atributos de este en lugar de un simple hash del mismo, permitiendo encontrar similitudes entre diferentes bloques y no solo coincidencias exactas.

Resumidamente, nuestro objetivo es combinar dos de estas ideas: utilizar tanta información del código como sea posible, como hacen BinDiff y Diaphora, y combinarla en un fuzzy hash como hace SSDEEP, con la diferencia de que cada componente del fuzzy hash será a su vez otro fuzzy hash en lugar de un hash tradicional. De esta forma, conseguiríamos un análisis de similitud lo más preciso y rápido posible sobre petabytes de ficheros.

II. Estado del arte

Antes de comenzar con la solución propuesta, llevaremos a cabo un estudio del estado del arte sobre la búsqueda de similitudes en malware. Son muchas las técnicas utilizadas; sin embargo, teniendo en cuenta nuestro objetivo, hay varias que parecen destacar y que comentaremos más detalladamente.

II-A. Estrategia basada en n-grams

Hay una gran cantidad de artículos y soluciones en el mercado que hacen uso de los n-grams para comparar distintos ficheros. Los n-grams no son más que ventanas deslizantes de tamaño variable, n en este caso, donde el objetivo es almacenar todas las ventanas posibles de n elementos para su posterior comparación. Estos elementos pueden ser muy diversos dependiendo del contexto y el campo donde se quieran utilizar. En cuanto a malware se refiere, estos elementos suelen ser bytes o instrucciones en ensamblador, permitiendo una comparación de más alto nivel de abstracción.

Sin embargo, muchos de los estudios existentes han dado un paso más y no tratan de usar únicamente bytes o instrucciones exactas, si no que hacen un pequeño análisis de la entrada filtrando aquellas características más relevantes. Por ejemplo, hay artículos [7] donde los elementos usados para los n-grams son los *opcodes* de las instrucciones en ensamblador, obviando los registros y operandos utilizados. Este método permite realizar una comparación centrándose en la funcionalidad del malware, haciendo que las características más variables entre distintas muestras no se tengan en cuenta.

Si bien la solución anterior puede ser interesante en muchos contextos, también presenta algunas debilidades. Una de las principales desventajas es la exactitud que requiere para poder encontrar similitudes. Lo entenderemos mejor con dos ejemplos. Como sabemos, en muchas ocasiones el orden de ejecución de un grupo de instrucciones no importa. Por ejemplo, al realizar una suma y una resta consecutiva por una constante el resultado será el mismo independientemente del orden en el que se realicen. De la misma forma, hay varios *opcodes* en código ensamblador que se utilizan para realizar las mismas operaciones, pudiendo conseguir objetivos idénticos con distintas instrucciones.

Para solucionar algunos de los problemas existentes con los n-grams de *opcodes*, hay estudios [8] que introducen el concepto de n-perms. Los n-perms no son más que una versión de los n-grams usando permutaciones. De esta forma, si hay varios n-grams idénticos almacenados, los repetidos se eliminan dejando un solo n-perm de cada clase. Además, los n-perms son insensibles al orden de ejecución, por lo que dos n-grams que contengan los mismos elementos en distinto orden serán tratados como iguales.

Usar n-perms provoca una mayor detección de similitudes para ficheros que usan los mismos *opcodes*. Sin embargo, este mayor nivel de abstracción puede dar lugar también a un mayor número de falsos positivos (FP) donde dos ficheros con funcionalidades totalmente distintas usen instrucciones similares. Llegamos de esta forma a un punto donde debemos decidir entre dos modelos, uno de ellos donde hay una detección menor pero más exacta acerca de los positivos, y otro donde la detección es mayor pero puede que la tasa de falsos positivos aumente.

II-B. Estrategia basada en grafos

Los Control Flow Graphs (CFG), son grafos de flujo de control. En dichos grafos se almacena el flujo de ejecución que sigue un determinado ejecutable, generalmente a nivel de código ensamblador. Para entender mejor el grafo resultante podemos hacer uso de la herramienta IDA Pro [9], donde se usa este concepto para la representación de las muestras. En la Figura 1 observamos un pequeño ejemplo de un CFG con código ensamblador.

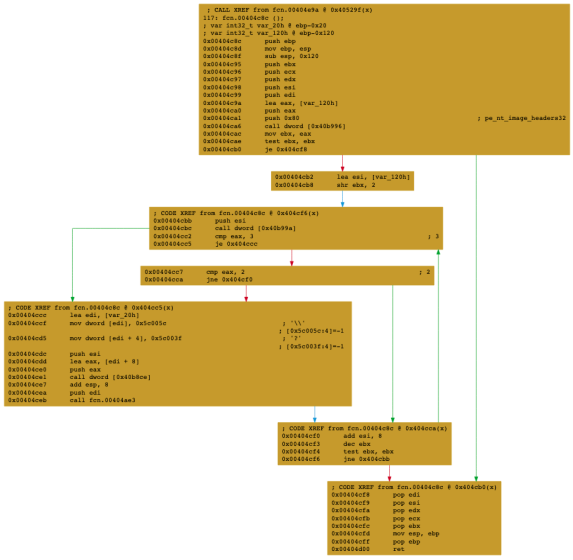

Figura 1. CFG con código ensamblador.

Como ocurre en todos los grafos, tenemos dos tipos de elementos: los nodos y las aristas, en este caso, dirigidas. Las aristas guardan la información acerca de los saltos existentes en la ejecución. De esta forma, cada salto, condicional o incondicional, se verá reflejado con una o varias aristas. En cuanto a los nodos, algunos estudios [10] prácticamente los dejan vacíos, dándole toda la importancia al flujo en sí. Sin embargo, en otros trabajos se usan dichos nodos para almacenar información mucho más diversa.

En [11] se propone el uso de atributos para caracterizar cada nodo, dando lugar al Attributed CFG (ACFG). Este enfoque trata de coger los nodos que genera una herramienta como IDA Pro y crear una firma basada en un vector de atributos para cada uno de ellos. Esta idea puede ser muy interesante si hacemos un pequeño estudio sobre qué atributos pueden caracterizar un bloque de código. Los atributos utilizados en este caso son los que se recogen en la Tabla I.

El uso de ACFG es muy interesante ya que permite almacenar información muy diversa en los nodos a la vez que permite combinar dicha estrategia con otras. Por ejemplo, en [12] se propone usar n-grams del CFG, donde los elementos sobre los que avanza la ventana son los nodos, que en este caso se caracterizan por el número de aristas entrantes y salientes, cubriendo todos los caminos posibles de n nodos consecutivos siguiendo el flujo establecido. Incluso un CFG se puede usar simplemente para ordenar el código y luego tomar n-grams de *opcodes*.

Otros autores [13] tratan de combinar los CFG con las llamadas a la API del subsistema del sistema operativo, creando

Tabla I
ALGUNOS ATRIBUTOS PARA CARACTERIZAR NODOS EN LOS CFG.

Tipo de atributo	Descripción
De la secuencia de código	# Constantes numéricas # Instrucciones de transferencia # Instrucciones de llamada # Instrucciones aritméticas # Instrucciones de comparación # Instrucciones de movimiento # Instrucciones de terminación # Instrucciones de declaración de datos # Instrucciones totales
De la estructura de los vértices	# Descendencia/hijos # Instrucciones en el vértice

un grafo donde los nodos son caracterizados por las llamadas realizadas a la API. Este método, aunque con ventajas para ejecutables no maliciosos, presenta serios inconvenientes si tenemos en cuenta que hay malware creado específicamente para no usar llamadas a la API del subsistema.

Por último, hay trabajos como [14], [15] y [16] que también hacen uso de los Data Flow Graph (DFG) o Program Dependence Graph (PDG), donde se almacena un grafo sobre la dependencia de los datos, es decir, de los registros del procesador utilizados.

Estos enfoques, aunque presentan grandes resultados en cuanto a la búsqueda de similitud, también tienen algunas desventajas. La principal de ellas es el tiempo que requieren. Si bien la estrategia basada en n-grams presenta un coste computacional de $O(n)$, los enfoques basados en CFG aumentan hasta $O(n^3)$ [17], algo que para algunos escenarios puede ser inviable.

II-C. Estrategia basada en Fuzzy Hashing

Existe una amplia cantidad de trabajos que estudia la utilidad y la precisión de las estrategias anteriormente mencionadas [18]. Sin embargo, hay otro tipo de enfoques con características muy interesantes, como son los fuzzy hashes.

Muchas herramientas conocidas emplean los fuzzy hashes para encontrar similitudes entre distintos ficheros. Sin embargo, ninguno de estos hashes es totalmente eficaz ya que depende mucho de la naturaleza de la muestra que esté siendo tratada. En VirusTotal podemos encontrar numerosas muestras donde, según el tipo de fuzzy hash utilizado, el porcentaje de similitud varía drásticamente.

Existen diferentes tipos de algoritmos de Fuzzy Hashing, entre los que destacan [19]:

- BBH (Block Based Hashing). Genera un hash para cada bloque de tamaño fijo de los datos de entrada y realiza la comparación entre los hashes de cada bloque. Cuanto más grandes sean los datos de entrada, mayor será el hash. El algoritmo dcfldd[1] by Harbour es un ejemplo de ello.
- CTPH (Context Triggered Piecewise Hashing). CTPH es muy similar a BBH pero usando trigger points en lugar de bloques de tamaño fijo. El problema del uso de trigger points es que se determinan en función del tamaño de los datos de entrada, lo que puede hacer que dos ficheros similares pero con tamaños muy diferentes tengan hashes totalmente distintos. El algoritmo SSDEEP pertenece a este grupo.
- SIF (Statistically Improbable Features). Trata de localizar un conjunto de características compuestas por secuencias de bits poco habituales, haciendo uso de la entropía, creando posteriormente un hash de dichas características. El algoritmo más conocido es sdhash [20].
- BBR (Block Based Rebuilding). Hace uso de datos externos a la propia entrada del algoritmo, que pueden ser aleatorios o constantes. Por ejemplo, a cada byte del fichero de entrada se le asigna 0xFF o 0x00 tras compararlo con sus bits vecinos. Si la mayoría de estos vecinos son 1, se convierte en 0x00, si no, en 0xFF. En general, la comparación se realiza mediante el cálculo de la distancia de Hamming. Un ejemplo de dicho grupo es el algoritmo mvHash-B [21].
- LSH (Locality Sensitive Hashing). Reduce el espacio de posibles entradas a una cantidad discreta de probabilidades, maximizando la probabilidad de que dos entradas similares generen una salida casi idéntica. Las técnicas utilizadas están relacionadas con la búsqueda de vecino más cercano.

Entre las herramientas analizadas en este campo hay dos que llaman especialmente nuestra atención, que son Machoc [22] y Machoke [23]. Ambas son muy similares en cuanto a su funcionalidad pero difieren ligeramente en la tecnología utilizada. Dichas herramientas obtienen el CFG de cada una de las funciones de un ejecutable, tal y como vemos en IDA Pro (de hecho, Machoke puede ser utilizada como plugin de esta), calculando para cada una un código hash basado en su CFG y concatenando los hashes para formar un fuzzy hash.

II-D. BinDiff y Diaphora

Por último, es necesario hacer especial énfasis en dos herramientas ya mencionadas: BinDiff y Diaphora. Su objetivo es encontrar diferencias entre dos ficheros. Para ello, se centran en primer lugar en las funciones en código ensamblador. Luego, realizan una comparación entre las distintas funciones, tanto a nivel de atributos de función como de segmento de código dentro de la función. En este caso, los bloques consisten en cada uno de los nodos del CFG de la función. Por último, da un porcentaje de precisión y confianza a las similitudes encontradas, en función de los atributos que hayan coincidido en la comparación.

Estas herramientas usan atributos muy diversos para encontrar similitudes entre las funciones de dos ficheros, muchos de ellos a partir de las ideas extraídas de los apartados anteriores. Entre los atributos destacan: distintos grafos generados a partir de la propia función, el nombre de la función, las cadenas de caracteres existentes, las instrucciones de cada bloque de código, etc. El uso de tantas heurísticas hace que la comparación llevada a cabo sea bastante precisa y costosa. Sin embargo, esto puede hacerse gracias a que únicamente se están comparando dos muestras. Si el número de ficheros creciese, este enfoque sería inviable debido al tiempo que sería necesario.

Nuestra herramienta trata de conseguir un resultado similar al de BinDiff y Diaphora, pero realizando la comparación entre una cantidad enorme (con un total de almacenamiento del orden de petabytes) de funciones, que pasan a ser nuestro

[1] http://dcfldd.sourceforge.net/

componente principal de trabajo en la búsqueda de similitud. Por ello, reunirá varios de estos atributos, compactando cada atributo en un hash de tamaño acotado, que se unirán formando un sola firma para cada función. De esta forma, al saber el tamaño ocupado (o número de bytes) por cada atributo, será posible comparar cada uno de forma rápida e independiente.

III. Solución propuesta

III-A. Marco teórico

Lo primero que debemos tener en cuenta, y que hemos comentado anteriormente, es que nuestro objetivo es comparar funciones de código ensamblador entre una ingente cantidad de muestras. Es decir, para cada muestra necesitamos almacenar un código hash en el que se recoja información acerca de las diferentes funciones. La complicación se encuentra en cómo caracterizar las distintas funciones antes de utilizar algoritmos de hashing sobre ellas. Estas funciones las podemos obtener con herramientas como r2pipe de radare2 [24] o IDA Pro.

A continuación, es el momento de indicar cómo caracterizar a cada función. Lo haremos creando una firma para cada una de ellas, que estará compuesta por una serie de atributos a los que se les hará un hash (individualmente) y que serán concatenados en un único hash. Es decir, el hash de una muestra estará compuesto por hashes de funciones que a su vez estarán compuestos por hashes de atributos, que será la unidad mínima de comparación. Procedemos a seleccionar los atributos que formarán la firma de nuestra función. En primer lugar, seleccionaremos algunos atributos de más alto nivel, que traten a la función como una caja negra. Téngase en cuenta que la justificación para su selección es similar en todos ellos: valores similares o muy parecidos entre dos funciones distintas, indican comportamientos similares.

- Nombre de la función. Si bien los nombres asignados por un desensamblador no son relevantes, la existencia de información de debug en el ejecutable podría hacer que este atributo adquiera suficiente importancia.
- Tamaño de la función en bytes.
- Número de entradas y salidas. Proporciona un grado de utilización de la función dentro del malware así como de la necesidad de ésta por parte de otras.
- Número de bloques. Refleja un punto intermedio entre la cantidad de instrucciones y los saltos existentes.
- La complejidad ciclomática. Entendida como $A-N+2C$ siendo A el número de aristas, N el número de nodos y C el número de nodos de salida.
- El propio CFG de la función. Para el cálculo del CFG se ha seguido un procedimiento similar al de [23].
- Número de variables locales.
- Número de argumentos de la función.

En segundo y último lugar, si bajamos el nivel de abstracción y nos fijamos en los bloques de código ensamblador (que observamos en IDA Pro) que componen la función, tenemos atributos como:

- Número de instrucciones.
- Tipo de instrucciones (y cantidad de cada tipo).

La elección de los atributos mencionados ha tenido en cuenta dos enfoques. El primero de ellos trata de encontrar código que actúe de forma similar, es decir, que realice operaciones similares. El segundo, y no menos importante, es detectar código que realice las operaciones de la misma forma.

Una vez que ya hemos reunido los atributos necesarios, es el momento de calcular el fuzzy hash de la función. Para ello, calcularemos un hash para cada uno de los atributos. El tamaño del hash calculado para cada atributo debe ser lo más pequeño posible a la vez que la probabilidad de colisiones sea lo suficientemente pequeña. En nuestro caso, tras diversas pruebas, el tamaño escogido ha sido de dos bytes, obteniendo finalmente un fuzzy hash de tamaño $2n$ bytes, siendo n el número de atributos. En la Figura 2 podemos observar cómo sería el resultado si usáramos cinco atributos.

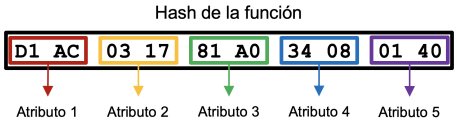

Figura 2. Hash de la función compuesto de hashes de los atributos.

CCBHash quedará limitado principalmente por esta elección de atributos llevada a cabo. Cuantos más atributos sean seleccionados mejor será la comparación, en detrimento de aumentar el tamaño del hash y el tiempo de comparación. Lo mismo ocurre si aumentamos el espacio asignado a cada uno de ellos. De esta forma, se ha debido llegar a un equilibrio entre calidad, espacio y rapidez.

Dado que el tamaño del hash de cada atributo es conocido, al comparar funciones de distintas muestras es posible hacerlo por atributo. De esta forma, dados los fuzzy hashes de dos funciones, podemos comparar paralelamente la parte correspondiente a cada atributo. Finalmente, sabríamos qué atributos han presentado coincidencias, pudiendo determinar un porcentaje de similitud en función del número de atributos iguales, así cómo la calidad del resultado en función de dichos atributos ya que hay heurísticas más fuertes que otras. En función del escenario donde se use este fuzzy hash, el criterio para considerar iguales dos funciones puede ser distinto. Un criterio aceptable sería que dos funciones son similares si al menos la mitad de los atributos coinciden. Aunque este valor es parametrizable y dependerá del contexto de uso de CCBHash. En la Figura 3 podemos ver un ejemplo con tres atributos.

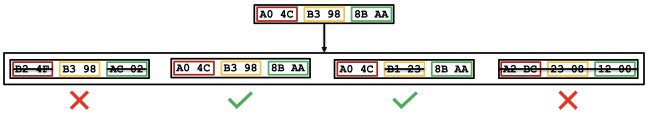

Figura 3. Buscando similitudes entre hashes de funciones.

Estableceremos, sin embargo, un sistema donde se ponderen los atributos en función de su importancia. De esta forma, heurísticas como el CFG o el tipo de instrucciones usadas tendrán más importancia que atributos como el nombre de la función o su tamaño (ya que estos son atributos fácilmente modificables y débiles en su grado de representatividad). En la Tabla II podemos ver como quedaría la puntuación según el atributo correspondiente. Una similitud perfecta daría como resultado 1 (100 %), mientras que dos funciones con ninguna

similitud tendría como resultado 0. Para nosotros, un resultado mayor que 0.7 (70 %) indicará que ambas funciones son similares. Este porcentaje se ha obtenido de manera empírica, y como explicamos en la sección IV, no es definitivo.

Tabla II
PUNTUACIONES SEGÚN ATRIBUTO.

Atributo	Puntuación
Nombre de la función	0.01
Tamaño de la función	0.05
Número de entradas	0.04
Número de salidas	0.04
Número de variables locales	0.04
Número de argumentos de la función	0.04
Número de instrucciones	0.10
Complejidad ciclomática	0.14
Número de bloques	0.04
Control Flow Graph	0.25
Tipo de instrucciones	0.25

Por último, solo queda calcular el CCBHash. Una vez que tengamos los hashes de cada función, los concatenamos y ya tenemos como resultado nuestro CCBHash. Dado que el tamaño del hash de cada función es conocido, al igual que lo era el de los atributos, se pueden comparar distintas funciones de manera simultánea. De esta manera, el CCBHash será de tamaño variable, ya que depende del número de funciones. En concreto, el tamaño de este es de $n \times f$ bytes, siendo n el número de funciones y f el tamaño del hash de la función, que será constante.

III-B. Marco práctico

Para terminar, solo quedaría poner en práctica el diseño preliminar de nuestra herramienta. Para ello, vamos a utilizar muestras de distintas familias de malware, como DarkSide o WannaCry, para ver las similitudes que se encuentran con muestras de una misma familia y con muestras de distintas familias.

Para abordar el problema crearemos un script en Python, ayudándonos de la librería r2pipe que nos permite desensamblar una muestra y obtener información muy útil acerca de ella. Gracias a dicha herramienta, obtenemos las funciones del ejecutable, los bloques que forman cada función y las instrucciones que forman cada bloque, así como información asociada a cada uno de estos elementos. Para crear los hashes, usaremos la librería hashlib de Python [25], en concreto el algoritmo blake2b [26] ya que permite generar hashes de tamaño arbitrario rápidamente, superando en velocidad a algoritmos como MD5, SHA-1, SHA-2 o SHA-3.

En un MacBook Pro con procesador 2.9 GHz Intel Core i5, CCBHash tarda una media de 30 milisegundos por función para ejecutables de entre 10 KB y 4 MB. Sin embargo, de esos 30 ms, solo 0.5 ms equivalen a la aplicación del algoritmo blake2b y la concatenación de los hashes de cada atributo, y el resto a la obtención de las funciones y sus atributos con ayuda de la librería r2pipe.

En primer lugar, vemos los resultados con varias muestras de DarkSide. Obtenemos el fuzzy hash propuesto para las funciones de las muestras y buscamos similitudes. Atendiendo a dos muestras cualesquiera, por ejemplo las muestras con hashes 9CEE...7297 y AFB2...8178, y comparando el CCBHash obtenido, observamos que el 75 % de las funciones tiene más de un 50 % de similitud y hay un 56 % de funciones con más de un 90 % de similitud.

Probamos de nuevo con varias muestras de otra familia, en este caso WannaCry. Atendiendo nuevamente a dos muestras, con hashes 4186...D982 y 09A4...CAFA, observamos que el 52 % de las funciones tienen una similitud superior al 50 % y hay un 31 % de funciones con más de un 90 % de similitud. Repitiendo el proceso con distintas muestras de una misma familia obtenemos resultados similares, obteniendo siempre porcentajes superiores al 30 % de las funciones para similitudes del 70 %.

Por último, vemos qué ocurre con muestras de distintas familias, por ejemplo 9CEE...7297 de DarkSide y 4186...D982 de WannaCry. En este caso, obtenemos que el porcentaje de funciones con más de un 50 % de similitud es del 22 %, y si la similitud sube al 90 % el porcentaje de funciones ya baja al 4 %. En concreto, este 4 % equivale a tres funciones. Estas corresponden a funciones donde la única instrucción ejecutada sirve para llamar a la API de Windows. Podemos decir, según el criterio comentado anteriormente, que entre estas dos muestras ha habido un 4 % de falsos positivos. Nuevamente estudiando distintas muestras de familias diferentes obtenemos resultados similares, no superando el 10 % de falsos positivos (y en su mayor parte debido a funciones poco interesantes donde la única función es una llamada a la API).

En la Figura 4 vemos una gráfica del caso anterior del número de funciones que presenta un porcentaje de similitud concreto. Los altos porcentajes de similitud, superiores al 75 %, se deben a funciones casi idénticas donde únicamente cambia el nombre, número de variables locales, número de instrucciones o número de entradas y salidas. En la zona central, con similitudes en torno al 50 %, aparecen funciones donde alguno de los atributos más fuertes, como el CFG o los tipos de instrucciones, no coincide. Por último, tenemos aquellas funciones con similitudes inferiores al 25 %, donde los atributos con puntuaciones altas no coinciden y cuya similitud se debe a simples coincidencias.

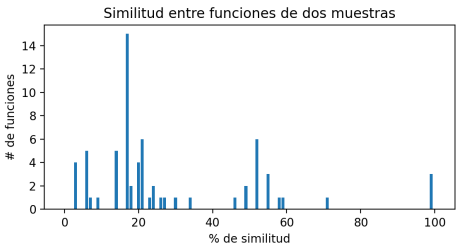

Figura 4. Similitud entre las funciones de dos muestras.

En cuanto a falsos negativos (FN), las pruebas realizadas hasta el momento han arrojado resultados prometedores: no se han encontrado falsos negativos. Es decir, no hemos encontrado muestras de la misma familia en la que no se encuentren similitudes. Como comentamos anteriormente, para nuestra herramienta es prioritario no dejar escapar ninguna similitud (por ello el uso de métricas débiles), aunque ello conlleve aceptar más falsos positivos en algunos escenarios. Es necesario, no obstante, fortalecer el diseño y realizar un número de pruebas estadísticamente relevante.

IV. Conclusiones y trabajo futuro

Con este proyecto conseguimos una propuesta inicial para encontrar similitudes entre funciones de una cantidad ingente de muestras de malware. Esta herramienta, en su estado actual, podría utilizarse como un fuzzy hash más, complementando a los actuales y pudiendo encontrar similitudes que otros no encontrarían. Sin embargo, esta idea puede mejorarse. En especial en lo referente a la medición exhaustiva de resultados con un conjunto de muestras malware suficiente y una comparación detallada con respecto a los métodos presentados en la sección I en cuanto a los FN y FP.

En primer lugar, se está realizando un estudio sobre qué funciones son conocidamente no maliciosas o poco interesantes (eg. funciones con una o dos instrucciones como hemos observado en III-B). Con ello queremos incluir un filtro donde dichas funciones se obvien en el cálculo del CCBHash, ahorrando tiempo y espacio. Con objeto de maximizar la eficiencia, estamos estudiando distintas posibilidades con las que extraer los atributos de las funciones, ya que la librería r2pipe consumía la mayor parte del tiempo en el cálculo de nuestro hash.

También hemos comenzado un proceso de elección de atributos más potentes. Si bien los atributos usados actualmente detectan con éxito similitud entre funciones de malware, creemos necesario seguir investigando en este campo para encontrar aquellos atributos óptimos que encuentren cualquier tipo de similitud.

En relación a lo anterior, actualmente el hash creado tiene una longitud fija, donde cada característica de la función ocupa un tamaño fijo. En el futuro nos gustaría mejorar esta propuesta haciendo que cada atributo ocupe un tamaño que pueda diferir del resto de atributos. Incluso almacenando para cada atributo distintas formas de representación, como un valor numérico en función de un rango en lugar de un simple hash. Esto haría nuestro fuzzy hash más flexible, pero no debemos perder el contexto de ejecución de CCBHash (petabytes de funciones).

Creemos así que la herramienta desarrollada tiene gran utilidad en la detección de similitudes en malware, aportando características que no es posible encontrar con las opciones disponibles actualmente. Además, demuestra tener gran potencial y que, ahondando en su investigación, daría lugar a un nuevo fuzzy hash que podría no solo servir de complemento a otras herramientas, sino llegar a sustituir a algunas de las actuales.

Agradecimientos

Este trabajo ha sido realizado gracias a la colaboración de la Universidad de Málaga (UMA) y VirusTotal bajo el amparo del proyecto SAVE cofinanciado por la Junta de Andalucía (PAIDI 2020) y el Fondo Europeo de Desarrollo Regional (FEDER).

Referencias

[1] I. Abadía, "Evaluación de algoritmos de fuzzy hashing para similitud entre procesos," 2017, trabajo de Fin de Grado. [Online]. Available: http://webdiis.unizar.es/~ricardo/files/PFCs-TFGs/Fuzzy-Hashing-Procesos/Fuzzy-Hashing-Procesos.pdf

[2] M. Singh and D. Garg, "Choosing Best Hashing Strategies and Hash Functions," in *2009 IEEE International Advance Computing Conference*. IEEE, mar 2009, pp. 50–55. [Online]. Available: http://ieeexplore.ieee.org/document/4808979/

[3] V. Díaz, "Why is similarity so relevant when investigating attacks." [Online]. Available: https://blog.virustotal.com/2020/11/why-is-similarity-so-relevant-when.html

[4] "BinDiff de Zynamics." [Online]. Available: https://www.zynamics.com/bindiff.html

[5] "Repositorio de diaphora." [Online]. Available: https://github.com/joxeankoret/diaphora

[6] J. Kornblum, "Identifying almost identical files using context triggered piecewise hashing," *Digital Investigation*, vol. 3, pp. 91–97, 2006, the Proceedings of the 6th Annual Digital Forensic Research Workshop (DFRWS '06). [Online]. Available: https://www.sciencedirect.com/science/article/pii/S1742287606000764

[7] L. Liu, B.-s. Wang, B. Yu, and Q.-x. Zhong, "Automatic malware classification and new malware detection using machine learning," *Frontiers of Information Technology & Electronic Engineering*, vol. 18, no. 9, pp. 1336–1347, sep 2017. [Online]. Available: http://link.springer.com/10.1631/FITEE.1601325

[8] A. Walenstein, M. Venable, M. Hayes, C. Thompson, and A. Lakhotia, "A.: Exploiting similarity between variants to defeat malware: "vilo" method for comparing and searching binary programs," in *In: Proceedings of BlackHat DC 2007. (2007) https: //blackhat.com/presentations/bh-dc-07/Walenstein/Paper/bh-dc-07-walenstein-WP.pdf.*

[9] Hex-Rays, "Ida pro." [Online]. Available: https://hex-rays.com/ida-pro

[10] G. Bonfante, M. Kaczmarek, and J. yves Marion, "Control flow graphs as malware signatures." [Online]. Available: https://hal.inria.fr/inria-00176235/document

[11] J. Yan, G. Yan, and D. Jin, "Classifying Malware Represented as Control Flow Graphs using Deep Graph Convolutional Neural Network," in *2019 49th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*. IEEE, jun 2019, pp. 52–63. [Online]. Available: https://ieeexplore.ieee.org/document/8809504/

[12] Y. Li, J. Jang, and X. Ou, "Topology-Aware Hashing for Effective Control Flow Graph Similarity Analysis," no. December, pp. 1–6, apr 2020. [Online]. Available: http://arxiv.org/abs/2004.06563

[13] J. Bergeron, M. Debbabi, J. Desharnais, M. M. Erhioui, Y. Lavoie, and N. Tawbi, "Static detection of malicious code in executable programs," *Int. J. of Req. Eng*, 2001.

[14] Mayrand, Leblanc, and Merlo, "Experiment on the automatic detection of function clones in a software system using metrics," in *1996 Proceedings of International Conference on Software Maintenance*, 1996, pp. 244–253. [Online]. Available: https://ieeexplore.ieee.org/stamp/stamp.jsp?tp=\&arnumber=565012

[15] J. Kim, H. Choi, H. Yun, and B.-R. Moon, "Measuring Source Code Similarity by Finding Similar Subgraph with an Incremental Genetic Algorithm," in *Proceedings of the Genetic and Evolutionary Computation Conference 2016*. New York, NY, USA: ACM, jul 2016, pp. 925–932. [Online]. Available: https://dl.acm.org/doi/10.1145/2908812.2908870

[16] J. Krinke, "Identifying similar code with program dependence graphs," in *Proceedings Eighth Working Conference on Reverse Engineering*. IEEE Comput. Soc, pp. 301–309. [Online]. Available: http://ieeexplore.ieee.org/document/957835/

[17] J. Liu, Y. Wang, and Y. Wang, "The Similarity Analysis of Malicious Software," in *2016 IEEE First International Conference on Data Science in Cyberspace (DSC)*. IEEE, jun 2016, pp. 161–168. [Online]. Available: http://ieeexplore.ieee.org/document/7866123/

[18] W. Song, "a framework for automated similarity analysis of malware," Master's thesis, Concordia University, September 2014, unpublished. [Online]. Available: https://spectrum.library.concordia.ca/id/eprint/978935/

[19] A. Lee and T. Atkison, "A Comparison of Fuzzy Hashes," in *Proceedings of the SouthEast Conference*. New York, NY, USA: ACM, apr 2017, pp. 18–25. [Online]. Available: https://dl.acm.org/doi/10.1145/3077286.3077289

[20] V. Roussev, "Data fingerprinting with similarity digests," in *Advances in Digital Forensics VI*, K.-P. Chow and S. Shenoi, Eds. Berlin, Heidelberg: Springer Berlin Heidelberg, 2010, pp. 207–226.

[21] F. Breitinger, K. P. Astebøl, H. Baier, and C. Busch, "mvhash-b - a new approach for similarity preserving hashing," in *2013 Seventh International Conference on IT Security Incident Management and IT Forensics*, 2013, pp. 33–44.

[22] [Online]. Available: https://github.com/ANSSI-FR/polichombr/blob/dev/docs/MACHOC_HASH.md

[23] [Online]. Available: https://www.conix.fr/machoke-hashing/

[24] [Online]. Available: https://www.radare.org/n/r2pipe.html

[25] "Documentación de la librería hashlib de python." [Online]. Available: https://docs.python.org/3/library/hashlib.html

[26] [Online]. Available: https://www.blake2.net/