3. sigMatch

3.1. Introducción

Ya en el primer capítulo se introdujeron los objetivos que perseguía **sigMatch**, básicamente se explicó que consistía en la implementación de un **filtrado previo** a un algoritmo de búsqueda de patrones. Para entenderlo mejor, en este capítulo se explicará en profundidad la propuesta que en 2010 lanzaron Jignesh M. Patel y su equipo de investigación de la Universidad de Wisconsin-Madison.

Normalmente, en cualquier sistema que la implemente, la búsqueda de patrones suele ser la tarea de más carga computacional. En muchos casos, cuando se realiza esta tarea, se produce un bajo porcentaje de coincidencias, es decir la mayoría de los elementos que se envían a esta búsqueda no provocan ningún positivo. Se podría pensar que una buena mejora sería la inclusión de un filtrado previo que evitara que gran parte de los elementos fueran analizados por la **unidad de verificación** final, que es donde se encuentra el algoritmo de búsqueda de patrones. De esta forma se ahorraría un importante consumo de recursos que podrían ser asignados a otras tareas. Para entender mejor la utilidad de tal filtro, en los siguientes párrafos se plantea un ejemplo de una situación real.

Supóngase una aplicación que se encargue de gestionar el control de escritura de ficheros en un sistema y una de sus funciones sea comprobar si dicho fichero ya existe previamente en el sistema. Para ello, no bastaría con comparar el nombre del fichero, sino que también habría que analizar su contenido, realizando una comparación byte a byte. Por tanto, se podría decir que estaríamos en el caso de un modelo de búsqueda de patrones en un texto, donde los patrones serían los ficheros existentes y el texto donde buscar se correspondería con los ficheros entrantes.

En unas condiciones normales de funcionamiento la probabilidad de que un fichero nuevo exista previamente en el sistema es relativamente baja y la comprobación mediante búsqueda de patrones requeriría mucho tiempo. Si antes del paso por la búsqueda de patrones, cada fichero pasara por un filtro previo que pudiera determinar rápidamente y con seguridad que un fichero no forma parte del sistema, se podría evitar el envío del mismo hacia la unidad de verificación final. De esta forma el sistema estaría ahorrando un coste computacional importante.

Quizás a esta altura el lector se haya preguntado qué patrones son los que el filtro busca para llevar a cabo su cometido. Para aclararlo, se introducirá una idea simple aunque más adelante se explicará con más detalle todo el proceso de selección de los patrones que intervienen en el filtro. La unidad de verificación cuenta con una base de firmas para la búsqueda de patrones. Lo que hace el filtro es obtener de cada firma una porción significativa a la que se le denominará **resumen**. El conjunto de resúmenes formará la base de firmas del filtro, con la que posteriormente se creará la estructura de detección. De esta forma, el filtro se encarga de buscar dichos resúmenes. Por tanto, es lógico

pensar que si no se ha encontrado el resumen, la unidad de verificación final tampoco encontrará su firma correspondiente y por tanto el resultado del elemento al pasar por la búsqueda de patrones será negativo.

Este es el objetivo de sigMatch y se presenta por los autores como una **técnica de filtrado rápida, versátil y escalable** orientada a la búsqueda multi-patrón. La versatilidad radica en el hecho de que puede implantarse en cualquier sistema o aplicación que lleve a cabo una búsqueda de patrones, sin importar qué tipo de algoritmo use, y la escalabilidad se refiere al hecho de que no se ve afectado por el incremento de número de reglas, es más cuando se muestren los resultados obtenidos por los autores, se comprobará que se obtienen mejores resultados al incrementar el número de reglas usadas en la búsqueda de patrones.

Trasladando el concepto de sigMatch a un IDS como Snort, la descripción general del funcionamiento, explicada en la situación anterior, se traduciría en la realización de un filtrado previo al paso por el algoritmo de búsqueda de patrones pertinente de todo el tráfico entrante, descartando de este modo un alto porcentaje de tráfico benigno y, por consiguiente, evitando que sea analizado posteriormente por la unidad de verificación final.

La motivación de crear sigMatch fue fruto del conocimiento previo, por parte de los autores, de la problemática referente a la búsqueda de patrones, la cual ha sido estudiada por diversos investigadores que han propuesto varios métodos de mejora, tanto hardware como software. Las soluciones hardware están limitadas y son diseñadas prácticamente para IDSs, mientras que la mayoría de aplicaciones que realizan una búsqueda de patrones se ejecutan desde máquinas convencionales, por lo tanto descartaron la posibilidad de implementar una solución hardware. En el campo del software, los algoritmos de búsqueda exacta multi-patrón están basados en autómatas finitos o **tablas de decisión** (*shift table*). Ya se han visto, en el segundo capítulo, los autómatas del tipo determinista (DFA) y no determinista (NFA), también conocidos como ***tries,*** así como los filtros Bloom, que son un tipo de tabla de decisión.

Las soluciones basadas en autómatas emplean un DFA o un NFA para expresiones regulares. Se sabe que los NFA ocupan menos memoria que los DFA pero, en detrimento, suelen ser más lentos que la versión determinista. No obstante, existe un fenómeno denominado “explosión de estados” en los DFA que provoca una disminución del rendimiento debida al incremento del uso de memoria a medida que crece la base de datos de firmas. Recientemente, han salido a la luz varios estudios en los que, mediante técnicas de agrupamiento y reescritura, reducen el uso de memoria usada por los DFA. Aún así, según los autores de sigMatch, el rendimiento de estos sistemas sigue decreciendo cuando se superan los cientos de firmas.

Como se ha visto en el segundo capítulo, un **filtro Bloom** es un vector binario (aunque suele representarse en forma matricial) usado para comprobar si un elemento pertenece a un conjunto. Inicialmente todos las posiciones del vector están a cero y cuando se añade un elemento, se le aplican k funciones hash distintas que devolverán k valores, también distintos. Estos valores representan las posiciones del vector que han de cambiar su valor a un uno lógico. Cuando ya han sido añadidos todos los elementos, entonces llega el momento de comprobar si un elemento entrante puede formar parte del conjunto. Para ello, se le aplican las mismas k funciones y se obtienen los k valores correspondientes a ciertas posiciones del vector. Si al menos una de las k posiciones del vector contiene un cero, entonces se puede asegurar que el elemento no pertenece al conjunto. Si por el contrario todas las posiciones contienen un uno entonces el elemento puede pertenecer al

conjunto. Esta incertidumbre es uno de los motivos por los que el filtro sigMatch deja pasar hacia la unidad de verificación elementos "candidatos", que luego podrá resultar que la búsqueda de patrones definitiva los considere positivos o negativos. No obstante, se cuenta con la seguridad de que nunca se descartará un elemento que pudiera provocar un positivo en la unidad de verificación final, es decir todo elemento descartado dará, con total seguridad, un resultado negativo al pasar por la posterior búsqueda de patrones.

3.2. Fundamento y conceptos

En este punto del capítulo se explicará la idea de global del filtrado que propone Patel y algunos conceptos importantes que servirán para entender mejor el funcionamiento final de sigMatch.

En primer lugar se explicarán los motivos que llevaron al equipo de sigMatch a elegir la estructura de detección usada en el filtro. Posteriormente, se verá como se obtiene el diccionario de patrones usado para construir dicha estructura. Y por último, se mostrará cómo se lleva a cabo el paso de un cierto texto o elemento a través de la estructura de detección.

3.2.1. Estructura de detección

El hecho de que el filtro sigMatch se encargue de buscar patrones en un texto lo convierte en un algoritmo más de búsqueda de patrones y, como tal, debe contar con una estructura de detección que sea lo más óptima posible teniendo en cuenta las características de los patrones. Puesto que es conocido que en función de la longitud y el número de patrones la eficiencia de un algoritmo varía, no debe elegirse esta estructura a la ligera.

Llegados a este punto, los autores de sigMatch se tuvieron que plantear qué estructura usar. Uno de los objetivos que establecieron para obtener un filtro eficiente fue el de aprovechar al máximo las capacidades de la memoria caché del procesador, la cual proporciona accesos mucho más rápidos que la memoria principal pero cuenta con menos capacidad de almacenamiento. Por tanto, para aprovechar la velocidad de los accesos a memoria caché, propusieron crear una estructura que fuera lo suficientemente pequeña como para que pudiera ser almacenada completamente en memoria caché.

Como se ha dicho anteriormente, se sabe que las estructuras más usadas en los algoritmos de búsqueda de patrones más extendidos son los *tries* y las tablas de decisión. Los *tries*, por lo general, son más rápidos que las tablas de decisión, pero su estructura crece rápidamente a medida que aumenta el número de firmas de la base de datos. Sin embargo, las tablas de decisión, y en concreto los filtros Bloom, pueden mantener la misma cantidad de memoria usada independientemente del tamaño de la base de firmas, aunque para reducir el número de colisiones convendría aumentar levemente el tamaño de la matriz. Por consiguiente, los filtros Bloom tienen un crecimiento casi nulo en comparación con las estructuras basadas en *tries*.

El conocimiento de las ventajas e inconvenientes de cada estructura, condujo al equipo de sigMatch a crear una estructura de detección innovadora, bautizada con el nombre de **sigTree**, la cual se

muestra en la figura 3.1. Su innovación radica en el hecho de combinar *tries*, en su versión no determinista, y filtros Bloom en una misma estructura, permitiendo aprovechar los puntos fuertes de cada método, a la vez que evitando las desventajas que acarrearía usarlos por separado. Esta combinación de *tries* y filtros Bloom no se conocía en otro algoritmo hasta la fecha. Podría considerarse a la estructura sigTree como un *trie* truncado con un filtro Bloom en cada uno de sus últimos nodos, denominados nodos hoja.

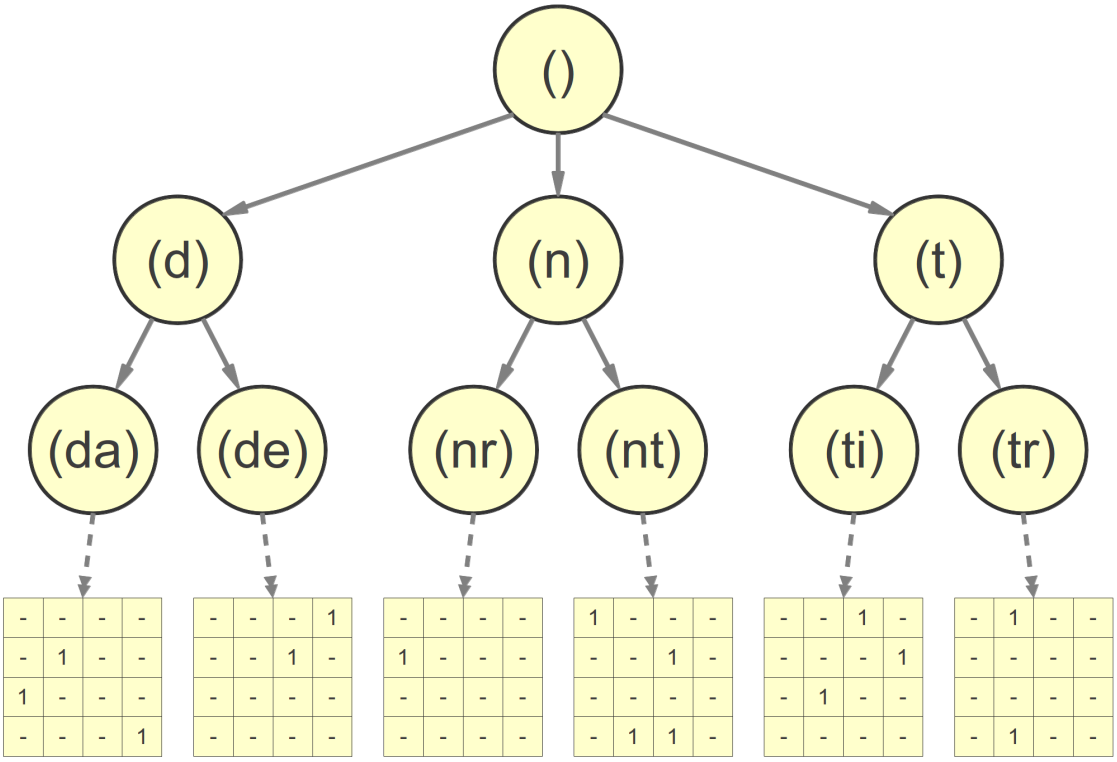

Figura 3.1: Estructura de detección de sigMatch: sigTree. b=2.

Una vez decidida la estructura de detección que se usará en el filtrado, han de definirse las características de cada una de las dos partes por separado. Los *tries* se construirán con patrones de longitud fija de ***b*** bytes, mientras que los filtros Bloom se crearán con patrones de ***ß*** bytes. Para no sufrir la problemática del crecimiento desmesurado en los *tries*, el valor *b* debería ser pequeño, por lo general se escogerá un valor igual a 2 bytes. Sin embargo, para el valor de *ß* un margen aconsejable iría de 3 a 6 bytes. La elección de los valores de *b* y *ß* será un compromiso entre el consumo de memoria y la tasa de falsos positivos. Cuanto mayor longitud se use para crear la estructura sigTree, menor tasa de falsos positivos tendrá, pero mayor coste computacional y mayor consumo de memoria requerirá. Por consiguiente, cada resumen que se obtenga de la base de firmas del sistema deberá tener una longitud fija de *b+ß* bytes. El conjunto de resúmenes formará la base de firmas de sigMatch, aunque también se la conocerá como **base de resúmenes**.

3.2.2. Resúmenes de la base de firmas

El siguiente punto del desarrollo de sigMatch consistió en decidir qué combinación de $b+\beta$ bytes se utilizaría de cada firma original para crear la base de resúmenes de firmas de sigMatch. Se propuso obtener un resumen de b bytes de las firmas original y posteriormente escoger los β bytes siguientes para rellenar la matriz del filtro Bloom. Los tipos de resúmenes más extendidos en el mundo del filtrado se basados en prefijo y en *q-gram*. Antes de seguir, sería necesario aclarar el concepto de *q-gram*. Dada una secuencia de elementos, como por ejemplo una cadena de caracteres, un *q-gram* es el conjunto de combinaciones de q caracteres de dicha secuencia. Por ejemplo, si tenemos la palabra *algoritmo*, los *3-grams* (también conocidos como *trigrams*) correspondientes serán todos los del conjunto {*alg*, *lgo*, *gor*, *ori*, *rit*, *itm*, *tmo*}.

Los filtros con resúmenes basados en prefijo cuentan con un conjunto de prefijos $P = \{p_1, p_2 ..., p_m\}$, tales que cada firma o patrón de la base de firmas $F = \{f_1, f_2, ..., f_k\}$ tendrá su prefijo en el conjunto P, donde el número de prefijos que forman el conjunto P será menor o igual que el número de patrones que forman la base de firmas, o dicho de otro modo, la cardinalidad de P será menor que la de F, es decir $m \leq k$.

Para obtener el conjunto de prefijos P, simplemente habría que copiar los primeros c caracteres de cada firma del conjunto F, donde c se establece al inicio del proceso, y en el caso de que hubieran prefijos iguales eliminar las duplicidades, de ahí que se pueda dar el caso de $m<k$. Si durante el proceso *online* del filtrado de un dato, no se encontrara ningún prefijo del conjunto P, entonces se podría asegurar que en ese dato tampoco se encontrará ninguna firma del conjunto F. En la figura 3.2 se muestra un ejemplo de la situación anterior, en la que se cuenta con una base de firmas F y se obtienen resúmenes basados en prefijo. Para ser lo más imparcial posible en la elección de las firmas, se han seleccionado palabras extraídas del primer párrafo de la obra *A un olmo seco*, de Antonio Machado.

Un enfoque similar siguen los filtros basados en *q-gram*, que forman un conjunto de resúmenes $R = \{r_1, r_2, ..., r_n\}$, tales que cada firma del conjunto F pueda ser representada por al menos un resumen perteneciente al conjunto R. En este caso también el número de *q-grams* será menor o igual al número de patrones que forman la base de firmas e incluso será menor o igual al número de prefijos, o lo que es lo mismo, la cardinalidad de R será menor o igual que la de P, que a su vez será menor o igual que la de F, es decir $n \leq m \leq k$. Esto quiere decir que la cardinalidad del conjunto de los *q-grams* será menor o igual que la del conjunto de prefijos, lo cual es un punto a favor. A pesar de esta ventaja que aporta el uso de *q-grams*, también hay que tener en cuenta que la fase *offline* será más costosa en términos computacionales. Para construir el conjunto R se extraen todos los *q-grams* posibles de la base de firmas F y se ordenan por frecuencia, de mayor a menor. Desde el primero de esta lista ordenada se van

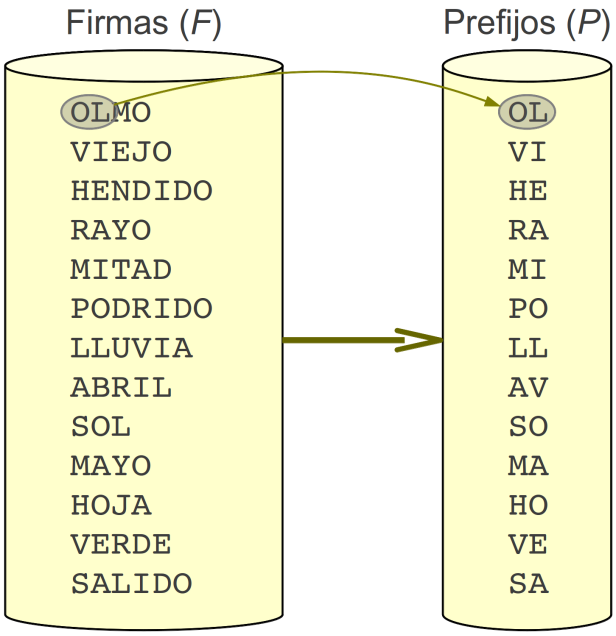

Figura 3.2: Filtrado basado en prefijo. En este caso la cardinalidad de P *es igual a la de* F.

añadiendo los *q-grams* al conjunto *R* hasta que todas las firmas sean representadas por al menos un elemento del conjunto *R*. En la figura 3.3 se explica la situación en el caso de un resumen basado en *2-gram*, usando como firmas las del ejemplo de la figura 3.2. En secciones posteriores se describirá con más detalle el proceso completo de que se lleva a cabo en la obtención de resúmenes de firmas.

Puesto que el objetivo es obtener una estructura que ocupe el menor espacio posible, para así poder estar el máximo tiempo disponible en memoria caché, y el rendimiento de la fase *offline* no debería ser un factor limitante, los autores recomiendan el uso del filtro basado en *q-gram* en lugar de en prefijo.

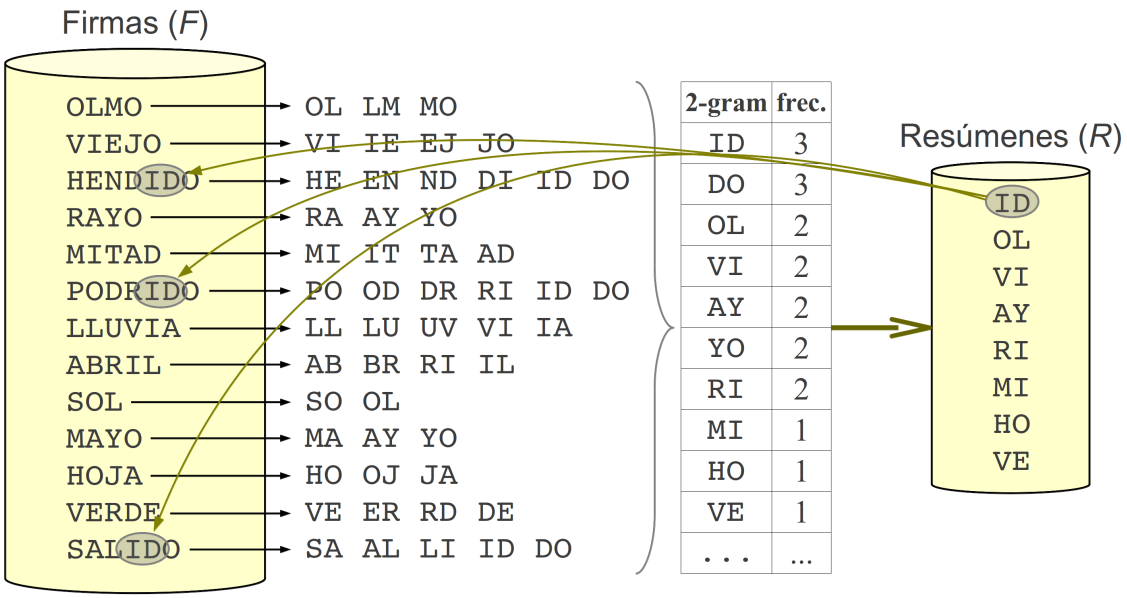

Figura 3.3: Filtrado basado en 2-gram*. En este caso la cardinalidad de* R *es menor a la de* F.

3.2.3. Paso por sigTree

Una vez que pasa un elemento por la estructura de sigTree y se detecta como candidato, puede que éste se trate de un **verdadero positivo**, si existe al menos un patrón de la base de firmas original en el elemento, o de un **falso positivo**, en el caso de que ningún patrón de la base de firmas original esté presente en el elemento.

Los falsos positivos se pueden producir por dos motivos diversos, por tanto existen falsos positivos de tipo A y de tipo B. Los del tipo A se dan cuando el resumen detectado en el filtro existe en el elemento analizado, pero la firma completa de la que se extrajo el resumen no. En cambio, los del tipo B se producen cuando ni siquiera el resumen detectado existe en el elemento. Este tipo de falso positivo se produce debido a las colisiones que presenta el filtro Bloom, ya explicadas anteriormente.

Es evidente que reducir el número de falsos positivos sería importante para así realizar menos llamadas a la búsqueda exacta de patrones y, por consiguiente, obtener un mejor rendimiento. Los

del tipo A se podrían reducir si se aumentara la longitud del resumen $b+\beta$. Al incrementar b aumentaría el coste computacional de los *tries* debido a una mayor penalización cuando se produjeran *backtrackings*, mientras que si se aumentara β afectaría al coste computacional del cálculo de los hashs. Para reducir los falsos positivos de tipo B, que únicamente dependen de la tasa de colisiones de los filtros Bloom, las posibilidades serían usar funciones hash que produzcan menos colisiones, aumentar el tamaño del filtro Bloom o aumentar la longitud β. Esta última opción ya se ha explicado en el caso de falsos positivos de tipo A. La opción de usar funciones hash más eficiente podría afectar al coste computacional. También se puede estudiar la posibilidad de incluir más funciones hash con coste computacional bajo, como proponen los autores.

En el siguiente punto se explicarán, con ayuda del ejemplo de funcionamiento de la fase *offline* mostrado en la figura 3.8, los casos en los que se produzcan negativos, verdaderos positivos y falsos positivos de ambos tipos.

3.3. Funcionamiento

Una vez explicados los conceptos más importantes del filtro sigMatch, en este punto se describirán con detalle:

- la **situación que tomará el filtro** dentro de un entorno de búsqueda de patrones,
- la **fase *offline***, que comprende la creación de la estructura sigTree usando la base de firmas original, y
- la **fase *online***, que comienza con el recorrido de los datos de entrada a través del conjunto final formado por el filtro y el algoritmo de búsqueda exacta de patrones.

3.3.1. Situación general

En la figura 3.4 se muestra una situación general de búsqueda de patrones en la que aparece la unidad de verificación, que analiza el flujo de datos y toma una decisión en función de la base de firmas originales y del contenido de los datos entrantes.

Como ya se ha dicho, la propuesta por parte del equipo de sigMatch es introducir el filtrado entre el flujo de datos y la unidad de verificación, a la vez que se sirve de la información proporcionada por la base de firmas originales. El esquema correspondiente se muestra en la

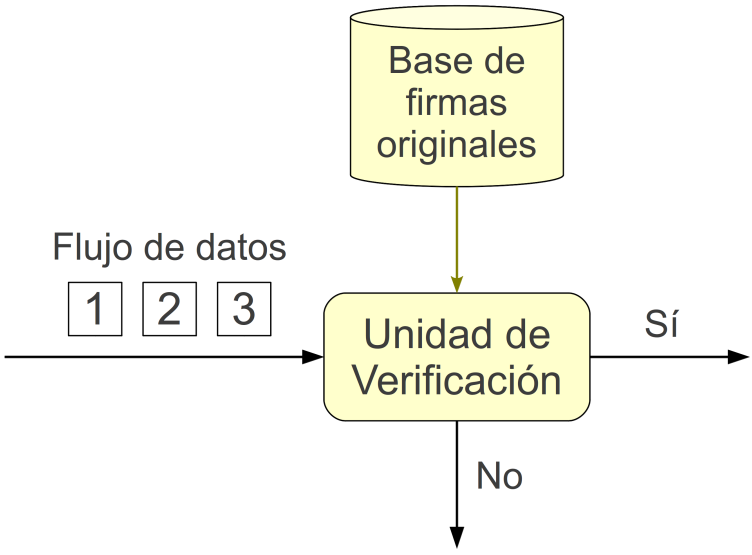

Figura 3.4: Esquema general correspondiente a un proceso de búsqueda de patrones.

figura 3.5. En él se aprecia como todo el flujo de datos pasa a través del filtro sigMatch y éste decide, en función del resumen de firmas, obtenido de las firmas originales, y del contenido de los datos, si debe continuar hacia la unidad de verificación, que será la encargada de realizar el análisis definitivo.

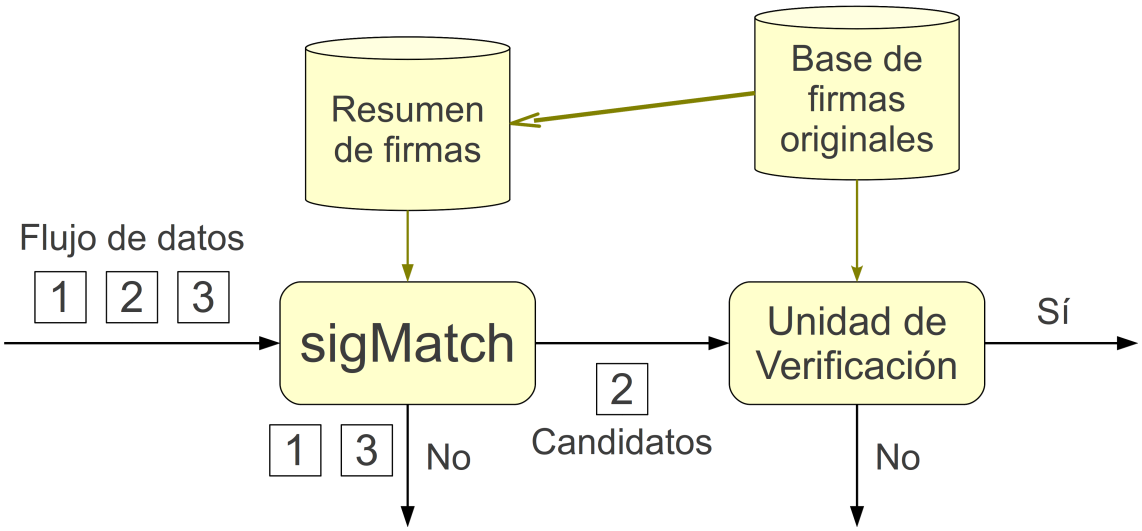

Figura 3.5: Esquema general correspondiente a un proceso de búsqueda de patrones que implemente el filtro sigMatch antes de invocar a la Unidad de Verificación.

3.3.2. Fase *offline*

Como ya se ha dicho, en la fase *offline* de cualquier proceso de búsqueda de patrones se crea la estructura de detección sobre la que luego se realizará dicha búsqueda. Normalmente, esta fase sólo se realiza una vez, al principio de la ejecución de la aplicación, por lo tanto en la mayor parte de las situaciones no será una fase en la que adquiera demasiada importancia el coste computacional, sino que primará la obtención de una estructura de detección lo más óptima posible en cuanto a consumo de memoria y que proporcione un mejor rendimiento en la posterior fase *online*, a pesar de que ello requiera de un consumo importante de recursos al inicio.

En el caso de sigMatch, en la fase *offline* se creará la estructura sigTree en base a las firmas originales. Para ello, se deben tener en cuenta algunas consideraciones; la primera es que los autores plantean que sigMatch permita en la base de firmas originales el **uso de expresiones regulares y caracteres comodín**, como `*` y `?`, y la segunda es que, habiendo establecido la longitud del resumen $b+ß$, cabe la posibilidad de que existan firmas originales que tengan una longitud menor a la requerida por las firmas resumidas. Estas situaciones, junto con demás consideraciones, se explicarán a continuación.

Como se ha dicho, los autores proponen que sigMatch permita el uso de expresiones regulares y caracteres comodín, como `*` y `?`, en la base de firmas originales. Puesto que la estructura sigTree consta de un *trie* NFA de altura fija *b* que, normalmente, tomará el valor 2 y de un filtro Bloom, que no han sido diseñados para soportar expresiones regulares ni caracteres comodines, el resumen que

se obtenga de las firmas originales no deberá soportar los caracteres comodín, sino que sólo admitirá resúmenes formados por patrones normales. Por tanto, dada una firma con expresiones regulares o con comodines, se deberán analizar únicamente las partes de la firma que contengan patrones normales y obtener el resumen únicamente de ellas. Por ejemplo, en el caso de *b=2* y *ß=4* y considerando la firma `HTTP*CONNECTION?_REFUSED`, en primer lugar se dividirá la firma en los fragmentos `HTTP`, `CONNECTION` y `_REFUSED`, en segundo lugar se analizará cada fragmento por separado y en tercer lugar se tomará el resumen cuyo *2-gram* sea el de mayor frecuencia, suponiendo que dicho *2-gram* sea `CO`, el resumen de la firma será `CONNEC`.

Cuando haya firmas demasiado pequeñas cuya longitud sea inferior a *b+ß*, el resumen de la firma será la misma firma en sí y se añadirá el primer *b-gram* al conjunto de *b-grams* recopilados hasta ese momento, para que forme parte de la lista ordenada por frecuencia.

En el caso de que las firmas, siendo lo suficientemente largas, contengan caracteres comodín que hagan que ninguno de sus fragmentos analizables tenga longitud mayor que *b+ß*, se escogerá el fragmento más largo posible como resumen de dicha firma, y se actuará como se explicaba en el párrafo anterior, es decir añadiendo su *b-gram* al conjunto de *b-grams* recopilados hasta el momento. Si hubieran dos o más fragmentos que compartieran la misma longitud mayor entonces se elegiría como resumen el fragmento cuyo *b-gram* contara con mayor frecuencia. Si a su vez, hubiera más de uno con la misma frecuencia entonces se escogería indistintamente entre cualquiera de ellos. De hecho, en el último ejemplo descrito se descarta el fragmento `HTTP`, y se tienen en cuenta únicamente los dos siguientes. En un nuevo ejemplo, teniendo la firma original `HTTPS*QUIT??HCK` y los valores *b=2* y *ß=4* puesto que todos tienen una longitud menor de 6 caracteres, tendríamos que elegir como resumen el fragmento más largo posible, que en este caso es `HTTPS`, y añadir `HT` al conjunto de *b-grams*.

Para este tipo de firmas, en las que la longitud es menor que *b+ß* pero mayor o igual a *b*, no se puede añadir la cadena que le sigue al filtro Bloom puesto que tendrá una longitud menor que *ß*, sino que tendrá que ser enlazada a una lista de firmas cortas.

Los autores de sigMatch no contemplan el caso de que haya firmas de longitud menor que *b*. Sin embargo, en la implementación que se ha realizado en Snort sí se ha debido tener en cuenta esta posibilidad, puesto que este tipo de firmas existen.

Una consideración a tener en cuenta en la obtención de los *q-grams* es que se deberán omitir los últimos *ß* caracteres de la firma del análisis, puesto que estos sólo podrían formar parte del filtro Bloom en el caso de que se escogiera el último *q-gram* posible de la firma. Por ejemplo, si se tuviera la firma original `CONNECTION` y los parámetros *b=2* y *ß=4*, entonces se tendrían en cuenta los *b-grams* `CO ON NN NE EC`, obviando los tres últimos *b-grams*, puesto que no sería posible escogerlos como inicio del resumen, ya que deberían seguirle *ß* caracteres para poder formar un resumen estándar.

Otro punto relevante es que una misma firma original no puede proporcionar más de una vez el mismo *q-gram* al conjunto de *q-grams*. Por ejemplo, la firma `CONGRATULATIONS` contiene por partida doble el *b-gram* AT, sin embargo, no se debe aumentar la frecuencia del *b-gram* en dos, porque pertenece a la misma firma, sino que se obviará la segunda vez que se cuente, incrementando la frecuencia sólo en una unidad.

En las figuras 3.6 y 3.7 se muestra un ejemplo de los procedimientos que se llevan a cabo en la fase *offline*, es decir la obtención de resúmenes en base a un conjunto de firmas y la creación de la estructura sigTree. En el ejemplo se muestran todos los casos posibles de firmas que se han descrito en los párrafos anteriores y se han elegido los parámetros *b=2* y *ß=4*.

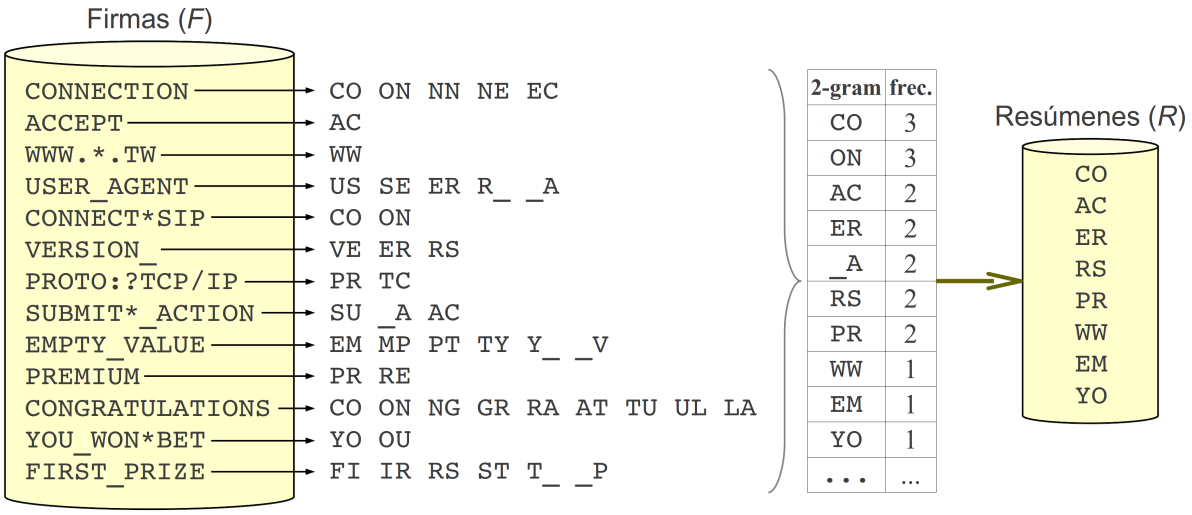

Figura 3.6: Obtención de los resúmenes R *en función de las firmas originales* F *(fase offline).*

A continuación, se explicarán los pasos seguidos para obtener el conjunto de resúmenes en base a las firmas originales (figura 3.6):

- Como se explicó con anterioridad, para la obtención de los *b-grams* se omiten los *ß* últimos caracteres puesto que estos formarán parte a lo sumo del filtro Bloom. Por tanto, tanto en la firma `CONNECTION`, como en las demás, no se tienen en cuenta los 4 últimos caracteres y se obtienen los *2-grams* a partir del patrón `CONNEC`.

- En la base de firmas del ejemplo existen firmas con caracteres comodín cuyos fragmentos tienen una longitud menor que *b+ß*, como es el caso de `WWW.*.TW`. En estos casos se toma como resumen el fragmento más largo y se añaden sus *b* primeros caracteres al conjunto de *b-grams*. En el ejemplo se trataría de `WW`.

- Para firmas con comodines, pero con fragmentos de longitud mayor o igual a *b+ß*, se procederá de la forma normal, es decir se obtienen los *b-grams* de cada fragmento descartando los *ß* últimos caracteres del análisis. En el caso de la firmas `CONNECT*SIP` y `SUBMIT*_ACTION` se obtienen los *2-grams* `CO` y `ON` a partir de la primera firma y `SU`, `_A` y `AC` a partir de la segunda.

- Una vez obtenida la lista de *2-grams* ordenada de mayor a menor frecuencia, se sigue el orden establecido en dicha lista para añadir al conjunto de resúmenes *R* aquellos *2-grams* que representen al menos a una firma que no haya sido previamente representada por otro *2-gram*. En primer lugar se elige el *2-gram* `CO`, que representa a las firmas `CONNECTION`, `CONNECT*SIP` y `CONGRATULATIONS`; el siguiente *2-gram* sería `ON`, que representa a las tres firmas anteriores, pero como ya están representadas por `CO`, el *2-gram* `ON` se descarta

como resumen; el siguiente entonces sería `AC`, que representa a `ACCEPT` y a `SUBMIT*_ACTION`. Se sigue con el mismo procedimiento hasta que todas las firmas *F* tengan al menos un *2-gram* que las represente.

Cuando ya se tiene el conjunto de resúmenes *R*, se dispone a crear la estructura de detección sigTree. Este proceso se describe a continuación (figura 3.7):

- Con la información que proporciona el conjunto *R* ya se puede crear el *trie* NFA de altura *b=2* que forma la primera parte de la estructura sigTree. La creación de dicho árbol a partir del conjunto *R* es bastante intuitiva. Para los nodos de primer nivel se escogen los primeros bytes de cada resumen del conjunto *R*, después se enlazarán estos nodos con los nodos hoja que son representados por el segundo y último byte de los resúmenes. La explicación gráfica de la figura 3.7 será mucho más aclaratoria que cualquier explicación escrita.

- Cada *2-gram* del conjunto *R* representa al menos a una firma del conjunto *F*. Una vez construido el *trie*, sus nodos finales, o nodos hojas, apuntarán a un filtro Bloom, a una lista enlazada o a ambos. En el caso del ejemplo, todos están apuntando a un filtro Bloom excepto el nodo hoja `WW`, que apuntará a una lista enlazada. Esto es así debido a que el fragmento escogido de mayor longitud de la firma `WWW.*.TW` tiene una longitud menor que *b+ß* y por lo tanto no hay una cadena de *ß* caracteres que se pueda añadir al filtro Bloom, sino que se deberá añadir la cadena que sigue a los *b* caracteres del fragmento a una lista enlazada. En este caso, puesto que el fragmento escogido de la firma es `WWW.`, el *2-gram* elegido es `WW` y la cadena introducida en la lista enlazada será `W.`.

- Cuando no se está en el caso de firmas cortas, entonces se aplica una función hash a la cadena de *ß* caracteres que va detrás de cada *2-gram*. El resultado devuelto por la función indicará la posición de la matriz que deberá cambiar su valor a un uno lógico.

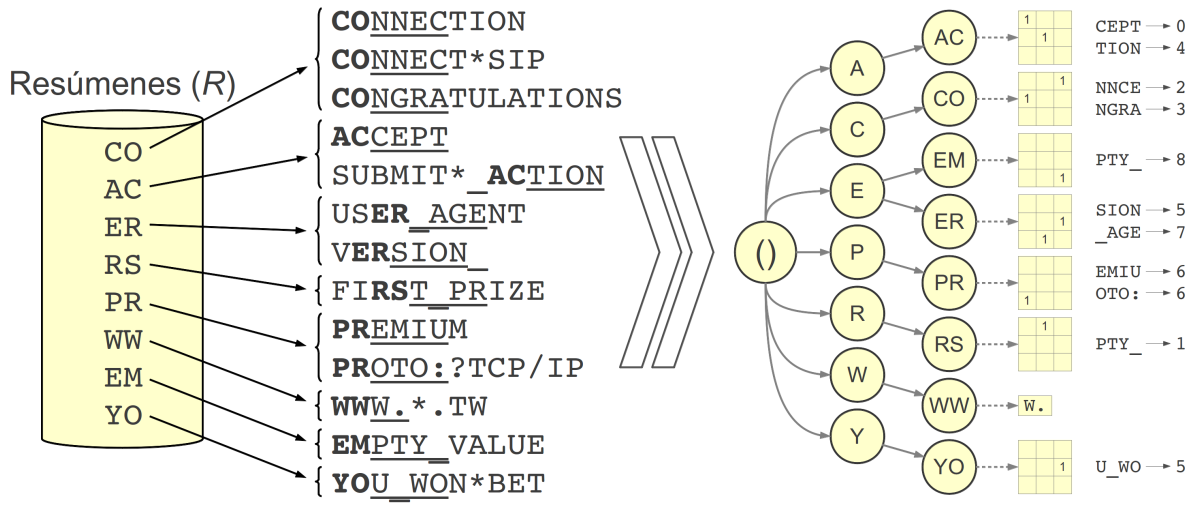

Figura 3.7: Creación de la estructura sigTree a partir de los resúmenes R *y las firmas originales* F *(fase offline).*

3.3.3. Fase *online*

Una vez construida la estructura de sigMatch se puede iniciar la fase *online*, que será donde se inspeccionarán los datos entrantes mediante el recorrido por sigTree para decidir si son candidatos para ser analizados por la unidad de verificación.

En la figura 3.8 se muestra un ejemplo en el que sigMatch analiza siete líneas que forman parte de una comunicación SIP. En este caso se considera que cada línea es un dato, por lo que serán susceptibles de ser tratadas por separado de las demás y por tanto cada línea podrá ser descartada o reenviada a la unidad de verificación como candidata, indistintamente del resto. Las líneas son analizadas carácter a carácter y pasan por el *trie* de tipo NFA. Si llegan al nodo final, entonces pasan al filtro Bloom y/o a la lista enlazada, según sea el caso. A continuación, se detallan los posibles casos que se producen en el ejemplo:

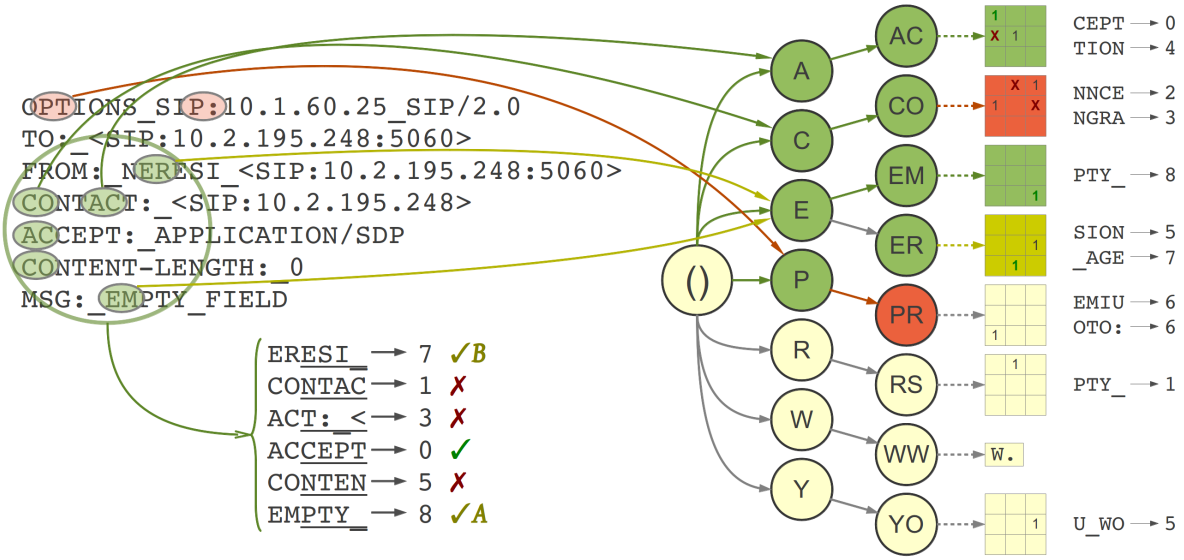

Figura 3.8: Representación del paso de datos por la estructura sigTree (fase online).

- En la figura 3.8 se puede comprobar que al recorrer las dos primeras líneas no se llega a ningún nodo final, por lo que dichas líneas no pasarán siquiera a la siguiente parte de la estructura sigTree y por consiguiente se descartarán.

- La tercera línea, sin embargo, contiene el *2-gram* `ER` que alcanza un nodo final. Dicho nodo apunta únicamente a un filtro Bloom, por lo tanto la función hash evalúa la cadena `ESI_`, formada por los siguientes *ß* caracteres, y devuelve la posición de una posición de la matriz. En este caso, dicha posición contiene el valor 1, por lo tanto, la línea pasa como candidata hacia la unidad de verificación. Se puede observar como estamos en el caso de un falso positivo, ya que la cadena `ESI_` no fue añadida al filtro Bloom en la fase *offline*. Estamos por tanto en el caso de un falso positivo de tipo B, puesto que el positivo se debe a una colisión producida en el filtro Bloom.

- En la cuarta línea tenemos esta vez dos *2-grams* que alcanzan un nodo final, son el caso de `CO` y `AC`, que les siguen las cadenas `NTAC` y `T:_<`, respectivamente. Estas cadenas son analizadas por la función hash, que determinan que no pertenecen al filtro Bloom, por tanto la cuarta línea no se envía como candidata. Notar que hubiera bastado que una de las cadenas perteneciera a un filtro Bloom para que se enviara la línea completa como candidata a la unidad de verificación.

- La quinta línea es el siguiente caso de positivo que se presenta. A través del *2-gram* `AC` que llega a un nodo final, se calcula el hash de la cadena `CEPT`, que resulta pertenecer al filtro Bloom y por lo tanto se envía la línea como candidata a la unidad de verificación. En este caso no se trata de un falso positivo de tipo B, puesto que verdaderamente la cadena `CEPT` sí fue añadida al filtro Bloom en la fase *offline*. Que no se trate de un falso positivo de tipo B no quiere decir que no lo pueda ser de tipo A. Se recuerda que un falso positivo de tipo A era aquél que se producía al encontrar un resumen en el dato, pero al pasar por la unidad de verificación se comprobaba que la firma correspondiente al resumen no formaba parte del dato. En este caso concreto sabemos que tampoco se trata de un falso positivo de tipo A puesto que la firma correspondiente al resumen es `ACCEPT`, que se encuentra en la línea. Aunque en este ejemplo es evidente que no es un falso positivo de tipo A, puesto que el resumen coincide con la firma, es evidente también que no siempre se dará esta coincidencia.

- La sexta línea también contiene un *2-gram*, `CO`, que alcanza un nodo final, pero en este caso la cadena `NTEN`, formada por los *ß* siguientes caracteres, no pertenece al filtro Bloom. Con lo cual, esta línea es descartada del análisis definitivo en la unidad de verificación.

- Por último, en la séptima línea el *trie* el *2-gram* `EM` alcanza un nodo final que apunta a un filtro Bloom. La función hash evalúa los siguientes *ß* caracteres `PTY_` y determina que la posición que se obtiene contiene un uno lógico, por tanto envía la línea como candidata a la unidad de verificación final. Estamos en el caso de un falso positivo de tipo A puesto que sabemos que el resumen `EMPTY_` forma parte de la línea pero que la firma de la que se obtuvo el resumen, `EMPTY_VALUE`, no forma parte de ella. Por tanto, la unidad de verificación descartará esta línea después de realizar su análisis.

3.4. Resultados y conclusiones

Patel y su equipo propusieron un filtrado previo a la búsqueda múltiple de patrones, bautizado con el nombre de sigMatch, que mejorara el rendimiento de sistemas y aplicaciones en los que una parte importante del coste computacional se atribuyera a dicha búsqueda de patrones. Estos sistemas y aplicaciones son, por ejemplo, los sistemas de extracción de información, las aplicaciones antivirus y los sistemas de detección de intrusos.

Por tanto, una vez establecidos los conceptos del filtro sigMatch, realizaron pruebas integrándolo con los tres sistemas nombrados en el párrafo anterior. Las aplicaciones elegidas fueron **DBLife**

como sistema de extracción de información, **ClamAV** como antivirus y **Bro** como sistema de detección de intrusos.

En las pruebas realizadas, DBLife contaba con una biblioteca de expresiones regulares de unas 61.000 firmas, ClamAV usaba 90.000 firmas y Bro sólo incluía 1.200 firmas. Mientras que el equipo en el que se realizaron las pruebas contaba con las siguientes características: procesador Intel Core 2 Duo a 2,00 GHz, memoria RAM de 3 GB, 32 KB de memoria caché L1 y 2 MB de memoria caché L2.

Después de comprender en profundidad, tanto el desarrollo de sigMatch, como el funcionamiento de los algoritmos de búsqueda de patrones, se podría afirmar, sin correr el riesgo de errar, que **faltan datos importantes** para poder valorar adecuadamente los resultados obtenidos en las pruebas hechas por el equipo de sigMatch. En principio, los datos considerados más importantes podrían ser dos, el primero de ellos sería la información acerca de la longitud de las firmas y el segundo el funcionamiento interno de la unidad de verificación de cada aplicación en la que se haya integrado sigMatch, para tener una idea del rendimiento que tiene la unidad por sí sola.

En el caso de la información acerca de la longitud de las firmas, después de lo visto hasta ahora en este capítulo se puede intuir que no será igual de eficiente el filtro sigMatch para los casos en los que la base de firmas original cuente con patrones cuya longitud sea de cientos de bytes que para los casos en los que dicha longitud sea de decenas de bytes. Es lógico pensar que la calidad del filtrado será mayor en el primer caso puesto que la estructura sigTree obtenida ocupará menos espacio en memoria, basándonos en el principio de que a igual número de firmas, cuanta mayor longitud tengan, más *q-grams* tendrán en común las firmas y por tanto la cardinalidad del conjunto de resúmenes R será menor. En principio bastaría con conocer la desviación típica de la longitud de las firmas. La única mención que se hace en el documento, en cuanto a la longitud de las firmas, es que las firmas usadas en Bro son más cortas que las usadas en ClamAV. De hecho, se comprueba como el rendimiento es bastante menor en Bro que en ClamAV.

En cuanto a la información acerca de las características de la unidad de verificación usada en cada aplicación en la que se ha integrado sigMatch para realizar las pruebas de rendimiento, sería lógico pensar que una posible mejora obtenida gracias a la integración de sigMatch tendrá mucho que ver con el rendimiento de la unidad de verificación. Es decir, suponiendo que se contaran con dos unidades de verificación distintas, denominadas unidad A y unidad B, que se usara la misma base de firmas y que se ejecutara un análisis de los mismos elementos, y suponiendo también que se obtuvieran mejores resultados en la unidad A que en la unidad B, sería lógico pensar que al implantar el filtro sigMatch en cada sistema, se obtendría una mejora de rendimiento mayor en la unidad B que en la unidad A. Puesto que esta conclusión quizás resulte confusa, se hará una demostración teórica ayudada del esquema mostrado en la figura 3.9 que muestre los tiempos de ejecución en cada caso.

Se define el tiempo de ejecución de la unidad A como t_A, y el tiempo de ejecución de la unidad B como t_B. Como se supuso anteriormente, la unidad A obtendrá mejores resultados que la unidad B, por tanto se tiene que $t_A<t_B$, es decir $t_A=\mu \cdot t_B$, con $0<\mu<1$. Al implementar sigMatch el tiempo de ejecución del filtro será t_F y suponiendo que analice N elementos, reenviará $k \cdot N$ elementos hacia la unidad de verificación final, siendo $0<k<1$. Por tanto, puesto que ambas unidades cuentan con la misma base de firmas y los elementos que deben analizar son idénticos en ambos casos, los nuevos tiempos de ejecución de las unidades A y B que se obtengan al implantar el filtro sigMatch serán el

resultado de multiplicar los tiempos anteriores por el factor *k*, que será el mismo para ambas unidades. Esta situación se muestra en la figura 3.9.

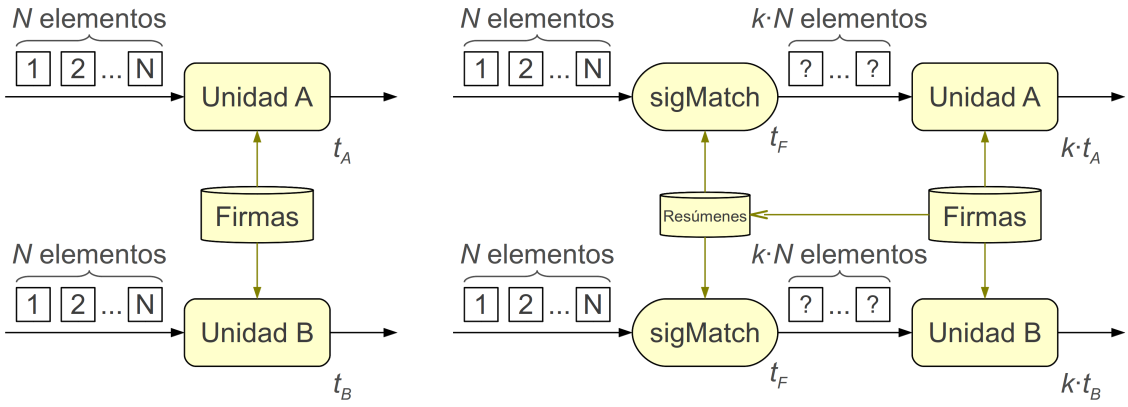

Figura 3.9 Tiempos de ejecución y factores de mejora de los sistemas A y B.

Se define también f_A (f_B) como el factor de mejora logrado en el sistema A (B) al implantar el filtrado previo. Este factor se calcula como el resultado de dividir el tiempo de ejecución del sistema A (B) sin ninguna implementación adicional por el tiempo de ejecución del sistema A (B) habiendo realizado un filtrado previo.

No hay que olvidar que la finalidad de este desarrollo era demostrar que si el sistema B cuenta con un tiempo de ejecución mayor que el del sistema A, también contará un factor de mejora mayor que el del sistema A, es decir si $t_B > t_A$ entonces $f_B > f_A$, o lo que es lo mismo $f_A = F \cdot f_B$, con $0 < F < 1$. Se demostrará la veracidad de esta afirmación en el siguiente desarrollo matemático.

Como ya se ha dicho, el factor de mejora que se obtiene en un determinado sistema se calcula dividiendo el tiempo de ejecución del sistema sin el uso del filtro por el tiempo de ejecución del sistema una vez que se implementa el filtro. Con ayuda de la figura 3.9 se tiene que los factores de mejora serán:

$$f_A = \frac{t_A}{t_F + k \cdot t_A} \qquad (1)$$

$$f_B = \frac{t_B}{t_F + k \cdot t_B} \qquad (2)$$

De la ecuación que relaciona los factores de mejora de los sistemas A y B y usando las ecuaciones (1) y (2) se tiene que:

$$f_A = F \cdot f_B \quad \rightarrow \quad \frac{t_A}{t_F + k \cdot t_A} = F \, \frac{t_B}{t_F + k \cdot t_B} \quad \rightarrow \quad F = \frac{t_A \left(t_F + k \cdot t_B \right)}{t_B \left(t_F + k \cdot t_A \right)} \qquad (3)$$

Usando ahora la relación entre t_A y t_B y continuando con la ecuación (3) se llega a:

$$F = \frac{t_A\left(t_F + k \cdot t_B\right)}{t_B\left(t_F + k \cdot t_A\right)} = \mu\,\frac{t_F + k \cdot t_B}{t_F + k \cdot t_A} = \frac{\mu \cdot t_F + \mu \cdot k \cdot t_B}{t_F + \mu \cdot k \cdot t_B} \qquad (4)$$

Puesto que se sabe que $0<\mu<1$, se podrá resolver que la ecuación (4) será siempre menor que 1 y por tanto se tendrá $0<F<1$. Como consecuencia, la desigualdad $f_B>f_A$ se cumplirá siempre. Queda así demostrada la hipótesis inicial de que a mayor tiempo de ejecución de un algoritmo, es decir peor rendimiento, el factor de mejora obtenido al incorporar un filtrado previo será mayor. Es por eso que se debería haber incluido una información más detallada de la unidad de verificación usada en cada aplicación a la hora de mostrar los resultados.

Se explica que ClamAV usa para la búsqueda de patrones una combinación entre Aho-Corasick, en su implementación *banded row* y Boyer-Moore extendido. Las implementaciones más usadas de Aho-Corasick son cuatro, ordenadas de mayor a menor rendimiento: Full matrix, Sparse, Banded y Sparsebands. Y en cuanto al algoritmo Boyer-Moore, se sabe que sólo es óptimo para búsqueda uni-patrón. Queda también por conocer si la combinación de ambos algoritmos proporciona un rendimiento óptimo en la unidad de verificación. En cuanto a DBLife y Bro, la única información proporcionada acerca de la unidad de verificación de cada una de ellas es que para la búsqueda de patrones usan una estructura DFA, sin más explicaciones.

Los resultados obtenidos por Patel y su equipo en la integración de sigMatch en los distintos sistemas se muestran en términos de tasa de transferencia (throughput), factor de mejora o de velocidad (speedup) y tasa de filtrado (filter rate).

A continuación, en la tabla 3.1 se muestra una comparativa de los tres sistemas usados en las pruebas de rendimiento, mostrando el tamaño de la estructura sigTree, el número de *cache misses* y el factor de velocidad.

Sistema	Tamaño de sigTree (KB)		L1 Cache Misses (en millones)			L2 Cache Misses (en millones)			Factor de velocidad (*speedup*)
	Sólo nodos	Nodos + BF	sin sigMatch	con sigMatch	% dec	sin sigMatch	con sigMatch	% dec	
ClamAV (90K sigs)	23	2700	603,1	76,2	87,4	139,8	3,9	97,2	11,7X
Bro (1.2K sigs)	17	970	441,3	103,6	76,5	81,7	9,8	88,0	4,4X
DBLife (61K sigs)	21	2400	764,3	152,1	80,1	176,4	15,7	91,4	15X

Tabla 3.1: Datos de la memoria usada por sigTree con y sin filtros Bloom (BF), número de cache misses de nivel L1 y L2 y factor de velocidad (speedup). Resultados obtenidos usando, para ClamAV los parámetros b=2, ß=6 y tamaño de BF = 2^{14} bits y 100MB de ficheros de tipo .exe, para Bro los parámetros b=2, ß=3 y tamaño de BF = 2^{14} bits y un archivo que contiene un escaneo TCP realizado el 06/03/98 y para DBLife los parámetros b=2, ß=4 y tamaño de BF = 2^{14} bits y un escaneo de una colección de 100MB de páginas web. Fuente sigMatch.

Comparación con ClamAV

Por aquel entonces, ClamAV contaba con 545.191 definiciones de virus en su base de datos, de las

cuales sólo **89.903 firmas** estaban formadas por expresiones regulares. En las pruebas únicamente usaron este conjunto de aproximadamente 90.000 firmas con expresiones regulares. En la figura 3.10 se muestra la comparación entre la tasa de transferencia de ClamAV usando sigMatch y sin usarlo. Para ello se generaron cuatro conjuntos de datos de 100MB cada uno, que fueron clasificados en función del tipo de archivo. El factor de mejora alcanzado varía de 10 a 12X dependiendo del tipo del conjunto de datos. El principal motivo de mejora es la reducción de *cache misses* al usar sigMatch, lo que provoca menos accesos a memoria RAM.

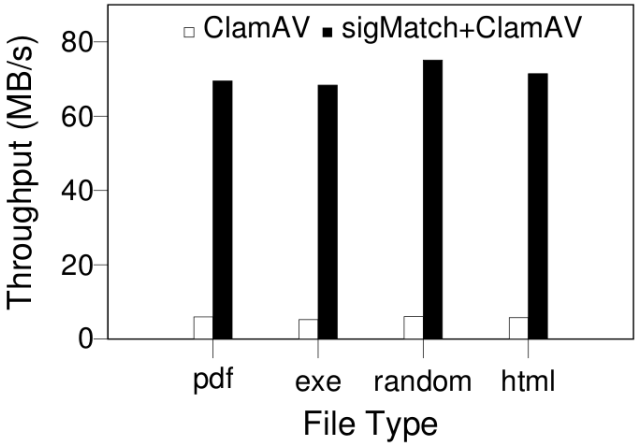

Figura 3.10: Pruebas con ClamAV usando cuatro conjuntos de 100MB de diferentes tipos de ficheros. 89.903 firmas, b=2, ß=6, tamaño del filtro Bloom = 2^{14} bits. Fuente sigMatch.

En este experimento, la estructura de búsqueda de ClamAV requiere cerca de 21MB de memoria y dado que la memoria caché L2 es de 2MB, la mayor parte de la estructura residirá en memoria principal. Por otro lado, la estructura completa sigTree ocupa menos de 3MB con lo que gran parte de ella podrá entrar en caché L2 y si se obviaran los filtros Bloom únicamente necesitaría de 27KB, por tanto podría entrar casi toda la estructura en caché L1. En esta prueba, la tasa de filtrado osciló entre un 93% y un 96%, es decir, sigMatch dejó pasar hacia la unidad de verificación únicamente entre un 4% y un 7% de los datos analizados.

Comparación con Bro

Al evaluar Bro, usaron un conjunto de firmas de Snort que contaba con **1.200 reglas** y una traza de paquetes TCP obtenida de los repositorios de DARPA. En esta prueba usaron el parámetro *ß=3* debido a que las firmas eran de menor longitud a las usadas por la base de firmas de ClamAV. En la figura 3.11 se muestra la comparación entre el uso de Bro con y sin sigMatch pasando como datos cinco capturas TCP, una por cada día de una semana concreta. La tasa de velocidad se ve aumentada en este caso en un factor de 4X y las *cache misses* de las memorias L1 y L2 fueron reducidas en un 76,5% y un 88% respectivamente. No se en especifica en este caso el uso de memoria requerida por la estructura de búsqueda creada por Bro ni su tasa de filtrado.

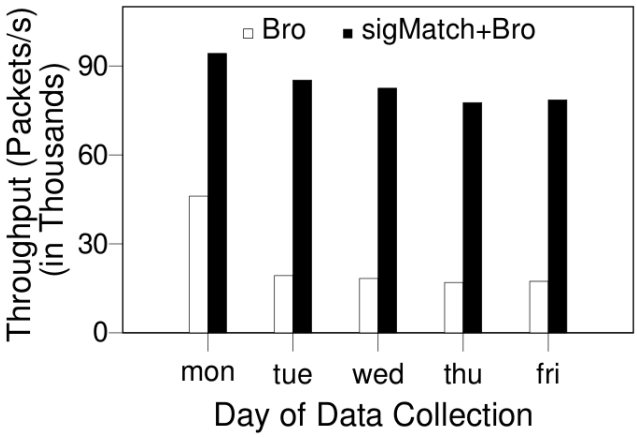

Figura 3.11: Pruebas con Bro usando una traza TCP. 1.200 firmas de Snort, b=2, ß=3, tamaño del filtro Bloom = 2^{14} bits. Fuente sigMatch.

A pesar de que no se proporcionen ni el uso de memoria por parte de la estructura de Bro, ni la tasa de filtrado al combinarlo con sigMatch, se podrían llegar a las siguientes conclusiones:

- El hecho de que Bro cuente con 75 veces menos firmas que ClamAV indica que la estructura de detección de Bro sería bastante menor que la de ClamAV. Por tanto, al tener una estructura de detección menor, se deberían producir menos *cache misses* y de hecho es así, puesto que observando la tabla 3.1 se puede comprobar que en Bro se producen un 26,8% menos de *cache misses* de nivel 1 y un 41,6% menos de nivel 2 que en ClamAV.

- En cuanto a la tasa de filtrado, de la tabla 3.1 se sabe que Bro cuenta con menos *cache misses* que ClamAV y que al incorporar el filtro sigMatch se tiene que ahora Bro cuenta con más *cache misses* que ClamA. Esto podría deberse a que la estructura sigMatch ocupara más memoria en Bro que en ClamAV y fuera esta la culpable de la diferencia entre ambos porcentajes de *cache misses*. Sin embargo, de la tabla 3.1 se conoce que no es así puesto que la estructura sigMatch ocupa menos memoria en la implementación en Bro que en la de ClamAV. Por tanto, la única opción posible que podría atribuirse al incremento de *cache misses* es que la tasa de filtrado sería peor en Bro que en ClamAV y debido a eso se realizarían más llamadas a la unidad de verificación final, lo que provocaría un incremento del número de *cache misses*.

Con estas dos conclusiones se puede entender mejor porqué la tasa de velocidad de Bro se ha visto disminuida drásticamente en comparación con la de ClamAV.

Comparación con DBLife

La base de datos de firmas de DBLife contaba entonces con **60.931 firmas** de expresiones regulares. Se realizaron las pruebas con una colección de 9.914 documentos web de unos 100MB aproximadamente. En este caso la tasa de velocidad se ve incrementada en un factor de 15X y las *cache misses* de nivel 1 y 2 se vieron reducidas en un 80,1% y un 91,4% respectivamente. Tampoco con DBLife se especifica el uso de memoria requerida por la estructura de búsqueda ni la tasa de filtrado al incorporar sigMatch.

Escalabilidad: comparación

Una de las ventajas defendidas por sus autores era la escalabilidad del filtro sigMatch en cuanto al número de firmas que podía soportar. En este punto se evaluará el rendimiento del sistema usando ClamAV con el filtro previo sigMatch y sin él y variando el tamaño de la base de datos de firmas. Para crear conjuntos con mayor número de firmas el equipo de sigMatch desarrolló un generador de firmas con el que consiguieron obtener un conjunto de hasta 300.000 firmas mediante el uso como modelo de la base de firmas original de ClamAV. La figura 3.12 muestra las diferencias en cuanto a la tasa de transferencia de un sistema ClamAV usado con y sin sigMatch para cuatro bases de datos de firmas de diferentes tamaños: 30.000, 90.000, 150.000 y 300.000.

En la tabla 3.2 se muestran más datos acerca de esta prueba de rendimiento, en la que se especifica el uso de memoria de la estructura de ClamAV y sigMatch por separado, las *cache misses* provocadas usando ClamAV con sigMatch y sin él, la tasa de filtrado conseguida y el factor de

mejora alcanzado. Se puede apreciar como el factor de mejora incrementa bastante, de 8,5X a 27,8X cuando se pasan de 30.000 a 300.000 firmas. El principal motivo por el que se tiene esta situación es debido a que desde 30.000 hasta 300.000 firmas el uso de la memoria usada por sigMatch aumenta en un factor menor a 4, de 1,4MB a 5,1MB, mientras que la memoria usada por ClamAV aumenta en un factor cercano a 10, de 8,2MB a 81,1MB. Esto provoca que el incremento de *cache misses* sea considerablemente mayor en ClamAV sin usar sigMatch que usándolo. De hecho, las cache misses de nivel 1 se reducen en mayor proporción a medida que aumenta el número de reglas. Con Bro realizaron experimentos de escalabilidad similares pero los resultados obtenidos no fueron publicados. No obstante, según los autores, se consiguieron unos resultados muy similares.

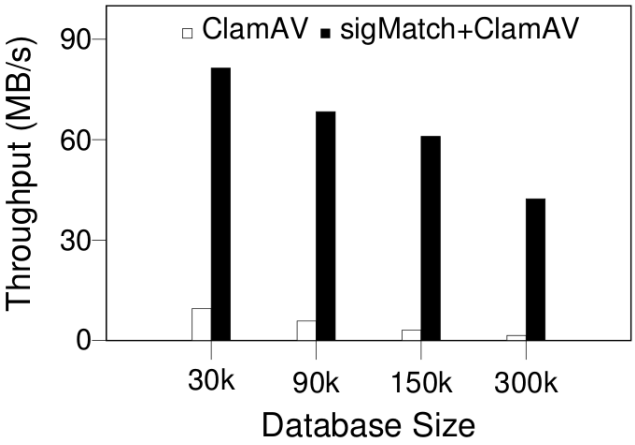

Figura 3.12: Pruebas con ClamAV usando una colección de 100MB de ficheros de tipo .exe y cuatro base de datos de firmas de diferentes tamaños. b=2, ß=6, tamaño del filtro Bloom = 2^{14} bits. Fuente sigMatch.

DB Size	Uso de memoria (en MB)		L1 Cache Misses (en millones)			L2 Cache Misses (en millones)			Tasa de filtrado (*filter rate*)	Factor de velocidad (*speedup*)
	ClamAV	sigMatch	ClamAV	sigMatch +ClamAV	% dec	ClamAV	sigMatch +ClamAV	% dec		
30K	8,2	1,4	352,2	66,3	81,2	42,5	1,5	96,5	0,958	8,5X
90K	21,1	2,7	603,1	76,3	87,3	139,8	3,9	97,2	0,947	11,7X
150K	40,5	3,7	2280,2	93,5	95,9	179,2	6,9	96,1	0,945	19,7X
300K	81,1	5,1	3051,9	121,2	96,0	272,7	19,0	93,0	0,939	27,8X

Tabla 3.2: Datos de la memoria usada por ClamAV y por sigMatch, número de cache misses de nivel L1 y L2, tasa de filtrado (filter rate) y factor de velocidad (speedup). Resultados obtenidos para cuatro tamaños diferentes de bases de datos de firmas, usando ClamAV, los parámetros b=2, ß=6 y tamaño de BF = 2^{14} bits y 100MB de ficheros de tipo .exe. Fuente sigMatch.

3.5. Expectativas en Snort

Hasta este momento nos encontrábamos en la primera fase del proyecto dedicada al estudio de sigMatch, por lo tanto no se había profundizado en el estudio de Snort y la única característica conocida del IDS que pudiera tenerse en cuenta para realizar una estimación del rendimiento que podría tener la incorporación de sigMatch en Snort era el número de firmas usadas.

La base de datos de firmas proporcionada por Snort contenía unas 12.000 reglas, de las cuales, solo unas 5.500 estaban activadas por defecto. Habiendo visto los resultados de las pruebas de rendimiento realizadas por el equipo de sigMatch se podía esperar que el rendimiento de Snort se

viera incrementado notablemente, más aún habiendo realizado también dichas pruebas en Bro, que usaba un conjunto de reglas extraídas de Snort.

En los siguientes capítulos, después de estudiar Snort en profundidad, de ver qué algoritmos de búsqueda de patrones usa y en qué modalidad, de conocer la organización interna de las reglas que lleva a cabo, etc. se redefinirán las expectativas que se tienen de la implementación de sigMatch en Snort.