6. Desarrollo del filtro

6.1. Introducción

Después de haber preparado el entorno de Snort, con el fin de que fuera capaz de albergar un filtrado previo a la búsqueda de patrones, se puede proceder con la fase de desarrollo del filtro. Para llevar a cabo esta fase se rescatarán las ideas de sigMatch expuestas en el capítulo tres y se intentarán implementar en Snort de la manera más óptima posible.

Se explicará en profundidad el desarrollo del filtrado en Snort, partiendo de las últimas ideas expuestas en el capítulo anterior, en las cuales se enumeraban las cuatro funciones principales que formarían parte del filtro y a las que invocaría el motor de búsqueda de patrones de Snort.

Esta explicación se estructurará de tal forma que se divida en tres apartados. El primero de ellos será el relativo a la fase *offline*, en el que se explica la formación de la estructura de búsqueda del filtro sigMatch. En el segundo apartado, definido como fase *online*, se definirá cómo se realiza la búsqueda de patrones dentro de la estructura creada en la fase *offline*. En el tercer y último apartado se hará un repaso por los parámetros configurables del filtro, como son los parámetros del filtro *b* y *ß*, el tipo de implementación realizada de sigMatch y otros parámetros relacionados con los filtros Bloom, como el tamaño de la matriz o las funciones hash empleadas; se aclarará la importancia de cada uno de ellos, para que de esta forma el usuario pueda realizar una elección más fundamentada y acorde a las necesidades de un entorno específico. Será en el siguiente capítulo donde hablará de la eficiencia de la implementación llevada a cabo y se mostrarán los resultados obtenidos en las pruebas de rendimiento.

Para terminar, y en un contexto distinto de sigMatch, se explicará brevemente lo más destacado del desarrollo de la segunda línea de investigación propuesta en el capítulo uno. Se recuerda al lector que en ella se valoraba la posibilidad de aprovechar al máximo las capacidades que ofrecen los procesadores actuales.

6.2. sigMatch

Ya en el capítulo anterior se introdujo el diseño de la estructura que tendría la implementación del filtrado en Snort. Esta estructura será el punto del que partirá la explicación del desarrollo del filtro previo.

Como ya se explicó entonces, la implementación gira en torno a cuatro funciones principales

contenidas en un nuevo fichero denominado `redb.c`. Estas funciones son `redbNew`, `redbAddPattern`, `redbCompile` y `redbSearch`. Las tres primeras formarán parte de la fase *offline* del filtro, mientras que la última compondrá la fase *online*.

Estas cuatro funciones serán invocadas por el motor de búsqueda de patrones, es decir desde el fichero `mpse.c`, e interactuarán con este módulo de Snort para indicarle qué paquetes pueden ser descartados del análisis definitivo y cuáles no.

En lo que sigue de apartado, se explicará el funcionamiento de cada función por separado, comenzando en primer lugar por las funciones de la fase *offline* y continuando con la función que forma la fase *online*. Servirá como apoyo el cuarto anexo, en el que se mostrarán las funciones más destacadas que forman parte de la implementación del filtrado. Una vez vistas ambas fases, se explicarán también las opciones configurables del filtro, como la elección de funciones hash y los parámetros *b* y *ß*.

6.2.1. Fase *offline*

Siguiendo el diagrama de secuencia mostrado en la figura 5.11 del capítulo anterior, se puede ver cuál es el orden, que sigue el motor de búsqueda de patrones de Snort, a la hora de invocar a las funciones que forman la fase *offline* del filtrado previo. En primer lugar, se llama a la función `redbNew`, que será la encargada de reservar el espacio de memoria necesario para la estructura `REDB_STRUCT`. Esta estructura contendrá, entre otros, los enlaces a los diferentes grupos de *q-grams* y filtros Bloom, la estructura de búsqueda en forma de árbol y los patrones de las firmas originales, además de información relacionada con el número de nodos del árbol, el número de *q-grams* significativos, etc.

Una vez asignada en memoria la estructura e inicializadas las variables a sus valores por defecto, se realiza la llamada a la función `redbAddPattern`, que será la encargada de añadir patrones a la estructura `REDB_STRUCT`. Cada patrón enviado a la función `redbAddPattern` se almacena en una estructura `REDB_PATTERN`, que contendrá el propio patrón, la longitud del mismo y el número de *q-grams* válidos que puede llegar a proporcionar el patrón. Esta estructura se añadirá a la lista enlazada de patrones que cuelga de `REDB_STRUCT`.

Después de haber añadido todos los patrones a la estructura `REDB_STRUCT` se procede con la compilación de la estructura de búsqueda del filtro. Para ello, el motor de búsqueda, mediante la función `mpsePrepPatterns`, invoca a `redbCompile`, que se encargará de actuar del modo descrito en el capítulo tres para así poder construir la estructura de búsqueda propuesta por sigMatch.

Antes de continuar, para conseguir una mayor comprensión de lo que queda de apartado, se recomienda encarecidamente al lector repasar el capítulo tres, en el que se explica la creación de la estructura de búsqueda del filtro sigMatch. De lo contrario, las explicaciones restantes de la fase *offline* resultarán de muy difícil comprensión.

Seguidamente, se muestra una lista resumida de los pasos a realizar para la obtención de la estructura sigTree, partiendo de una base de firmas originales:

1. Obtención de los *q-grams* de cada patrón original.
2. Ordenación de los *q-grams* por frecuencia de repetición.
3. Obtención de la lista de *q-grams* representativos, es decir de todos los *q-grams* que son capaces de representar a cada uno de los patrones de la base de firmas originales.
4. Construcción del árbol de altura *b*.
5. Construcción de los filtros Bloom y de la lista enlazada de patrones cortos.

Siguiendo estos pasos, se explicará cómo a partir de la función `redbCompile` se consigue crear la estructura de búsqueda propuesta por sigMatch, a partir de los patrones originales. Esta explicación se realizará a nivel de código fuente. En la descripción se hará mención a algunas de las funciones que se han desarrollado para implementar el filtrado previo. Estas funciones se encuentran a disposición del lector en el anexo IV de este documento.

1. Obtención de los *q-grams* de cada patrón original
   - Al recopilar los *q-grams* también habría que almacenar de alguna forma la frecuencia con la que se repite cada uno de ellos. Para ello se crea el vector `qgramCount`, cuyo tamaño será el correspondiente al número de *q-grams* posibles en función del parámetro *b* –por ejemplo, si *b=2* y el número de caracteres posibles es 256, entonces el número total de *q-grams* posibles será 256^2, que equivale a 65536–. En un principio, todos los posibles *q-grams* parten con frecuencia cero y ésta se incrementa a medida que vayan apareciendo *q-grams* representativos en los patrones.
   - Una vez creado e inicializado el vector `qgramCount`, se recorren todos los patrones almacenados en la estructura `REDB_STRUCT` y se ejecuta la función `get_qgrams()`, que se encarga de aumentar la frecuencia de los *q-grams* significativos del patrón. Esta función tiene especial cuidado en no añadir *q-grams* duplicados a la lista de *q-grams*.
   - Si el patrón no tuviera una longitud suficiente, es decir si su longitud fuera menor que el parámetro *b*, entonces dicho patrón se añade a una lista de patrones muy cortos cuya estructura es `REDB_NOQG`.
2. Ordenación de los *q-grams* por frecuencia
   - A partir del vector `qgramCount`, se obtendrá una lista de *q-grams* ordenada por frecuencia, que será almacenada en la matriz `qgramSorted`. Esta matriz contará con cuatro columnas y con tantas filas como número de *q-grams* significativos se hayan obtenido en el paso anterior. La primera columna contendrá el valor del *q-gram*, la siguiente almacenará su frecuencia y en las dos últimas se guardará el valor correspondiente al número de veces que aparece el *q-gram* en un patrón largo y en uno corto, respectivamente.
   - Después de haber creado la matriz `qgramSorted`, se procede a ordenar los *q-grams* por frecuencia, para ello se invoca a la función `sort_qgramsHi2Lo()`, que almacenará en `qgramSorted` los *q-grams* y sus frecuencias por orden de mayor a menor frecuencia de *q-gram*. Esto se consigue usando la información proporcionada por el vector `qgramCount`.

3. Obtención de la lista de *q-grams* representativos

   - Una vez que se tiene la lista de *q-grams* ordenados por frecuencia, el siguiente paso es obtener otra lista de *q-grams*, normalmente más reducida, en la que el conjunto de sus *q-grams* pueda representar al conjunto de los patrones originales. Para ello se invoca a la función `build_qgramCover()`, que lleva a cabo este procedimiento.

   - Una vez dentro de la función `build_qgramCover()`, cada vez que se obtiene un *q-gram* representativo se pasa el control a la función `add_qgbf()`, que se encarga de reservar en memoria una instancia de la estructura `REDB_QGBF` y de añadir la cadena de texto, que sigue al *q-gram*, a la lista de patrones del filtro Bloom –si la longitud del patrón original es mayor o igual que $b+\beta-$ o a la lista enlazada de patrones cortos –si, por el contrario, la longitud es mayor que b y menor que $b+\beta-$.

   - Cada instancia de una estructura `REDB_QGBF` contiene un identificador que corresponde unívocamente con el *q-gram* correspondiente. Por tanto, cuando un único *q-gram* representa a más de un patrón, estos se almacenan en una misma estructura `REDB_QGBF`, y cada uno de los patrones se almacenará convenientemente en dicha estructura en función de la longitud que posea: mayor o igual que $b+\beta$ (firma normal), mayor que b y menor que $b+\beta$ (firma corta), o menor o igual que b (firma muy corta).

   - Una estructura `REDB_QGBF` proporciona información necesaria para la fase *online*. Si en dicha fase, al inspeccionar un paquete de red se encuentra un *q-gram* representativo, su estructura `REDB_QGBF` indicará si se debe chequear el filtro Bloom, la lista enlazada, ambos o directamente marcar el paquete como candidato y enviarlo al análisis definitivo. Esto último sucederá cuando el resumen obtenido represente a una firma muy corta, lo que significará que al encontrar el *q-gram* de longitud b en el flujo de datos, también se habrá encontrado la firma representada, puesto que la longitud de esta última será menor o igual a b. En estos casos, es cuando el paquete se marca en ese mismo momento como candidato.

   - Posteriormente, y para optimizar el uso de memoria usada, se invoca a la función `reallocate_qgbf()`, que tratará de reducir en la medida de lo posible el consumo de memoria de la estructura `REDB_QGBF`.

   - Una vez que se han completado las estructuras `REDB_QGBF`, ya se tiene toda la información relativa a los *q-grams* representativos y a las cadenas de texto que completarán los filtros Bloom o las listas enlazadas de patrones cortos.

4. Construcción del árbol (NFA) de altura b.

   - Para construir el árbol se invoca a la función `build_rbtree4()`, que devolverá un puntero a la estructura `ROOT_NODE`. Esta estructura será de donde parta el árbol de altura b. De ella colgarán los nodos hojas, almacenados en las estructuras `LEAF_NODE`. En función de la altura del árbol, de los nodos hojas colgarán más nodos hojas o nodos finales que apuntarán a estructuras `REDB_QGBF`.

   - La función `build_rbtree4()` invocará a las funciones recursivas `create_qbranch4()` y/o `create_nbranch4()`.

   - La primera de ellas será ejecutada cuando se desee crear una rama que vaya a simbolizar

a un *q-gram* representativo. En este caso se crea la rama hasta llegar a una altura *b*, finalizando la rama con la estructura `REDB_QGBF` correspondiente con el *q-gram* en cuestión.

- La segunda, sin embargo, se invocará cuando existan patrones muy cortos de longitud menor que *b*. En estos casos se sabe que el patrón no puede completar la altura del árbol, por tanto se rellenarán todos los casos posibles hasta llegar a la altura *b*, y una vez allí se colocarán nodos finales cuya estructura `REDB_QGBF` devuelva directamente el paquete como candidato, sin pasar por un filtro Bloom o una lista de patrones cortos.

5. Construcción de los filtros Bloom

   - En este punto se invoca a la función `build_bloomFilter()`, que construye los filtros Bloom de cada estructura `REDB_QGBF` que lo requiera.

   - La estructura `REDB_QGBF` es la que contiene la información que indica si se debe construir un filtro Bloom para un *q-gram* en concreto. Esto se producirá cuando dicha estructura represente patrones de longitud mayor o igual a $b+\beta$.

   - Para ello, en primer lugar se crearán las matrices de los filtros Bloom con un tamaño fijo o variable, en función del número de patrones que se vayan a añadir al filtro –esta opción es configurable a través del archivo `redb.h`–. La creación de las matrices se realiza a través de la función `bloom_createMatrix()`.

   - Una vez que las matrices han sido creadas e inicializadas a cero, se deben rellenar. Para ello se invoca a la función `bloom_addStrings()`, que obtendrá los resúmenes hash de los patrones de longitud normal almacenados en la estructura `REDB_QGBF`. Cada resumen hash que se obtenga se usará para especificar qué bit de la matriz deberá tomar un valor uno. El tipo de función hash que se desee usar, así como la cantidad de ellas, se indicará en el fichero `redb.h`.

A modo de resumen, en la figura 6.1 se mostrará el diagrama secuencial de la función `redbCompile`, en la que aparecerán, en orden de ejecución, todas las funciones anteriormente descritas y explicadas.

Llegados a este punto, se puede considerar finalizada la fase *offline*. A partir de aquí se devuelve el control al motor de búsqueda de patrones para que continúe con su ejecución normal y comience con la recepción de paquetes de la red.

6.2.2. Fase *offline*

Después de haber creado la estructura de búsqueda del filtro, no queda más que comenzar con la fase *online*, en la que se realizará la búsqueda de patrones resumidos dentro de cada paquete que atraviese Snort. Para ello, el motor de búsqueda de patrones fue modificado de tal manera que, antes de la llamada a la unidad de verificación definitiva, invocaría al filtro previo. Este filtro tomaría la decisión de enviar el paquete como candidato al algoritmo de búsqueda definitivo, o por el contrario lo descartaría, redirigiéndolo hacia la salida del motor de búsqueda.

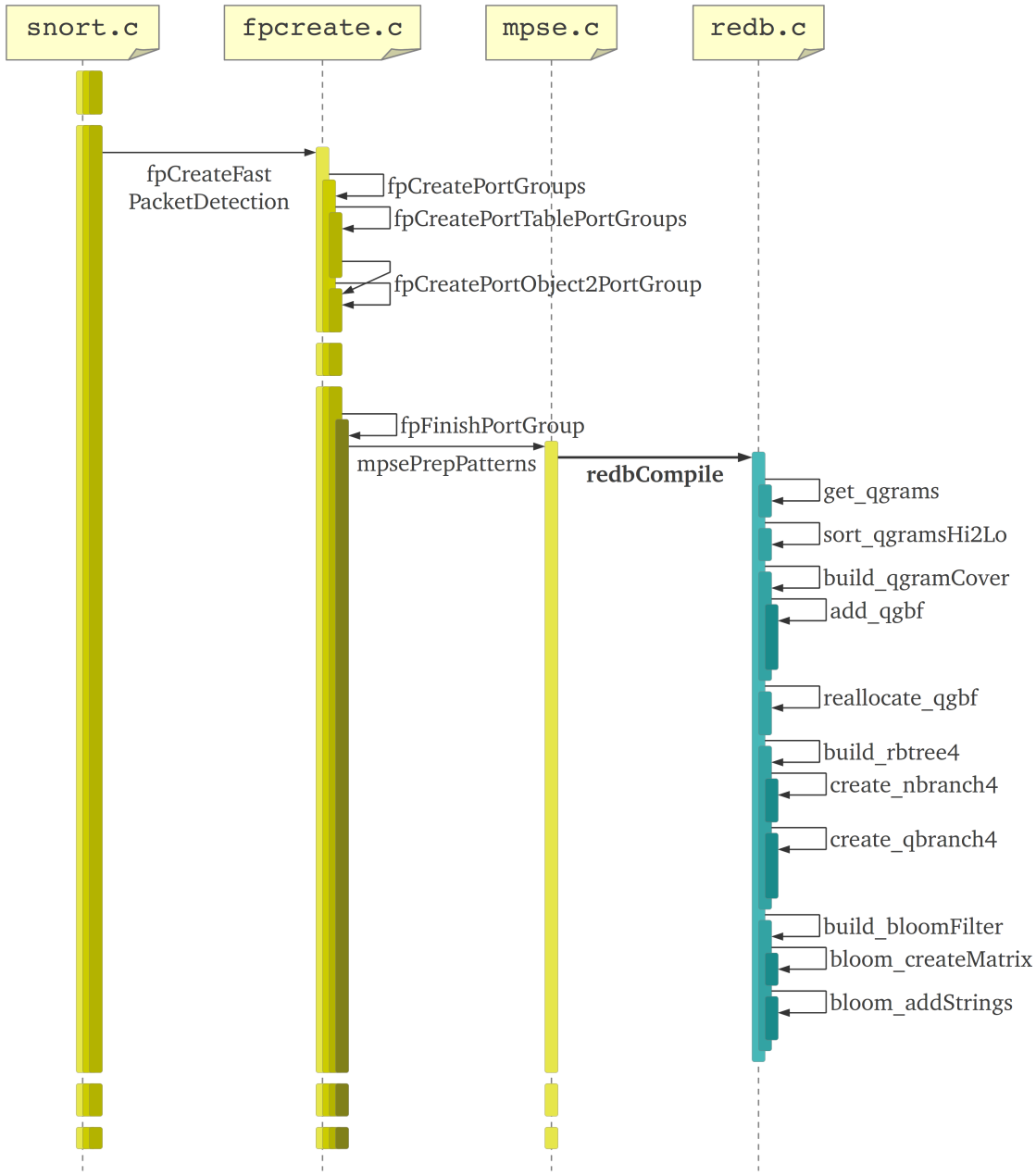


Figura 6.1: Diagrama de secuencia de la función redbCompile, correspondiente a la fase offline del filtrado previo introducido en el motor de detección de Snort.

Para llevar a cabo esta búsqueda, se ejecutará la función `redbSearch4`. Esta función recibirá como parámetros el payload del paquete en cuestión, su longitud y la estructura `REDB_STRUCT`, que contendrá, a su vez, la estructura de búsqueda del filtro, también conocida como sigTree. En esta función, la estructura `REDB_QGBF` juega un papel importante. En la fase *offline* se ha explicado en qué consiste la estructura de búsqueda de patrones del filtro –denominada sigTree–: comienza con un árbol de decisión de altura *b* y en los nodos finales se colocan las estructuras `REDB_QGBF`.

En la búsqueda de patrones del filtro, el hecho de alcanzar un nodo final significa que se ha

encontrado un *q-gram* dentro del paquete que está siendo analizado. La información relativa al *q-gram* –tipo y cantidad de patrones representados– se encuentra almacenada en la estructura `REDB_QGBF`, a la que se accede a través del nodo final del árbol. De esta forma, al llegar a un nodo final, la estructura `REDB_QGBF` es la que indica cuáles son los siguientes pasos que se deben ejecutar: marcar el paquete como candidato, analizarlo en el filtro Bloom o comprobar la existencia de patrones cortos

Se recuerda que podían haber firmas muy cortas, cortas o normales: las firmas muy cortas serían las que tuvieran una longitud menor o igual a b, las cortas se corresponderían con longitudes mayores que b y menores que $b+\beta$ y las normales estarían formadas por patrones cuya longitud fuera mayor o igual a $b+\beta$.

También, se sabe que más de una firma original podría ser representada por un *q-gram*. Por lo tanto, un mismo *q-gram* podría representar tanto firmas cortas como normales, contemporáneamente. Más adelante se hará de mención al tratamiento de las firmas muy cortas, que ya se explicó en el punto anterior, denominado fase *offline*.

Esta situación, que en principio puede resultar algo ambigua a la hora de tener que realizar una búsqueda dentro de ella, se aclara en este párrafo. Al llegar a un nodo final del árbol de altura b, se sabe que se alcanza a una estructura `REDB_QGBF`. Esta estructura servirá para decidir cuales serán los próximos pasos. La figura 6.2 puede servir como ayuda a la explicación que sigue a continuación:

- La variable `nostr` contabiliza el número de firmas originales, representadas por el *q-gram* en cuestión, que no tienen la suficiente longitud como para proporcionar un patrón que pueda ser añadido a una lista enlazada de patrones cortos o a una matriz dedicada al filtro Bloom.
    - En el caso en el que esta variable sea mayor que cero, significará que hay, al menos, una firma original cuya longitud sea menor o igual a b. Lo que significa que, al haber alcanzado un nodo final, la firma original ha debido ser detectada. Por tanto, el paquete deberá ser marcado como candidato para que el algoritmo de verificación definitiva se encargue de evaluarlo.
- La variable `bfstr` contiene los patrones que se incluyen en la matriz que forma el filtro Bloom. Esta variable estará vacía si ninguna de las firmas originales representadas por el *q-gram* en cuestión es mayor o igual a $b+\beta$.
    - Siempre que el paquete no se haya marcado previamente como candidato – es decir si el valor de `nostr` es cero–, y la variable `bfstr` no esté vacía, se evaluarán los caracteres necesarios del paquete en el filtro Bloom. Si el filtro Bloom resuelve de forma positiva, se marcará el paquete como candidato para una posterior evaluación por parte de la unidad de verificación final.
- La variable `shortstr` contiene los patrones que se incluyen en la lista enlazada de patrones cortos. Estos patrones provienen de las firmas originales, representadas por el *q-gram*, cuyas longitudes sean superiores a b e inferiores a $b+\beta$. Si no se da esta circunstancia, la variable `shortstr` permanecerá vacía.

- En el caso de que, en primer lugar, la variable `nostr` sea igual a cero y, en segundo lugar, la variable `bfstr` esté vacía o, no siendo esta la situación, el filtro Bloom haya devuelto un resultado negativo, se procederá, siempre y cuando la variable `shortstr` no esté vacía, con la comprobación de los siguientes caracteres del paquete con la lista enlazada de patrones cortos almacenada en `shortstr`. Si los siguientes caracteres coincidieran con algún patrón almacenado en la lista enlazada de patrones cortos, entonces se marcaría el paquete como candidato, en caso contrario, se descartaría de su análisis definitivo.

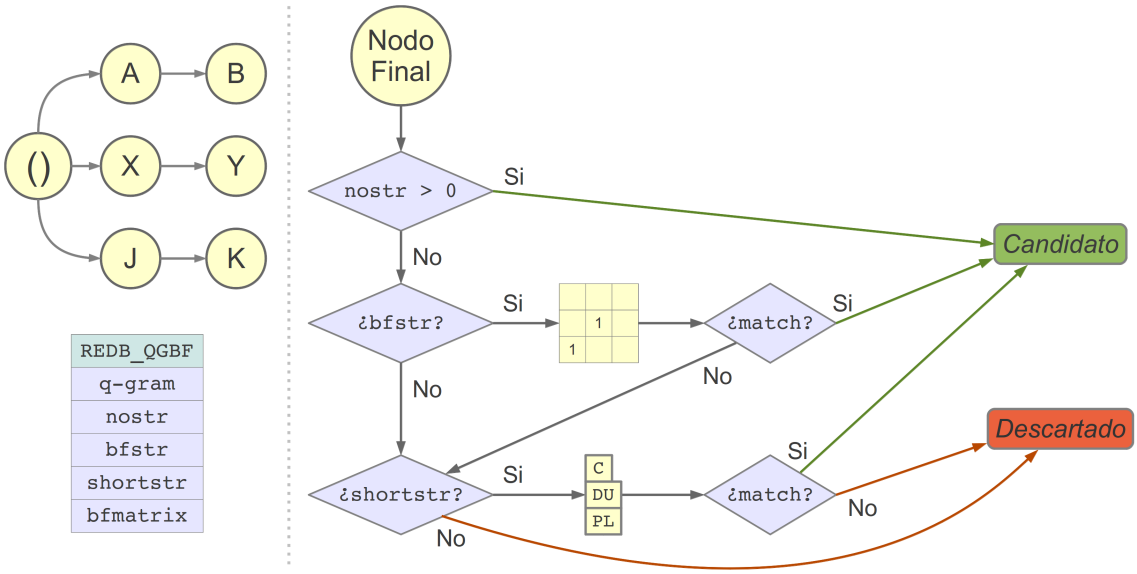


Figura 6.2: Funcionamiento de la búsqueda dentro de la estructura sigTree.

Ya se explicó cómo se procedía con las firmas muy cortas en cuanto a la creación de la estructura de búsqueda sigTree. Se rellenaba el árbol con todas las posibilidades hasta alcanzar la altura *b*. Posteriormente se colocaba en cada nodo final una estructura `REDB_QGBF` cuya variable `nostr` fuera mayor que cero. De esta forma, si en la fase *online* se alcanzaba algún nodo final correspondiente a una firma muy corta, se procedería como se ha explicado anteriormente, marcando el paquete como candidato y devolviéndolo al motor de búsqueda de Snort para que se encargara de enviarlo a la unidad de verificación final.

A continuación, se detalla un ejemplo, ilustrado gráficamente en la figura 6.3, del comportamiento que se adopta en la fase *online* ante un conjunto de firmas concretas:

> Se cuentan con 8 firmas originales. Al tratarlas en la fase *offline*, y considerando los parámetros *b=2* y *ß=4*, se extraen 3 *q-grams* que representan al conjunto formado por las 8 firmas originales. Estos *q-grams* son `CO`, `AC` y `EC`, y son almacenados en la estructuras `qgbf_1`, `qgbf_2` y `qgbf_3`, respectivamente. En cada estructura se almacenará la información necesaria relativa a las firmas originales representadas y que posteriormente será necesaria en la fase *online*.
>
> Una vez construida la estructura de búsqueda sigTree, en la figura 6.3 se han añadido

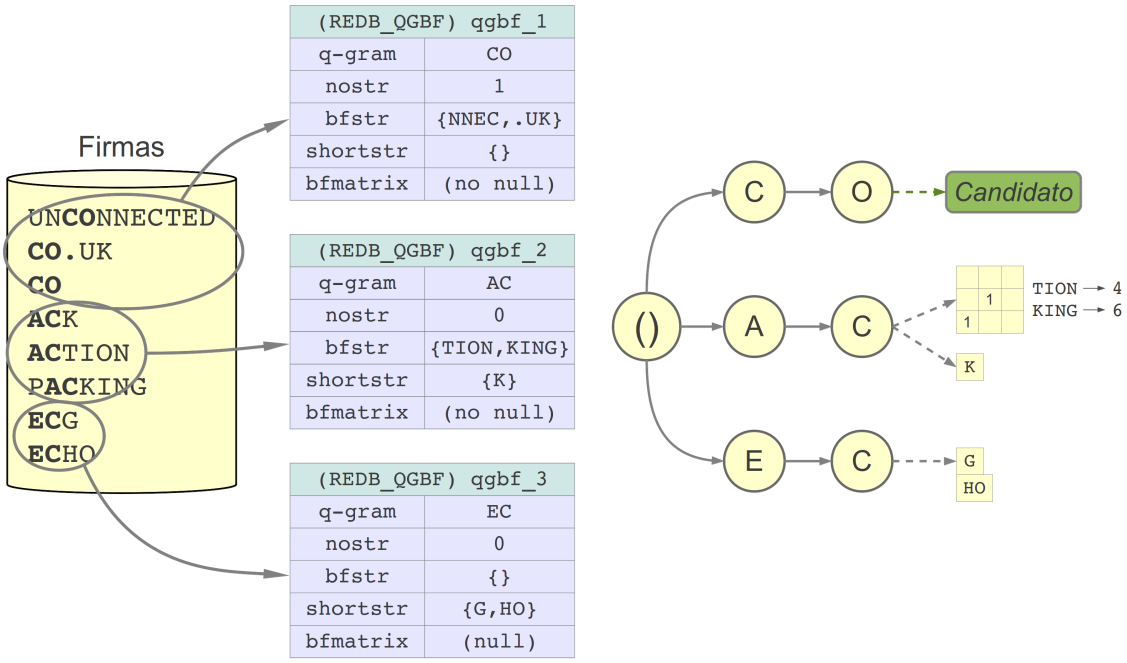


Figura 6.3: Ejemplo de búsqueda en la estructura sigTree.

también las decisiones que se tomarán en cada caso, dependiendo de la información contenida en cada estructura `REDB_QGBF`.

En la rama superior del árbol, que representa al *q-gram* `CO`, al llegar al nodo final se marca directamente el paquete como candidato. Esto es así puesto que la variable `nostr` es mayor que cero. Efectivamente, esto sucede porque `CO` es también una firma original y, por lo tanto, si se ha alcanzado dicho nodo final, es debido a que la firma original ha sido hallada en el paquete. En el caso de una posible firma original muy corta `C` la variable `nostr` también sería mayor que cero.

En la rama intermedia, al llegar al nodo final se debe comprobar el resultado del filtro Bloom. En caso de ser positivo se marcaría el paquete como candidato. En el caso contrario se comprobaría la existencia de la firma en la lista enlazada de patrones cortos. Si se produjera una coincidencia se marcaría el paquete como candidato, mientras que se descartaría en el caso contrario.

En la última rama, sólo existen firmas originales cortas, por lo tanto únicamente se comprueban las posibles coincidencias con los patrones de la lista enlazada de patrones cortos. De igual forma, ante una coincidencia se marcaría el paquete como candidato, y en caso contrario se descartaría.

6.2.3. Parámetros configurables

Como se explicó en el tercer capítulo, y se ha podido ver de nuevo en la explicación de este apartado, el filtro cuenta con ciertos parámetros configurables. Esta configuración se puede realizar

a través del fichero `redb.h`. En él se podrán modificar los parámetros *b* y *ß* del filtro, el número y el tipo de funciones hash que se deseen utilizar, el tamaño de la matriz del filtro Bloom y el tipo de implementación deseada para la estructura de búsqueda sigTree.

A continuación, se explicará cada una de las cuatro opciones configurables. En primer lugar, se describirán las cuatro implementaciones desarrolladas, explicando cómo evolucionaron y qué rendimiento ofrece cada una. Posteriormente, se hablará de los parámetros de filtrado, aclarando que su elección dependerá mucho de la longitud de las firmas originales. En tercer lugar, se abordará en profundidad todo lo relativo a las funciones hash usadas en los filtros Bloom. Y por último, también relacionado con los filtros Bloom, se explicará la importancia del tamaño de las matrices de tales filtros.

6.2.3.1. Implementaciones

Para la estructura de búsqueda se han desarrollado cuatro implementaciones distintas. La primera de ellas se basa en una estructura en árbol –también conocida como NFA o *trie*– de altura fija *b*. En esta estructura se introducen las firmas cuya longitud sea mayor o igual a *b*, almacenando las firmas muy cortas en una lista enlazada formada únicamente estructuras de firmas muy cortas.

No todos los grupos de reglas contienen firmas muy cortas, de hecho son una minoría. No obstante, si un grupo en concreto contiene alguna firma corta, en esta primera implementación, se debería recorrer en primer lugar la estructura de firmas cortas en búsqueda de alguna coincidencia, si no se produjera ninguna, entonces se recorrería el árbol de manera normal. Esto hace que la implementación sea algo ineficiente y algo más lenta de lo que debiera ser.

Para evitar el problema anterior, la segunda implementación se enfoca de manera diversa, construyéndose también una estructura basada en árbol, aunque en este caso de altura variable. De esta forma se consigue que las firmas muy cortas tengan cabida en él y, por tanto, no sea necesario comprobar la existencia de firmas muy cortas en una lista enlazada, ahorrando bastante tiempo de ejecución. Al tratarse de un árbol de altura variable, cada nodo de la estructura puede ser de dos tipos, final o intermedio, por lo que en cada transición se debe comprobar la naturaleza del nodo. Esta continua comprobación al cambiar de nodo quizás sea el motivo por el que esta implementación no produjera los resultados esperados, ofreciendo aún un rendimiento insuficiente.

En la tercera implementación, se da un giro radical y se construye una estructura de búsqueda basada en matriz. Dicha matriz cuenta con tantas dimensiones como indique el parámetro *b*, y la longitud de cada dimensión la dictamina el número de caracteres posibles, en este caso 256. Por tanto, la matriz tendrá una dimensión de 256^b. Para evitar un exceso en el consumo de memoria se fija el valor del parámetro *b* a 2, de esta forma la matriz no alcanzará un tamaño desmesurado.

El funcionamiento consiste en posicionarse en el elemento de la matriz indicado por dos caracteres del contenido del paquete que se está analizando, cada carácter corresponde con una dimensión de la matriz y el conjunto de ambos caracteres forman un posible *q-gram*. Si este *q-gram* se corresponde con un resumen representativo de las firmas originales, entonces en el elemento correspondiente de la matriz se encontrará un puntero a una estructura `REDB_QGBF`, en caso contrario el elemento tendrá un valor nulo.

Idealmente, esta implementación podría considerarse la más eficiente, puesto que se necesitan pocas operaciones para acceder a la estructura `REDB_QGBF`. Sin embargo, el uso de memoria es mayor que en el caso de estructuras basadas en árbol, por lo que el rendimiento se ve reducido. En este momento, el lector podría pensar que, aunque la estructura fuera mayor, quizás este factor no sería suficientemente significativo como para afectar demasiado al rendimiento, puesto que la estructura de búsqueda se carga una sola vez en memoria caché, y una matriz de estas características no parece ser tan grande como para no caber en ella. En la práctica, esta implementación resultó ser menos eficiente que las dos anteriores.

En este punto se debe hacer un inciso para aclarar algunas posibles dudas. A lo largo del documento se ha explicado que las firmas originales se dividen en varios grupos, por lo que si, por ejemplo, hay 100 paquetes que atraviesan Snort, puede darse el caso de que cada uno de ellos pertenezca a un grupo distinto. Esto provoca que por cada paquete que pase por Snort, para ser analizado, se deba cargar en memoria una estructura de búsqueda distinta. De esta manera, el tamaño de la estructura de búsqueda se convierte en un factor aún más crítico de lo que se podía pensar en un principio.

Para entender mejor esta problemática se presenta un caso práctico: suponiendo que se tienen dos implementaciones denominadas X e Y con estructuras de búsqueda distintas, que la implementación X cuenta con un tiempo de análisis de paquete mayor que el de la implementación Y, pero que la estructura de la variante X ocupa un espacio menor en memoria que la de la variante Y –siendo ambas lo suficientemente pequeñas como para poder ser almacenadas en memoria caché–. A continuación, se presentan dos situaciones en las que se puede comparar la eficiencia de ambas implementaciones.

En la primera situación se supone que por Snort pasan cinco paquetes y que todos ellos corresponden al mismo grupo de reglas, por lo que la estructura de búsqueda se carga sólo una vez en memoria. En la segunda situación, sin embargo, cada uno de los cinco paquetes pertenecen a grupos de regla distintos, por lo que con la llegada de cada paquete se debe cargar en memoria una nueva estructura de búsqueda.

Este ejemplo se muestra en la figura 6.4. En él se puede observar como la implementación Y resulta ser más rápida en la primera situación, a pesar de que su estructura de búsqueda ocupe más espacio. Esto es así porque la estructura únicamente se carga una vez en memoria caché en todo el proceso de análisis. Sin embargo, en la segunda situación, en la que en se cargan tantas estructuras en memoria como paquetes lleguen a Snort, resulta bastante más rápida la implementación X.

Cuando una base de firmas produce una gran cantidad de grupos, y estos grupos tienen una población relativamente uniforme de reglas, se producirán más situaciones como la segunda, en la que frecuentemente deberán cargarse nuevas estructuras de búsqueda. En un conjunto de firmas normal, como las proporcionadas por Sourcefire, en cierta manera se tiende a esta situación.

Por tanto, comparando las implementaciones del ejemplo con las reales, se podría considerar que la segunda implementación, basada en árbol, se trataría de la implementación X, mientras que la tercera, basada en matriz, se asemejaría más a la implementación Y.

Debido a esto, en la cuarta y última implementación se vuelve al concepto de árbol. En este caso, la implementación está basada en árbol de altura fija, pero se consiguen incluir en él las firmas muy cortas –como ya se explicó en los apartados correspondientes a las fases *offline* y *online*–. De esta

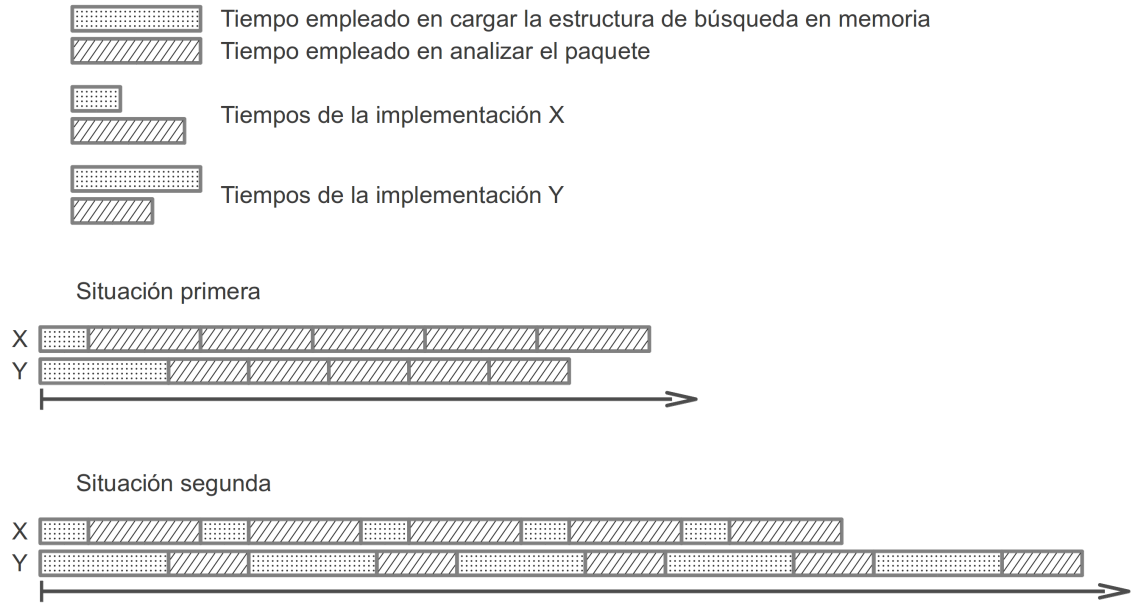


Figura 6.4: Supuesto práctico relacionado con la tercera implementación.

forma se evita la continua comprobación del tipo de nodo. Se conocerá si se trata de un nodo final o intermedio en función de la profundidad alcanzada en el árbol.

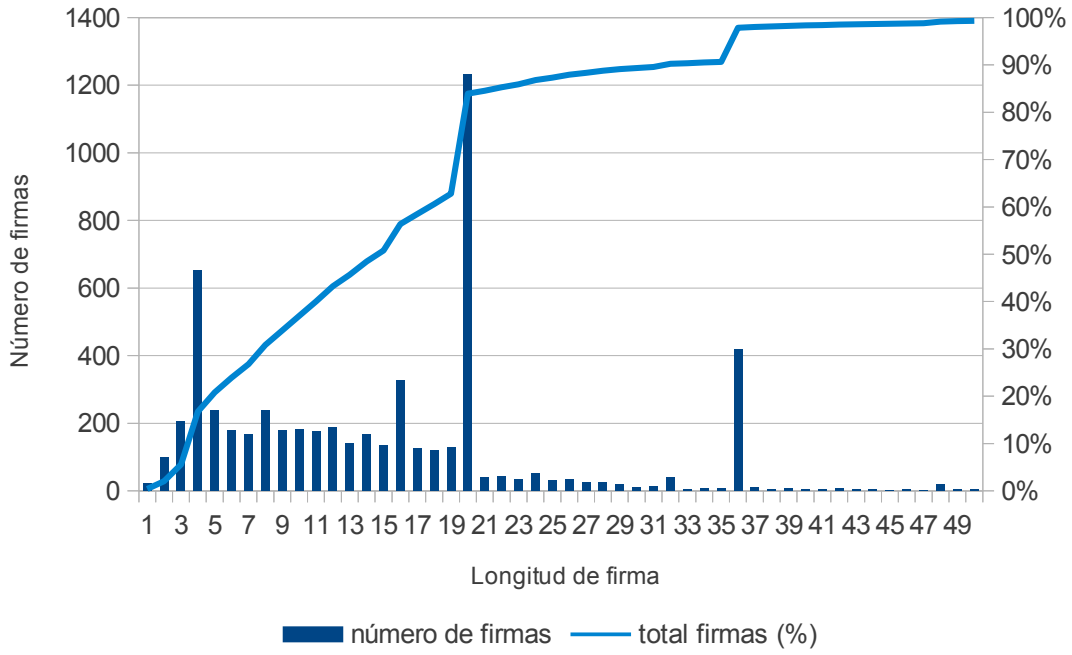
Esta última implementación es la que se usa por defecto, puesto que ofrece los mejores resultados en cuanto a rendimiento. Aún así, en el fichero `redb.h` se brinda la posibilidad de cambiar el tipo de implementación.

6.2.3.2. Parámetros del filtro

La configuración de los parámetros *b* y *ß* debería depender de las longitudes de las firmas originales. A priori, el conjunto de firmas usadas suele ser actualizado cada poco tiempo, por lo que las longitudes de firmas también cambiarían. No obstante, con cada actualización, la distribución de la longitud de las firmas permanece casi inalterada.

A través de la gráfica 6.1 se puede observar una distribución del número de firmas en función de su longitud –la base de firmas utilizada corresponde a un conjunto de 5.551 reglas–. Las columnas representan el número de firmas que comparten la longitud indicada en el eje de abscisas; mientras que la línea representa el porcentaje de firmas cuya longitud es menor o igual a la señalada por el eje de abscisas.

La distribución mostrada corresponde a un conjunto de firmas proporcionadas por Sourcefire después de haber pasado por Snort y haber truncado parte de sus firmas a una longitud máxima de 20 caracteres. Se recuerda que esta opción se podía establecer en el fichero de configuración de Snort, pero no ha de afectar a todas las reglas, existen algunas reglas, que debido a sus opciones, no pueden ser truncadas, de ahí que haya reglas que tengan una longitud superior a 20 caracteres. Este valor de longitud máxima es la que viene configurada por defecto en Snort.



Gráfica 6.1: Distribución de firmas en función de la longitud.

En esta distribución, se puede observar que más del 80% de las firmas tienen una longitud menor o igual a 20 caracteres, mientras que más del 97% de las firmas no superan los 36 caracteres. Conociendo estos datos, se podría pensar que una configuración razonable de la longitud de los resúmenes de sigMatch debería ir de 4 a 6 caracteres. No se puede precisar cuál es el valor óptimo para *b* y *ß*, sin embargo se pueden hacer pruebas con varios valores. Si se establece un valor *b* igual a 2, entonces el parámetro *ß* podría variar de 2 a 4. Por defecto, se tomará un valor *2+3*, aunque en el próximo se mostrarán resultados para distintos valores de *ß*.

6.2.3.3. Filtro Bloom: funciones hash

La elección del número y del tipo de funciones hash es otro aspecto que puede ser configurado en el filtro. En el tercer capítulo se estudió la problemática que giraba en torno al uso de filtros Bloom. En dichos filtros podían producirse colisiones; se recuerda que una colisión era el hecho de que patrones diversos devolvieran resúmenes que se correspondieran con una misma posición en la matriz del filtro Bloom. Las colisiones provocaban que se aumentara la tasa de falsos positivos, lo que iba en detrimento de la eficiencia del filtro, empeorando su tasa de filtrado.

Para disminuir la tasa de colisiones, se recuerda que se proponía el uso de más de una función hash; de esta forma disminuía bastante la tasa de falsos positivos, lo que afectaba de manera positiva en el rendimiento. No obstante, el uso de más de una función hash provoca que el tiempo de análisis aumente considerablemente, por lo que el rendimiento podría disminuir si, con esta medida, la tasa de filtrado no mejorara lo suficiente. Para conseguir el mejor rendimiento posible se debe buscar un equilibrio del número de funciones hash empleadas.

También se sabe que, dependiendo del tipo de función hash utilizada, se puede obtener una tasa de colisiones mayor o menor, así como un tiempo de computación diverso. Que una función hash proporcione una tasa de colisiones menor no siempre se traduce en un tiempo de ejecución mayor, o viceversa.

Por tanto, para la elección del tipo de función hash se pueden tener en cuenta dos criterios: el tiempo de ejecución y el número de colisiones que devuelve. El último de ellos se cuantificará en función de la distribución de frecuencia de los resúmenes proporcionados por la función hash. Esta distribución indica el número de veces que se repite un determinado resumen, después de haber enviado a la función hash todos los patrones posibles de una cierta longitud, especificada previamente. Por tanto, el número de colisiones será menor cuanto más uniforme sea la distribución de frecuencia de los resúmenes proporcionados por la función hash. Tiene lógica pensar que cuanto más uniforme sea, la tasa de colisiones será menor. Mirándolo desde otro punto de vista, si existen una probabilidad mayor de obtener un cierto grupo de resúmenes a la salida de la función hash, quiere decir que se producirán más colisiones, por lo que conviene uniformidad en la salida.

Para llegar a una decisión fundamentada se ha llevado a cabo un estudio de seis funciones hash distintas:

- RS: algoritmo Robert Sedgwicks.
- XOR: operaciones XOR.
- SAX: operaciones de rotación de bits.
- SDBM: algoritmo usado en el proyecto de tipo Open Source denominado SDBM.
- AM: operaciones de sumas y multiplicaciones.
- AA: operaciones de sumas.

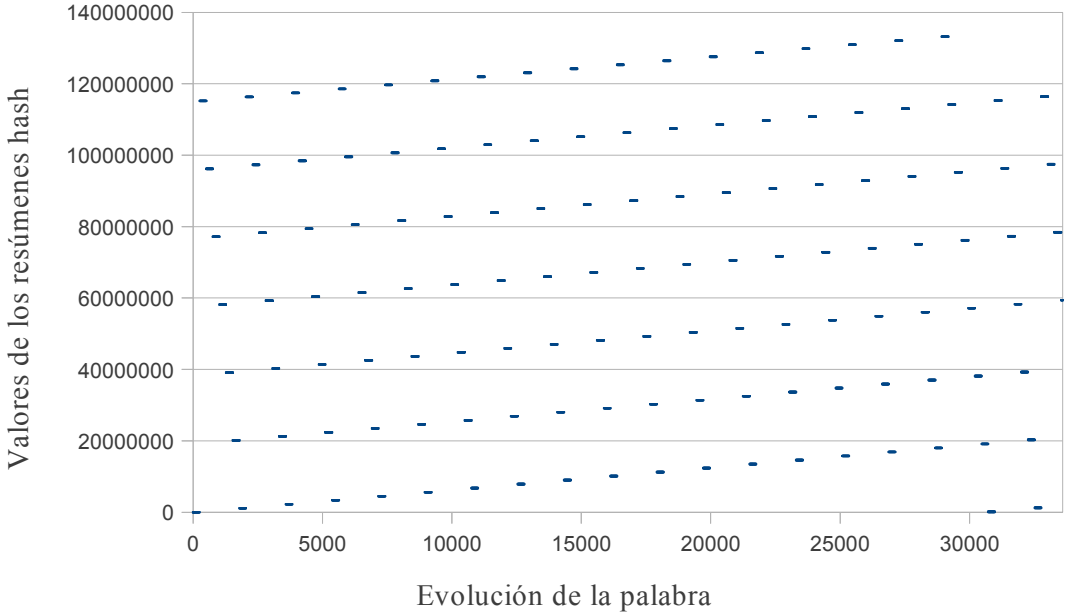
Las dos primeras funciones hash han sido obtenidas del código publicado por los autores de sigMatch, las dos siguientes provienen del sitio web literateprograms.org y las dos últimas son de creación propia. En el cuarto anexo se puede encontrar el código fuente de cada una de ellas.

El estudio llevado a cabo consiste en aplicar cada función hash a todos los patrones posibles de tres caracteres. Teniendo en cuenta que se cuenta con un juego de 256 caracteres, cada función hash se ejecutará 256^3 veces. Se obtienen el tiempo de ejecución de cada función hash y todos los resúmenes obtenidos de cada posible patrón de tres caracteres. De esta última tabla de valores se extraerá una gráfica en la que se representarán los valores obtenidos del resumen correspondientes a cada patrón de los 256^3 posibles.

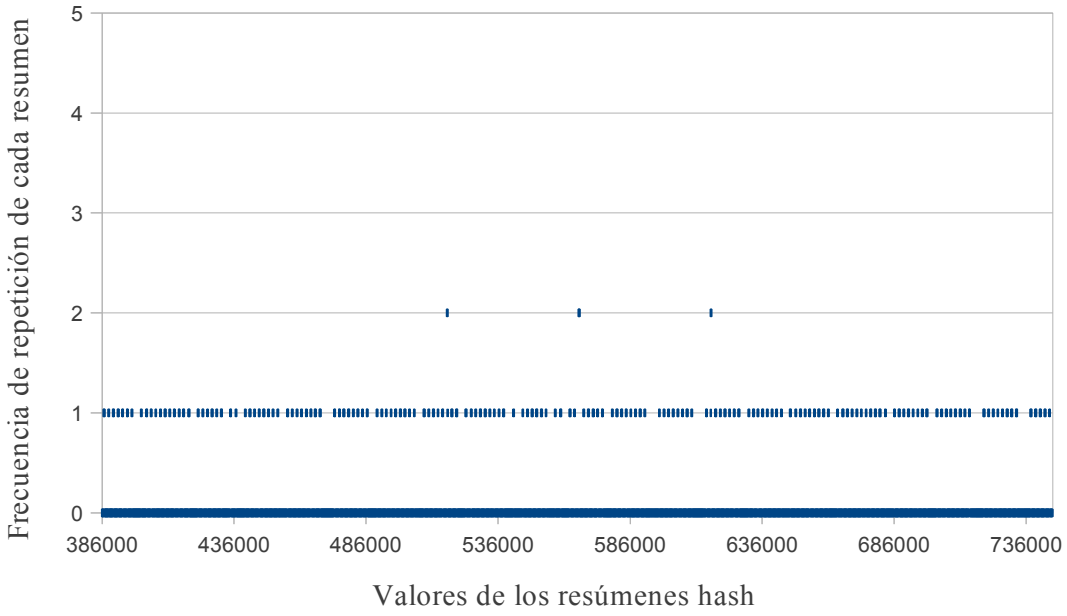
La tabla de resúmenes almacenados se usará también para obtener la frecuencia de cada resumen. Esta nueva tabla de frecuencias se mostrará en una gráfica, en cuyo eje de abscisas aparecerán los valores de los resúmenes y en el de ordenadas la frecuencia de repetición de cada resumen.

A continuación, se mostrarán las gráficas de los valores obtenidos de cada hash y sus distribuciones de frecuencia correspondientes. En algunos casos, debido a la gran cantidad de datos y a la periodicidad de determinadas gráficas, se mostrarán intervalos representativos en lugar del rango completo. Notar que el rango completo de las gráficas correspondientes a los valores de los resúmenes hash debería ir de 0 a 256^3 (16.777.216), mientras que el rango completo de las gráficas relativas a las frecuencias de cada resumen irá del valor mínimo al máximo de cada resumen obtenidos en cada una de las funciones hash.

A la hora de estudiar el comportamiento de una función hash, en cuanto a la tasa de colisiones esperada, únicamente serán importantes las gráficas relativas a las distribuciones de frecuencia de los resúmenes. Las gráficas correspondientes a los valores de los resúmenes se muestran únicamente con una finalidad informativa y para mostrar al lector de qué datos se parten para



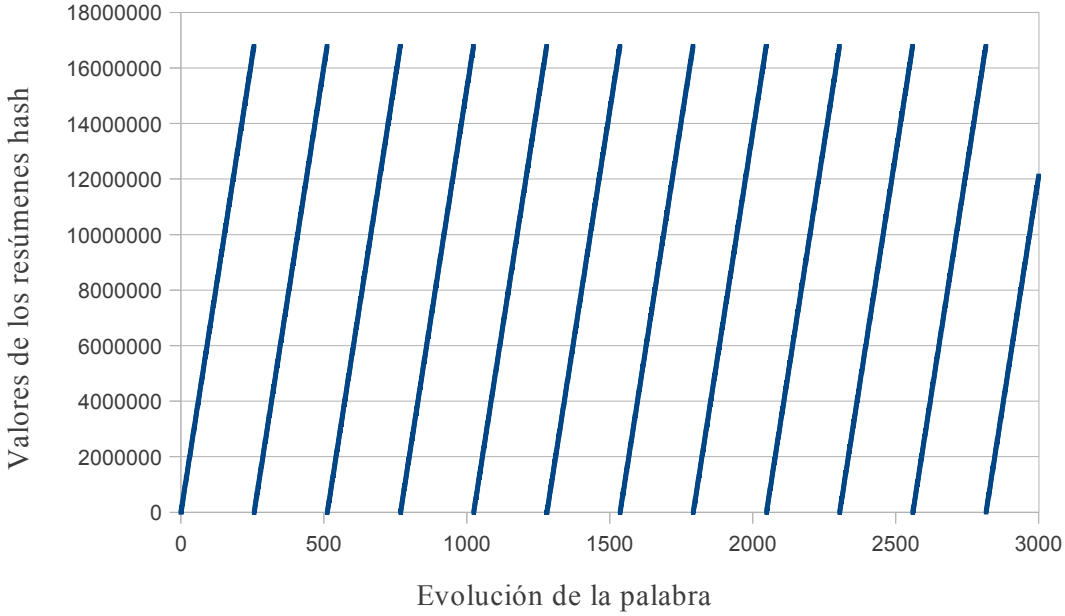
Gráfica 6.2: Valores de los resúmenes correspondientes a la función RS.



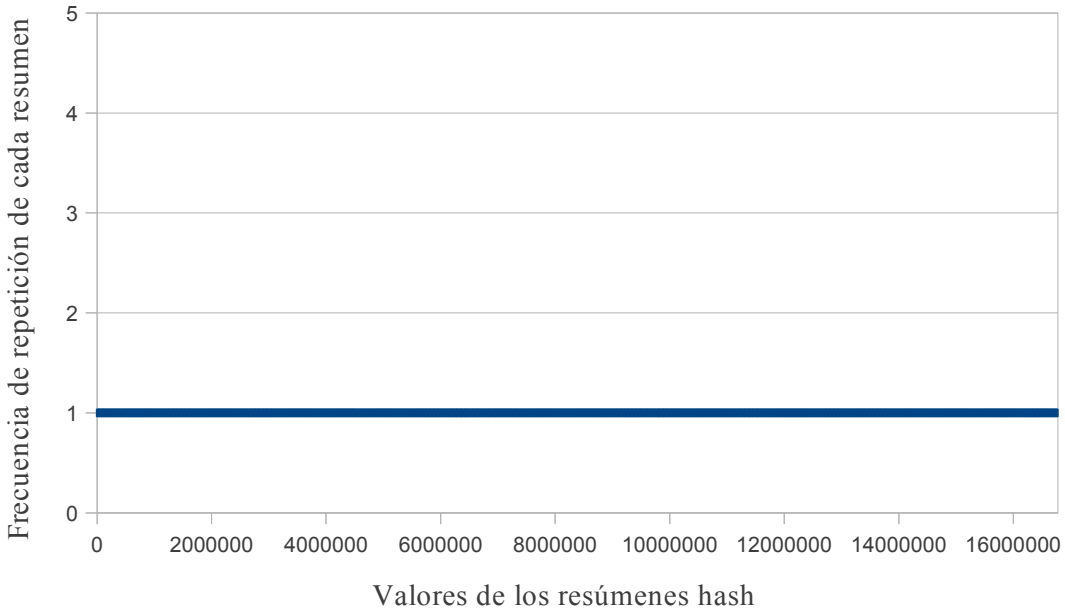
Gráfica 6.3: Distribución de frecuencia de los resúmenes correspondientes a la función RS.

obtener las frecuencias de los resúmenes.

Las primeras gráficas que se muestran serán las correspondientes a la función RS. En la gráfica 6.2 se muestran los valores de los resúmenes obtenidos, mientras que en la gráfica 6.3 se representa su distribución de frecuencia.

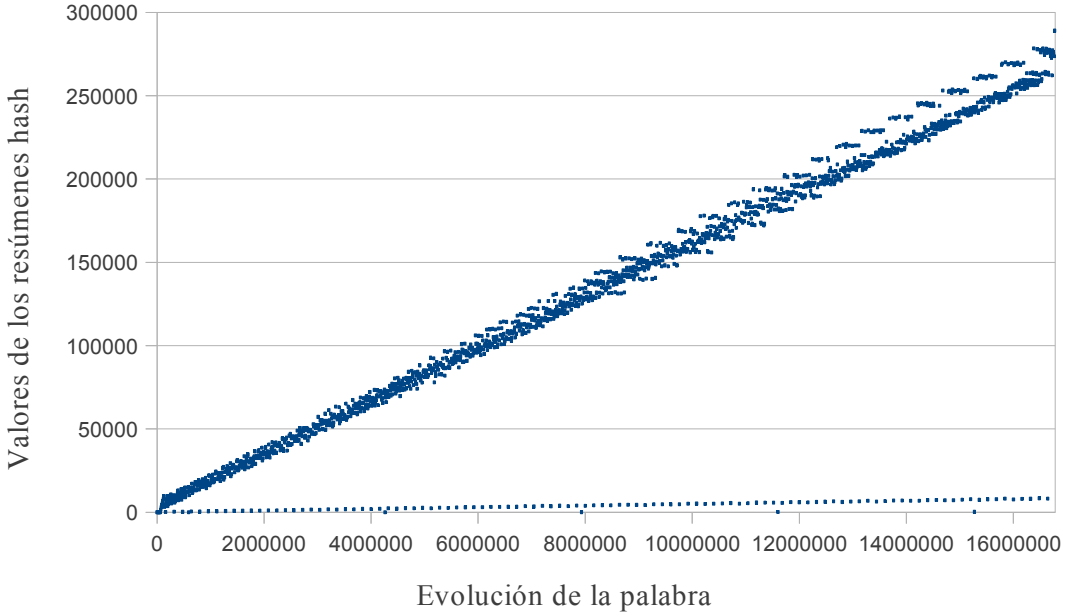


Gráfica 6.4: Valores de los resúmenes correspondientes a la función XOR.

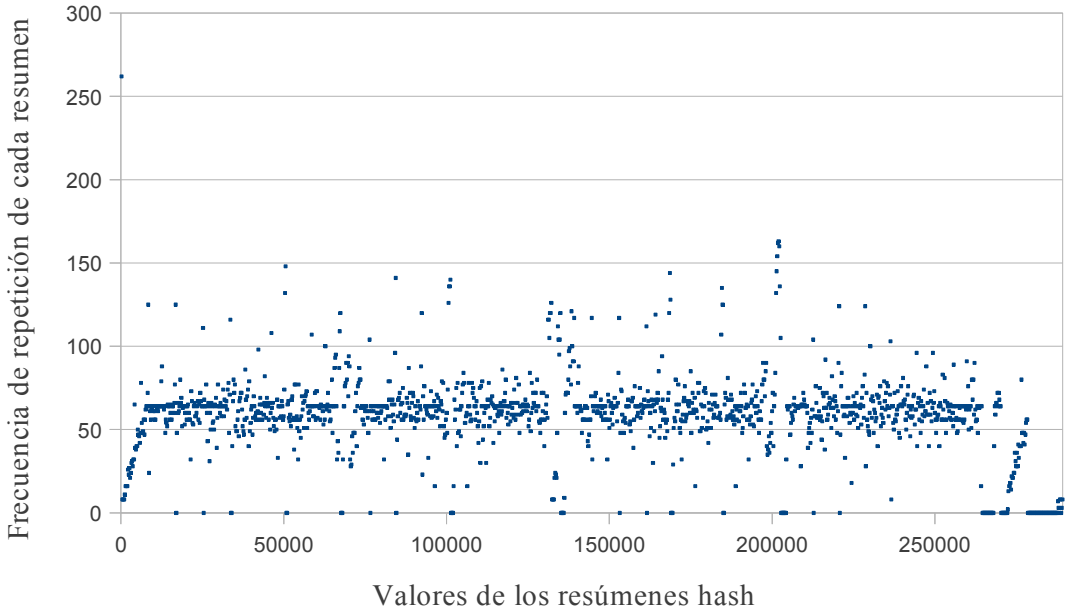


Gráfica 6.5: Distribución de frecuencia de los resúmenes correspondientes a la función XOR.

En la gráfica 6.3 se muestra un rango de datos extraído de un conjunto mayor que va de 0 a 134.216.261. En todo el conjunto se reproduce, prácticamente de manera uniforme, lo que se muestra en este rango menor. Se puede deducir que la mayoría de los resúmenes hash que se presentan tienen frecuencia 1, salvo en ciertos casos en los que la frecuencia es 2. Se observa también, que la amplitud del rango de resúmenes es mayor que la amplitud de los posibles patrones



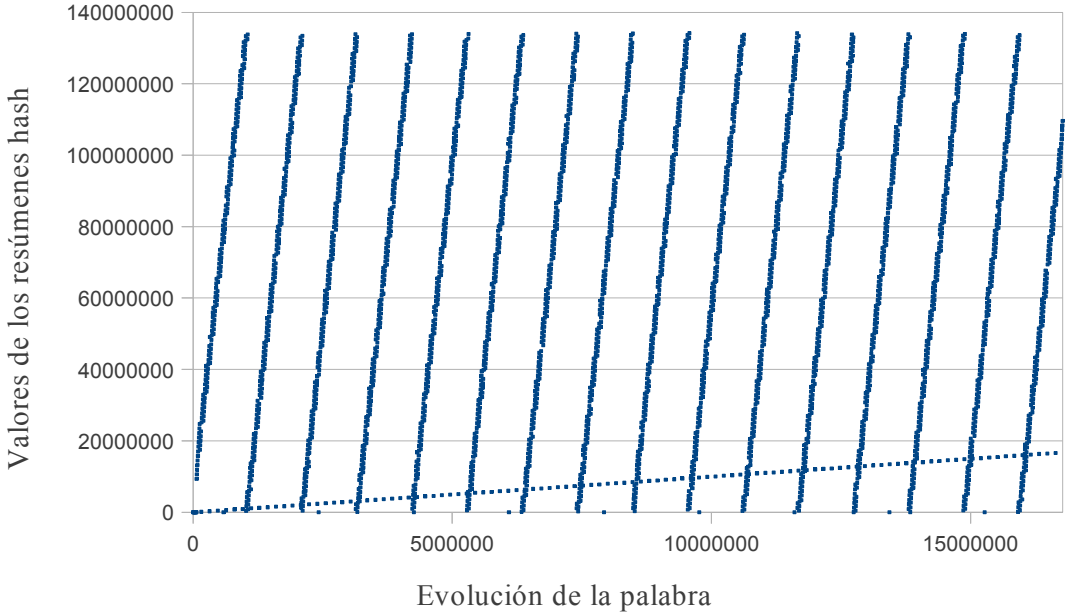
Gráfica 6.6: Valores de los resúmenes correspondientes a la función SAX.



Gráfica 6.7: Distribución de frecuencia de los resúmenes correspondientes a la función SAX.

de 3 caracteres. Eso hace que no se obtengan muchos de los posibles resúmenes, de ahí que aparezca una gran sucesión de puntos situada en la frecuencia 0.

En las siguientes dos gráficas se mostrarán los resultados obtenidos a partir de la función XOR. En la primera de ellas, la gráfica 6.5, se puede ver como la distribución de frecuencia de los resúmenes

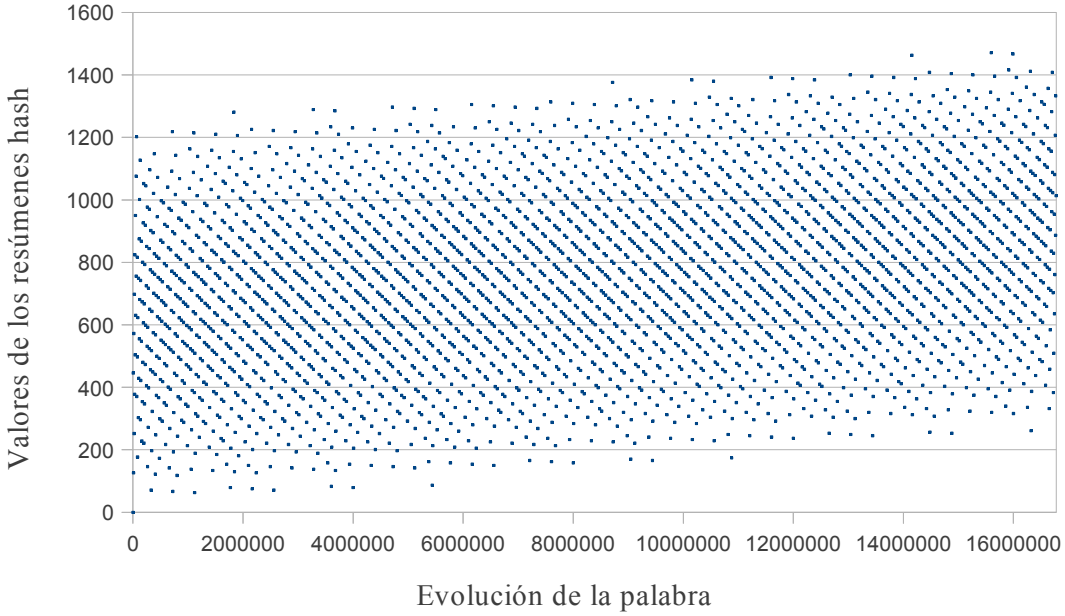


Gráfica 6.8: Valores de los resúmenes correspondientes a la función SDBM.

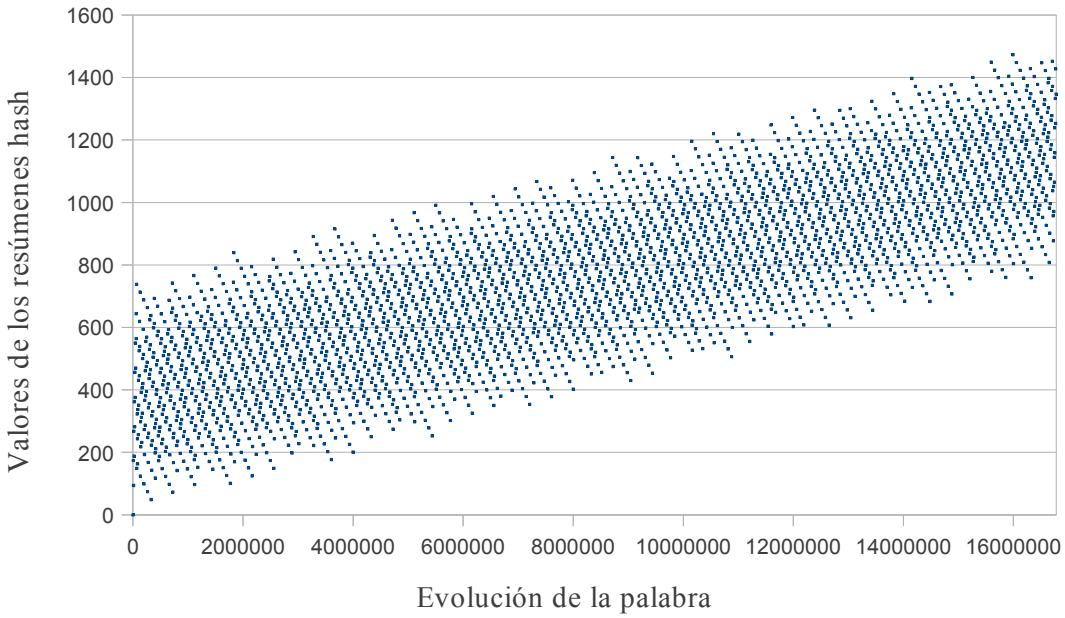
Gráfica 6.9: Distribución de frecuencia de los resúmenes correspondientes a la función SDBM.

obtenidos de la función XOR es uniforme en todo su rango. Este caso se corresponde con una situación ideal por lo que, en un principio y sin tener en cuenta el tiempo de ejecución de la función, podría decirse que esta función es una firme candidata para seleccionarla en el filtrado.

En el caso de la función SAX, se puede comprobar en la gráfica 6.7 como la distribución de



Gráfica 6.10: Valores de los resúmenes correspondientes a la función AM.



Gráfica 6.11: Valores de los resúmenes correspondientes a la función AA.

frecuencia de los resúmenes no es tan uniforme como la distribución de la función XOR, aún así se podría decir que cuenta con una buena uniformidad que va en torno a un valor de frecuencia cercano a 60.

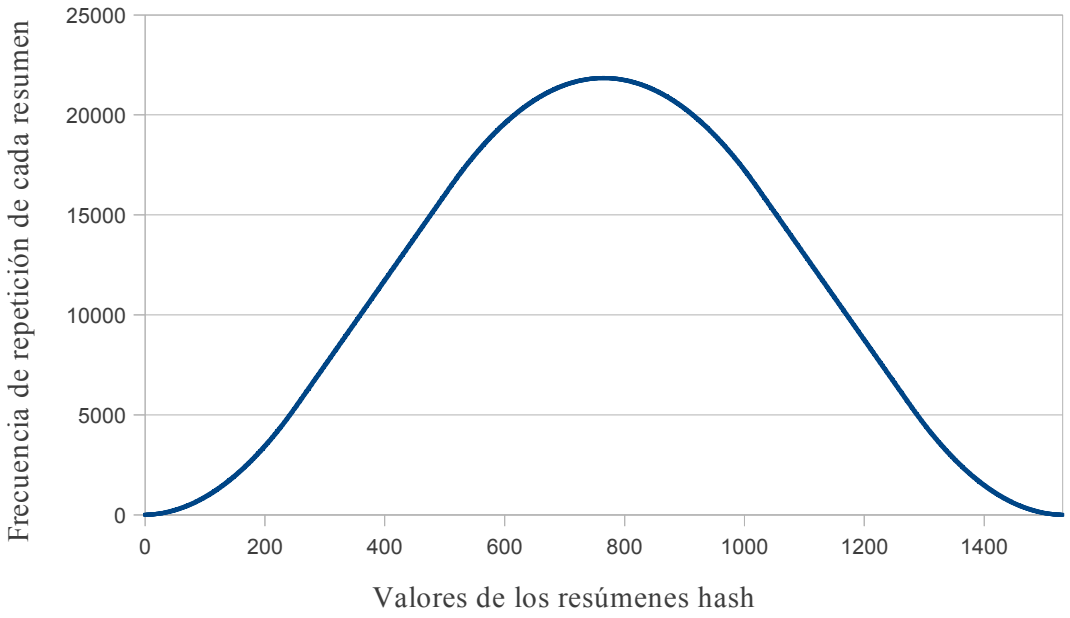
En esta distribución de frecuencia se ha podido mostrar todo el rango de resúmenes, sin haber tenido que descartar un gran conjunto de datos para su representación. Esto se ha podido hacer puesto que el rango de salida de la función SAX es mucho menor que las anteriores, no llegando a 300.000. Esta circunstancia provoca que las frecuencias sean mayores que en los dos casos anteriores.

La próxima función hash a estudiar será la SDBM. La distribución de frecuencia de dicha función, representada en la gráfica 6.9, muestra unos resultados que, en principio, parecen ser poco uniformes. En este caso, la amplitud del rango de resúmenes es bastante mayor, por lo que, al igual que en la función RS, la mayoría de los valores se sitúan en el valor de frecuencia 0.

A continuación, en las gráficas 6.10 y 6.11 se mostrarán las gráficas correspondientes a los valores de los resúmenes obtenidos de las funciones AM y AA, respectivamente, mientras que en la gráfica 6.12 se mostrará la distribución de frecuencia de los resúmenes de las funciones AM y AA. Estas funciones, debido a su funcionamiento interno, comparten esta última gráfica.

A través de la gráfica 6.12 se puede comprobar como las funciones AM y AA tienen un comportamiento gaussiano en cuanto a la frecuencia de los valores de los resúmenes obtenidos, por lo que ambas tienen una muy baja uniformidad.

Dicho esto, sólo quedan por comprobar los tiempos de ejecución de cada una de las funciones hash. En la gráfica 6.13 se presentan los valores de dichos tiempos. Estos valores son la media aritmética de los resultados de cinco ejecuciones secuenciales de un mismo programa que calcula los



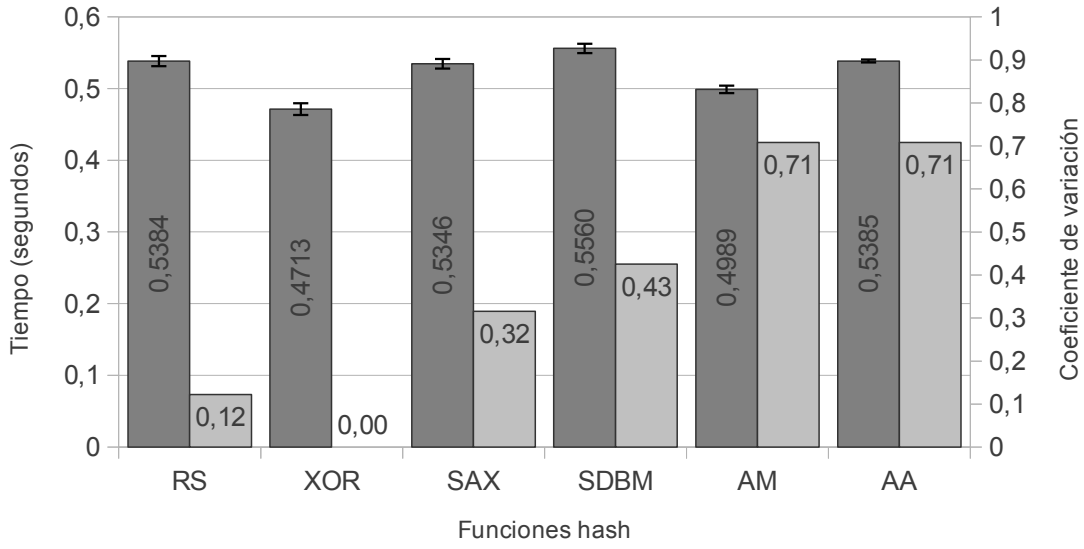
Gráfica 6.12: Distribución de frecuencia de los resúmenes correspondientes a las funciones AM y AA.

resúmenes hash de todos los posibles patrones de longitud 3. También, se muestra el posible error de cada uno de los promedios obtenidos.

En esa misma gráfica, junto a los valores del tiempo de ejecución, se encuentran los coeficientes de variación de las frecuencias de los resúmenes obtenidos de cada función hash. Estos valores están relacionados con la uniformidad que presenta cada función hash en cuanto a la distribución de frecuencia de los resúmenes obtenidos, un valor pequeño indica una alta uniformidad, y viceversa. De esta forma, en la misma gráfica se puede realizar una comparación más directa entre las diferentes funciones hash, al tener en cuenta sus dos características más importantes en cuanto a eficiencia: el tiempo de ejecución y la uniformidad de la frecuencia de los resúmenes obtenidos.

Se puede comprobar como el algoritmo más eficiente, en cuanto a tiempo de ejecución, es el XOR, seguido de cerca por el AM. Más alejados están los algoritmos SAX, RS y AA, y por último se encuentra el algoritmo SDBM, que es el que ofrece peor tiempo de ejecución.

Si se compara la uniformidad de todos ellos, el XOR sigue siendo la mejor opción, seguido del RS, SAX y SDBM, respectivamente. Por último, los algoritmos AM y AA son los que presentan una peor uniformidad.



Gráfica 6.13: Comparativa entre las diferentes funciones hash en cuanto a eficiencia.

A partir de aquí, para elegir un algoritmo u otro, se deberá sopesar la importancia de cada uno de los dos factores. Si, por ejemplo, se da mucha más importancia al tiempo de ejecución que a la uniformidad, en una posible elección que incumbiera a los algoritmos AM y SDBM, se elegiría el primero de ellos. No obstante, hay que tener en cuenta que la uniformidad de la distribución de frecuencia de los resúmenes puede ser alterada en una ejecución normal de Snort.

En un filtro Bloom, los resúmenes proporcionados por las funciones hash se han de dimensionar, con operaciones de módulo, para que indiquen la posición de una matriz –una operación de módulo es una división entre dos números, guardando como resultado el resto de la operación–. Se muestra

un ejemplo de dimensionamiento de datos:

> *Se quiere dimensionar el valor 123.456.789 a un máximo de 8 bits, para ello, se pueden seguir dos caminos. El primero consistiría en realizar una operación de módulo contra 2^8 (256), mientras que el segundo sería realizar una operación AND contra 2^8–1 (255). En ambos casos se obtiene el mismo resultado:*
>
> *123456789 (0x75BCD15) → 21 (0x15)*

Este dimensionamiento del resumen es necesario puesto que el rango de salida de los resúmenes suele ser bastante mayor que el tamaño de la matriz del filtro Bloom. En el ejemplo anterior, suponiendo que el valor 123.456.789 fuera un resumen obtenido de una función hash y que el tamaño de la matriz fuera de 256 posiciones, una vez dimensionado el resumen, éste se correspondería con la posición 21 de la matriz del filtro Bloom.

Sabiendo esto, se puede comprender mejor que la uniformidad de la distribución de frecuencia de los resúmenes se debería calcular en base a los resúmenes dimensionados, para que así, la comparación de una función hash respecto a otra esté más sujeta a la realidad que se producirá al pasar por el filtro Bloom.

En la gráfica 6.14 se muestran los coeficientes de variación de las frecuencias de los resúmenes de cada una de las funciones hash. Se representan tres coeficientes por cada función hash:

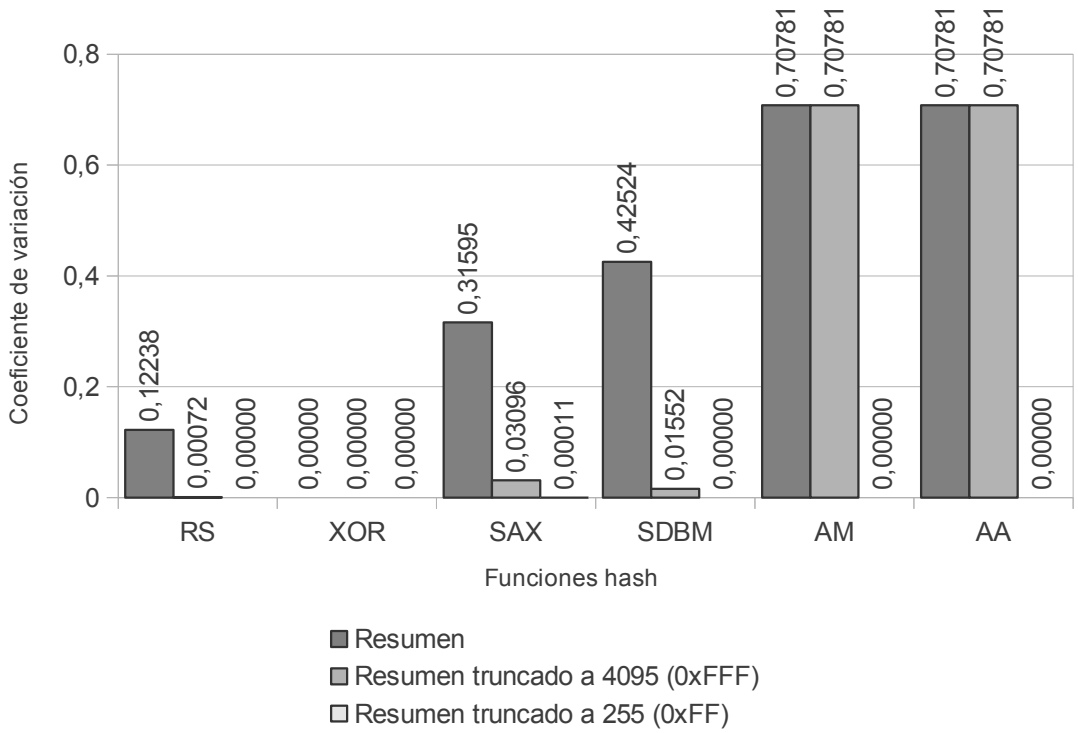
- El primero de ellos, situado a la izquierda, se corresponde con las frecuencias de los resúmenes sin que hayan sido dimensionados, con lo que resultan ser los mismos valores que los mostrados en la gráfica 6.13.
- El segundo, hace referencia a las frecuencias de los resúmenes después de haberlos dimensionado a los 12 bits menos significativos, lo que corresponde con una matriz de filtro Bloom de 4096 posiciones.
- El tercero, más situado a la derecha, corresponde también con las frecuencias de los resúmenes dimensionados, pero esta vez usnado los 8 bits menos significativos, tratándose en este caso de una matriz de filtro Bloom de 256 posiciones.

Si se diera el supuesto en el que, para desarrollar un filtro Bloom, hubiera que elegir entre implementar la función RS o la AM, habría que sopesar ambos factores que conforman la eficiencia. La función AM ofrece un mejor tiempo de ejecución, pero la uniformidad de sus datos es bastante peor. Si esta decisión se tomara sabiendo que todos los resúmenes se deberían dimensionar a un rango de 256, únicamente se tendría en cuenta el tiempo de ejecución, puesto que el coeficiente de variación en ambos casos sería nulo, y por tanto se elegiría la función AM, que presenta un tiempo de ejecución menor.

Seguramente, llegados a este punto, el lector haya observado que cuanto menor sea el tamaño de la matriz del filtro Bloom, menos influirá la distribución de frecuencia de los resúmenes, puesto que tenderá a ser 1 a medida que disminuye el tamaño de la matriz.

En el caso de la implementación del filtrado realizada en Snort, se brinda la opción al usuario de elegir el tamaño del filtro Bloom. Este tamaño puede ser fijo o variable en función del número de patrones que se deban introducir en el filtro. Estas opciones podrán ser configurables en el archivo

`redb.h`.



Gráfica 6.14: Coeficientes de variación de las distintas funciones hash (con y sin dimensionamiento).

Como conclusión, se podrá decir que se usará la función XOR, por ser la que mejor resultado ofrece en cuanto a eficiencia, puesto que es la que ostenta un menor tiempo de ejecución y un coeficiente de variación nulo. También, se realizarán pruebas en las que se usarán dos funciones hash, para así disminuir la tasa de falsos positivos; una de ellas será con seguridad la función XOR, mientras que la otra podrá ser la función RS, SAX o AM, que son las funciones que, a priori, podrían ofrecer un buen rendimiento, por detrás de la función XOR. Estas pruebas se mostrarán al final de este capítulo.

6.2.3.4. Filtro Bloom: tamaño de matriz

Para terminar, se explicará un último apunte acerca de los parámetros configurables del filtro sigMatch, que están a disposición del usuario. Respecto a la eficiencia de los filtros Bloom, ya se ha hablado acerca de las funciones hash, pudiéndose modificar el tipo y el número de funciones hash a utilizar en el proceso de búsqueda llevado a cabo por el filtrado. Esta elección influiría, no sólo en la velocidad de ejecución del filtro Bloom, sino también en su tasa de falsos positivos, o lo que es lo mismo, en su tasa de filtrado.

Otro aspecto que influye en la tasa de filtrado de un filtro Bloom es el tamaño de la matriz que almacena los datos binarios responsables de devolver un resultado positivo o negativo. Es lógico pensar que el tamaño de la matriz será decisivo a la hora de obtener una determinada tasa de falsos

positivos; ante una base de firmas idénticas, la probabilidad de que se produzcan colisiones será mayor en una matriz de menor tamaño. De aquí se puede obtener la relación directa que establece que, a mayor tamaño de matriz, el número de falsos positivos será menor y por tanto se conseguirá una mejor tasa de filtrado.

Llegados a este punto, el lector habrá podido intuir que será necesario establecer una solución de compromiso que relacione el número de colisiones con el tamaño de la matriz del filtro Bloom. Esta relación de compromiso será necesaria puesto que no se puede aumentar el tamaño de la matriz de manera indefinida. Se recuerda que una de las bases en las que se asentaba sigMatch era que el consumo de memoria debería ser suficientemente pequeño como para que su estructura de búsqueda pudiera ser almacenada en memoria caché.

Evaluar una relación de compromiso de este nivel implicaría tener en cuenta muchos factores. Habría que tener en cuenta, entre otros factores, el tipo de tráfico de la red y las propiedades del equipo en el que se ejecutaría Snort. Estos factores son muy variables y el resultado obtenido no se podría extender a cualquier otro ámbito. Por tanto, la mejor opción manera de conseguir una buena relación de compromiso será realizando pruebas con diferentes valores de tamaño de matriz. Estas pruebas se realizarán en el próximo capítulo.

Se plantean dos configuraciones para indicar un tamaño de matriz. La primera de ellas consiste en establecer un tamaño fijo para todos los filtros Blooms que se construyan. Se sabe que el conjunto del número de reglas total se divide en varios grupos; cada subgrupo nuevo no tiene porqué contener el mismo número de reglas que el resto, de hecho es muy posible que la distribución de número de reglas por grupo sea muy diversa.

Debido a las razones anteriores, no parece muy lógico que todos los grupos tengan una matriz de filtro Bloom con el igual tamaño, cuando esta matriz no contará con un número de patrones similar, sino muy dispar. La segunda configuración se basa en este razonamiento. En ella se propone que, en un grupo de reglas concreto, cada matriz de filtro Bloom tenga un tamaño acorde con su número de patrones, estableciendo un factor que, multiplicándolo por el número de patrones, devuelva un tamaño de matriz.

Existe otra configuración alternativa, fruto de la combinación de las dos anteriores. Se trata del mismo principio de la segunda configuración, pero estableciendo un límite de tamaño, por lo que existe un tamaño máximo fijo.

6.2.3.5. Resumen de las opciones

Para terminar, se muestra un resumen de todas las opciones configurables, explicadas anteriormente, a través del fichero `redb.h`. Las líneas resaltadas corresponden a los valores por defecto de cada parámetro.

1. En el primer apartado se especifican los valores de los parámetros *b* y *ß* del filtro sigMatch.
2. En el siguiente apartado se establece el tamaño de la matriz del filtro Bloom. Siguiendo la explicación anterior, la primera variable se corresponde con un factor que establece un tamaño fijo para la matriz. Este factor actuará como la potencia de una base binaria, por lo que un factor 8 devolvería un tamaño de matriz de 256 posiciones –bits–, o lo que es lo

mismo 32 bytes. La segunda variable es una relación de tamaño. Su valor se multiplica por el número de patrones que contendrá el filtro Bloom y el resultado será el número de posiciones de la matriz. Con la conjunción de ambas variables se establece un tamaño variable con cota superior.

3. En el tercer apartado se elige, tanto el tipo, como el número de funciones hash que se deseen utilizar en el filtro.
4. Mientras que el cuarto y último apartado seleccionaría el tipo de implementación que se deseara usar.

```
    /* b and ß parameters */

#define RB_B              2   // bytes
#define RB_BETA           3   // bytes


    /* bloom Filter: matriz size */

//#define RB_BFFACTOR     8   // (2^RB_BFFACTOR)
#define RB_BFRATIO       90   // (number of bloom strings * RB_BFRATIO)


    /* bloom Filter: Hash functions */

#define RB_NUM_HF         1   // How many hash functions will be used
//#define RB_HF_RS            // Robert Sedgwicks Algorithm
#define RB_HF_XOR             // XOR operations
//#define RB_HF_SAX           // Scrolling operations
//#define RB_HF_SDBM          // used in the open source SDBM project
//#define RB_HF_AM            // Addition and Multiplication
//#define RB_HF_AA            // Addition and Addition


    /* implementations */

#define RB_V4                 // Options: RB_V1, RB_V2, RB_V3, RB_V4
```

6.3. Nuevas tecnologías

En el primer capítulo se realizó una introducción acerca de los posibles puntos de mejora de Snort. Uno de ellos trataba la posibilidad de implementar un filtrado previo del tráfico que debía ser analizado por la aplicación –ya analizado en capítulos anteriores–. La otra propuesta tenía que ver con el aprovechamiento, en mayor medida, de las ventajas que ofrecían las nuevas tecnologías.

La evolución que se está produciendo en el campo de los procesadores ofrece nuevas oportunidades, tanto al usuario convencional, como a profesionales cuyo ámbito laboral se centra en el desarrollo de aplicaciones en entornos de alto rendimiento. Son estos últimos los que deberían

explotar al máximo las capacidades que ofrecen los procesadores de última generación.

Uno de los fabricantes pioneros en este campo es Intel. En su afán por conseguir aportar mayor rendimiento a sus procesadores, así como ofrecer soluciones a desarrolladores expertos, desarrolla las librerías IPP (Integrated Performance Primitives), que contienen funciones propias que aprovechan los últimos avances conseguidos en sus procesadores. Una parte de estos avances se corresponde con el desarrollo de instrucciones SIMD (Single Instruction, Multiple Data).

Cada vez que Intel consigue mejorar su tecnología hardware, lanza una nueva gama de procesadores que ofrece nuevas instrucciones SIMD mejoradas. De ahí radica el motivo principal por el que tienen sentido las librerías IPP. Las funciones alojadas en estas librerías consiguen aprovechar al máximo estas instrucciones SIMD y la mayoría de las aplicaciones no se beneficia de este tipo de mejoras.

En 1999, Intel diseña un conjunto de instrucciones SIMD para Pentium III como respuesta al 3DNow! de AMD. Este conjunto de instrucciones es bautizado con el nombre de SSE (Streaming SIMD Extensions). A lo largo de los años, y siempre de la mano de los avances tecnológicos desarrollados en sus procesadores, Intel no ha cesado en la ampliación del conjunto de instrucciones de SSE, dando lugar a las versiones SSE2, SSE3, SSSE3 y SSE4. Tanto fue su éxito que AMD decidió dar su apoyo al conjunto de instrucciones SSE, usándolo en sus procesadores Athlon XP y Duron.

En concreto, la tecnología SSE4 permitía realizar operaciones sobre registros de 128 bits. Algunas de las instrucciones del conjunto de SSE4 que podían trabajar con este tipo de registros estaban ideadas para la búsqueda de bits. Para dar una idea al lector acerca del potencial que este tipo de instrucciones ofrecían, se plantea el caso en el que se quiere realizar una búsqueda de un carácter dentro de un texto de 16 caracteres. Del modo tradicional, debería recorrerse el texto byte a byte y comparar cada uno con el carácter que se desea encontrar. Sin embargo, usando las instrucciones de SSE4, bastaría con colocar el texto en un registro de 128 bits y llevar a cabo una instrucción que compare dicho registro con el carácter deseado, tratando este último como un dato inmediato. Quizás ahora se haya entendido mejor la importancia que ofrecen los conjuntos de instrucciones SSE.

Intel, en su empeño por conseguir acercar estas ventajas a la programación de alto nivel, pone a disposición un gran elenco de funciones que aprovechan al máximo las ventajas del conjunto de instrucciones SSE. Entre todo el abanico de funciones, se encuentra un conjunto relacionado con el manejo de cadenas de caracteres. En él se encuentran funciones como ippsFind, ippsCompare o ippsHash, que en principio podrían resultar muy útiles para la búsqueda de patrones.

El modo de funcionamiento de estas funciones es ideal para la búsqueda de un único patrón, superando posiblemente el rendimiento ofrecido por el algoritmo Boyer-Moore –recuerde el lector el segundo capítulo de este documento–. No obstante, a la hora de realizar una búsqueda de un gran número de patrones, le ocurre lo mismo que al resto de algoritmos de su categoría, que su rendimiento decrece proporcionalmente con el aumento del número de patrones a buscar.

Por supuesto, esta es la gran desventaja de estas funciones, la deficiente integración en un entorno de búsqueda multi-patrón. Está claro que al realizar una búsqueda multi-patrón llevando a cabo el método simple –que no es otro que buscar patrón a patrón– no se conseguiría un mejor rendimiento,

sino todo lo contrario.

Se propuso un método –que podría ser definido como un prototipo de algoritmo de búsqueda multi-patrón– en el que se obtenían resúmenes de firmas y se trataban de tal forma que quedaran únicamente cuatro cadenas de caracteres para buscar. En cada cadena se almacenarían los caracteres, de una determinada posición, de todos los resúmenes obtenidos. Es decir, en la primera cadena se almacenarían todos los primeros caracteres de cada resumen, la segunda cadena contendría todos los segundos caracteres de cada resumen, y así sucesivamente.

La fase de la búsqueda se realizaría comenzando por la primera cadena, mediante el uso de la función ippsFindCAny, que devolvería la primera posición en la que se encontrara uno de los caracteres de la cadena. Continuando por esta posición, se buscaría a través de la misma función en la segunda cadena. De la misma forma, al encontrar la primera posición se evaluarían el resto de caracteres.

En una primera implementación de esta función no se obtuvieron los resultados esperados. Uno de los motivos fue debido a la alta dependencia del tipo de texto. Es decir, la probabilidad de que un texto produjera muchas coincidencias en primeros caracteres era muy alta, pero la mayoría de estas coincidencias no llegaban a resultar una ocurrencia completa. Estas "falsas coincidencias" provocaban que el algoritmo tuviera un bajo rendimiento, por lo que se decidió descartar este camino, retomando el desarrollo de la última implementación de sigMatch.

Aún así, gracias a la experiencia adquirida en esta fase y a un estudio más profundo del perfil de tráfico de red, se adquirió una mayor comprensión acerca del comportamiento de algunos de los algoritmos de búsqueda multi-patrón más empleados. Por lo que se pudieron plantear nuevos caminos hacia el desarrollo de un nuevo algoritmo de búsqueda que consiguiera alcanzar un mejor rendimiento.