7. Filtrado a prueba

7.1. Introducción

Después de haber explicado el desarrollo del filtro previo y de profundizar acerca de los principales parámetros configurables, llega el momento de poner a prueba la implementación. Pero antes de antes de ello, se hará una reflexión previa sobre los resultados previstos.

Para ello, se realiza una valoración de todo lo desarrollado hasta el momento, comparando algunos aspectos de la integración de sigMatch en Snort con el entorno en el que el equipo de sigMatch trabajó. Se recuerda que sigMatch fue integrado en ClamAV, Bro y DBLife. Cada uno de estos sistemas usaba una base de firmas distinta, obteniendo resultados de rendimiento diferentes. Por este motivo, algunos de los puntos que se pretenden comparar son el tamaño de la base de firmas, la longitud de las mismas y el tipo de algoritmo utilizado en la unidad de verificación final. Esto último no se especifica en todos tres sistemas anteriores, por lo que no siempre será posible realizar dicha comparación.

Esta previsión se realiza en base a todos los conocimientos adquiridos en capítulos anteriores. En ella se confrontarán algunos aspectos de la integración de sigMatch en Snort con el entorno en el que el equipo de sigMatch trabajó. Estos aspectos no incluyen únicamente la cantidad de firmas que se emplea en cada análisis –como hizo el equipo de sigMatch–, sino que también se tendrán en cuenta las longitudes de dichas firmas, el tratamiento que realiza Snort de las reglas y las consecuencias que ello provoca, los tamaños de las estructuras de búsqueda tanto de sigMatch como de Aho-Corasick, y los rendimientos teóricos de ambos algoritmos en base al consumo computacional de cada uno.

Todo esto aportará nueva información acerca del funcionamiento y eficiencia de sigMatch; una información que, bajo el punto de vista del autor de este documento, debió ser incluida en el estudio publicado de sigMatch para, de esta forma, mostrar también los puntos débiles de la propuesta de filtrado, y no centrarse únicamente en los puntos fuertes.

Posteriormente, se mostrarán los resultados obtenidos en la integración de sigMatch en Snort, indicando también las características del equipo en el que se realizaron las pruebas, se realizará una reflexión sobre la información obtenida y se llegarán a unas conclusiones finales.

7.2. Previsión

En este punto se retoma la última reflexión que se hizo en el capítulo tres. En dicho capítulo se exponían las expectativas, que sin haber implementado aún el filtro sigMatch ni haber estudiado en profundidad Snort, se tenían de un posible filtrado previo del tráfico en Snort. Estas expectativas auguraban una mejora del rendimiento al implementar sigMatch en Snort.

En las pruebas realizadas por sigMatch se usaron un antivirus, un sistema de extracción de información y un IPS. Dicho IPS se trataba de Bro. En las pruebas llevadas a cabo en Bro, se llegó a alcanzar un factor de mejora de velocidad (*speedup*) de 4,4X. También se sabe que una de las conclusiones postuladas por el equipo de Jignesh M. Patel era que sigMatch mostraba mejores resultados a medida que aumentaba la cantidad de firmas empleadas. Puesto que, para el caso de Bro, se usaron unas 1.200 firmas, se esperaban unos resultados prometedores para Snort, ya que su base de firmas contaba con unas 5.500 reglas aproximadamente.

Todo lo descrito en el párrafo anterior hace alusión a las expectativas que, al final del capítulo tres, se tenían acerca de la integración de sigMatch en Snort. Este optimismo se fundamentaba únicamente en el hecho de que, a mayor número de reglas usadas, la mejora conseguida en una aplicación que llevara a cabo una búsqueda de patrones, y que implementara un filtrado sigMatch, se vería incrementada notablemente, obviando otros muchos aspectos, como podían ser la longitud de dichas firmas, el propio rendimiento de la aplicación por sí misma, o la cantidad de ocurrencias producidas. Ninguno de estos tres aspectos fueron mencionados en la publicación de sigMatch.

En lo que sigue, se abordarán diversos puntos que el autor considera importantes a la hora de medir el éxito de un filtrado de estas características en Snort. Estos puntos son: el número de firmas, junto a sus longitudes; el *splitting* llevado a cabo en Snort y los tamaños de las estructuras de búsqueda de las implementaciones de Aho-Corasick y sigMatch desarrolladas en Snort; y por último el rendimiento teórico de cada algoritmo.

7.2.1. Base de firmas

Como se ha explicado anteriormente, en el estudio de sigMatch intervinieron tres aplicaciones de diferente tipología, cada una de ellas con una base de firmas distinta. ClamAV contaba con un conjunto de 90.000 firmas, DBLife usaba una base de 61.000 firmas, mientras que Bro sólo incluía 1.200 firmas. Esta es la única información acerca de las bases de firmas implicadas en el estudio.

En el tamaño de una estructura de búsqueda, como puede ser el caso de un DFA o un NFA, influye el número de patrones incluidos, pero también es muy importante la longitud de los mismos. Entre dos bases de firmas con igual número de patrones, la longitud media de los patrones será importante, puesto que la estructura creada ocupará un mayor tamaño en memoria, repercutiendo en un peor rendimiento.

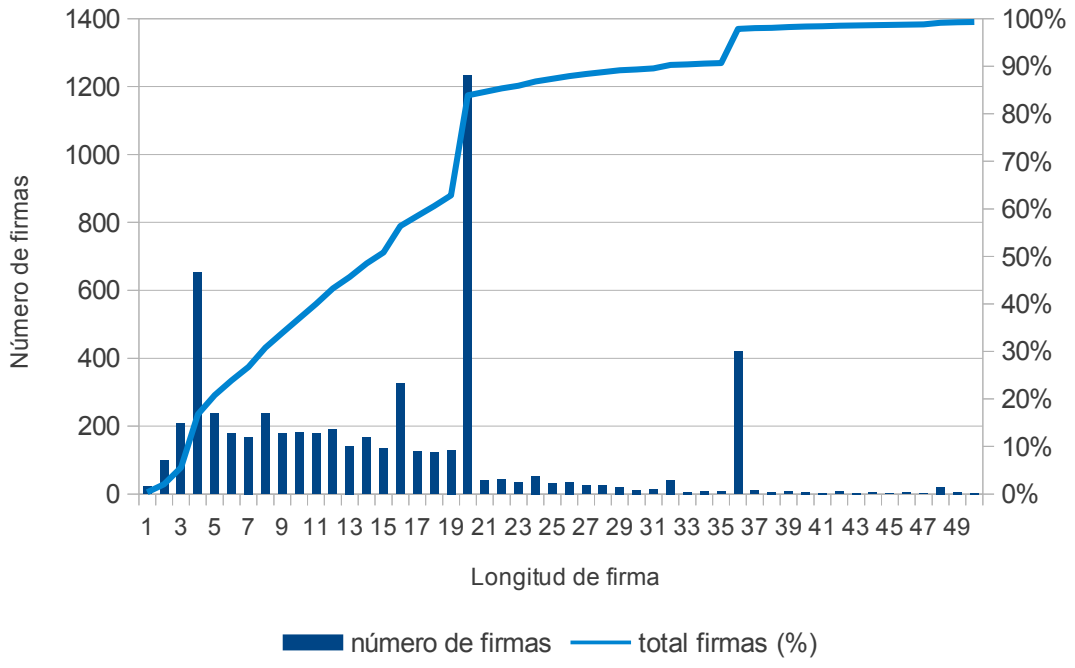
Ya en el capítulo tres se trató esta problemática y se llegó a la conclusión de que *a mayor tiempo de ejecución empleado por un algoritmo –es decir peor rendimiento–, el factor de mejora obtenido al incorporar un filtrado previo será mayor*. Por tanto, en el estudio de sigMatch, se debería haber considerado el hecho de aportar información acerca de la distribución de la longitud de las firmas usada por cada aplicación. De esta forma, se podría saber con más exactitud con qué tipos de bases de firmas proporcionarían una mayor mejora al usar el filtro sigMatch.

Aunque no se hace mención a la longitud concreta de las firmas, en la *release* publicada por el equipo de sigMatch, en la que se proporciona parte de una implementación realizada en ClamAV, se cuenta con una base de 30.000 firmas cuyas longitudes van de 138 a 392 bytes –o caracteres–. Esto hace indicar el tipo de firmas con la que se ha tratado en el estudio, y puede ser indicativo para tener una idea de los resultados que pueden esperarse de la implementación de sigMatch en Snort.

Con bastante frecuencia se publica una nueva versión de la base firmas VRT de Snort, por lo que para este estudio se han usado las firmas de la versión 2.9.1.0. Dichas base de firmas cuenta con 5.551 reglas. Esta cifra aumenta hasta 5.698 cuando se incluyen los conjuntos de reglas (*rulesets*) no activados por defecto. De igual forma, dentro de cada *ruleset* existen reglas que están deshabilitadas; si se activan estas reglas, la cifra aumenta hasta 11.173 para los *rulesets* incluidos por defecto y hasta 19.543 al incluir todos los *rulesets* disponibles.

En adelante, a no ser que se indique lo contrario, cada vez que se haga deba usar información acerca de un conjunto de reglas, bien sea para realizar representaciones gráficas o para llevar a cabo cálculos estadísticos, se elegirá el grupo de 5.551 reglas. Aún así, las mismas conclusiones que se viertan del uso de este grupo de reglas, podrán extenderse al resto de los cuatro grupos definidos anteriormente, ya que todos ellos comparten las principales características que serán de utilidad en este capítulo, como por ejemplo la distribución de longitud de firmas.

De esta forma, para las pruebas de rendimiento se contarán con cuatro grupos de firmas diferentes, formadas por 5.551, 5.698, 11.173 y 19.543 reglas cada uno. En todos estos grupos, más del 80% de los patrones no superan los 20 caracteres de longitud y más del 95% no superan los 36 caracteres. En la gráfica 7.1 se ilustran estos datos, obtenidos únicamente del grupo de 5.551 reglas –el resto de rulesets presentan resultados muy similares–.



Gráfica 7.1: Representación del número de firmas que cuentan con una longitud determinada.

En dicha gráfica, las columnas representan el número de reglas que tienen la longitud que determina el eje de abscisas; por ejemplo, la columna más alta indica que existen más de 1.200 reglas de longitud 20. En cambio, la línea representa el porcentaje total de firmas cuya longitud es menor o igual a la indicada por el eje de abscisas; por ejemplo, en el valor de abscisa 20 la línea supera el 80%, lo que quiere decir que más del 80% de firmas tienen una longitud menor o igual a 20.

Comparando estos valores de longitud con las longitudes de las firmas usadas en la integración de sigMatch en ClamAV, se podría intuir que el factor de mejora que se obtenga de las pruebas realizadas al conjunto formado por sigMatch y Snort será bastante peor que el que se obtuvo al probar sigMatch en ClamAV.

Respecto a Bro no existen datos acerca de la longitud de la longitud de las firmas usadas en las pruebas de rendimiento. No obstante, en el estudio comentan que las firmas de Bro son más cortas que las usadas en la base de firmas de ClamAV. Éste pudo ser uno de los motivos por el que se obtuvo un factor de mejora menor del alcanzado en ClamAV.

No obstante, se conoce que las firmas utilizadas por Bro pertenecían a un grupo de reglas de Snort, por lo que se puede esperar que la distribución de longitud de las firmas de Bro se asemejara bastante a la mostrada en la gráfica anterior.

Si esto fuera así, en principio quedaría descartado el problema de la falta de información acerca de las firmas usadas en el estudio de sigMatch, pudiendo esperar unos resultados como mínimo parecidos a los obtenidos en Bro. No obstante, existen otros aspectos que habrían de tenerse en cuenta y se describirán a continuación.

7.2.2. Tratamiento de las reglas en Snort

Ya se ha hablado en más de una ocasión del *splitting* llevado a cabo por Snort. Esta tarea consiste en dividir el conjunto de reglas en varios grupos. En cada uno de los grupos formados residirán las reglas que compartan la misma información de cabecera y, al mismo tiempo, cada grupo contará con una estructura de búsqueda independiente. De esta forma, existirá más de una estructura de búsqueda que albergará un menor número de patrones del esperado. Es decir, si la base de firmas cuenta con 5.551 reglas, al dividir este conjunto en varios grupos, cada estructura de búsqueda multi-patrón se habrá construido con un número bastante menor.

Esta forma de tratar las reglas que tiene Snort mejora considerablemente el rendimiento de la aplicación. Lo que a su vez provocaría que el factor de mejora que se obtuviera al integrar el filtro sigMatch fuera menor de lo esperado.

Que dicho factor de mejora sea mayor o menor dependerá de cómo se repartan las reglas dentro de cada grupo. Suponiendo que la distribución de reglas por grupo sea uniforme, y que se creen 100 grupos, usando la base de firmas de 5.551 reglas, se obtendrían 55,51 firmas por grupo. Suponer ahora que se introdujera un filtrado previo a un algoritmo de búsqueda multi-patron, y que se usara una base de firmas de 56 patrones. Es evidente que esta cantidad está muy lejos de las 1.200 firmas que se usaron para las pruebas en Bro.

El hecho de que en un algoritmo como Aho-Corasick, basado en un DFA, se cuente con un número

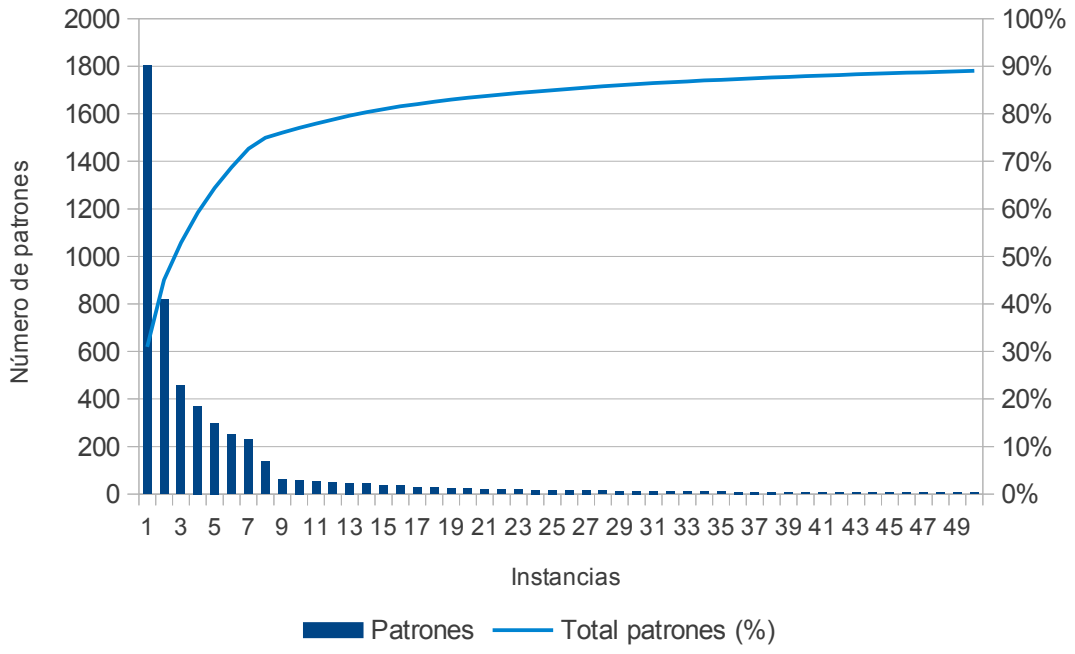
de patrones tan pequeño no debería provocar una merma excesiva del rendimiento ideal –se recuerda que el rendimiento de Aho-Corasick se ve reducido cuantos más patrones se usen para crear su estructura de detección–, lo que quiere decir que el factor de mejora obtenido al integrar el filtro sigMatch podría llegar a ser nulo, o incluso empeorar el rendimiento global.

Para entender esto mejor se pide al lector que repase el capítulo dos dedicado a la algoritmia, en concreto los puntos relacionados con DFA, NFA y filtros Bloom. Ya se sabe que Aho-Corasick usa una estructura basada en DFA, y que sigMatch emplea una combinación de NFA y filtros Bloom. En dicho capítulo se explicó que el rendimiento ideal de un DFA era mayor al de un NFA, puesto que este último sufría la problemática del *backtracking*. También, se dijo que el rendimiento ideal de Aho-Corasick era mayor al del algoritmo Rabin-Karp, basado en filtros Bloom.

En ambos casos, cuando el número de patrones superaba una cierta cantidad, los NFA y los filtros Bloom conseguían un mejor rendimiento que los DFA. Este es el principio por el que se basa sigMatch. No obstante, cuando el número de patrones es lo suficientemente pequeño, sigMatch no sólo no mejoraría el rendimiento de una aplicación basada en Aho-Corasick, sino que lo empeoraría. Este inconveniente es algo de lo que no se hace eco el equipo de sigMatch.

No obstante, se sabe que la distribución de reglas en los grupos no es totalmente uniforme, sino que más bien responde a una forma exponencial negativa; existe un porcentaje reducido de grupos que alberga la mayoría de las reglas, mientras que el resto quedan repartidas, de manera algo más uniforme, en el resto de grupos.

En la gráfica 7.2 se muestra la citada distribución de patrones en las instancias creadas por Snort. Usando el conjunto de 5.551 reglas, se han creado 420 instancias, de las cuáles únicamente se representan las 50 primeras. Se puede comprobar como solo las 14 primeras instancias contienen el



Gráfica 7.2: Distribución de patrones en las instancias creadas por Snort.

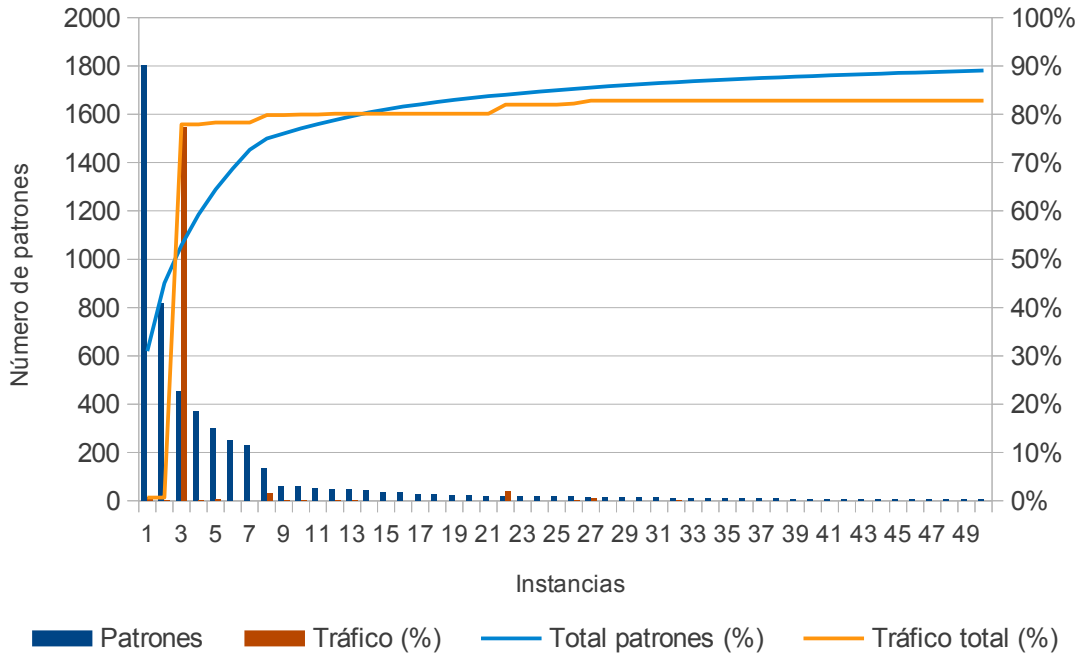
80% de los patrones que están presentes en el conjunto de reglas. Al ordenar los grupos por número de patrones, las instancias más pobladas contienen 1.802, 820, 455, 370, 300, 251, 229, 135, 62 y 59 patrones. Sólo 1 instancia supera los 1.000 patrones, mientras que únicamente 8 instancias, de las 420 existentes, superan los 100 patrones.

Podría ser lógico pensar que, por mera probabilidad, la mayoría del tráfico atravesaría los grupos con mayor número de reglas. Por lo que, a pesar del *splitting*, se conseguiría un rendimiento aceptable, debido al hecho de que el análisis de la red se realice sobre grupos con una cantidad de patrones considerable. No siempre sucederá así.

Como se ha explicado con anterioridad, la agrupación de las reglas depende de la cabecera de cada una. Dicha cabecera indica una dirección en el flujo de datos, por lo que el análisis de determinados grupos dependerá del tipo de tráfico que se produzca en la red y de cómo estén construidas las reglas; con una óptima construcción se consigue que la distribución de patrones sea lo más uniforme posible, de este modo se conseguirá evitar que existan instancias con gran número de firmas, mejorando así el rendimiento de la búsqueda de patrones.

En la gráfica 7.3 se presenta la misma distribución mostrada en la gráfica anterior, pero acompañada por el tránsito de tráfico que soporta cada instancia al analizar una captura de datos[1] concreta. Hay que tener en cuenta que si se hubiera analizado otra captura de datos distinta, la distribución mostrada a continuación podría variar considerablemente.

Se puede comprobar como la mayoría del tráfico (77,26%) pasa a través de la tercera instancia, que cuenta con 455 patrones, mientras que aproximadamente un 20% del tráfico atraviesa instancias con



Gráfica 7.3: Distribución de patrones y del tráfico analizado por cada instancia creada en Snort.

1 *Captura de datos de 1GB realizada por el Information Technology & Operations Center (ITOC) en las dependencias de la National Security Agency (NSA).*

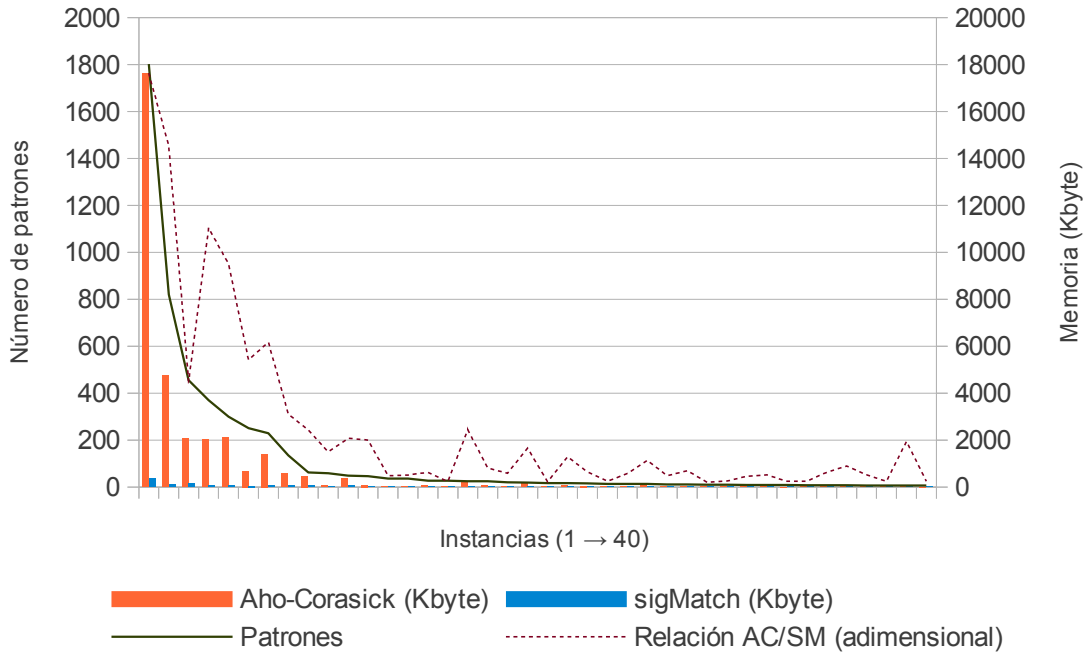
menos de 50 patrones.

En este momento, el lector podrá comprender que la mejora de rendimiento, que podría proporcionar la integración de sigMatch en Snort, difícilmente alcance el factor 4,4X que se consiguió en Bro, puesto que en este caso concreto –se considera un caso concreto el uso de una base de firmas y tipo de tráfico específicos– se espera que el rendimiento de Aho-Corasick por sí solo sea bastante bueno, debido a que sólo un 0,66% del tráfico atraviesa las dos instancias con mayor número de patrones –1.802 y 820–.

Hasta aquí, se ha considerado que el rendimiento real de sigMatch superaría al de Aho-Corasick, cuando se estuviera en un entorno que contara con una base de firmas de un tamaño suficientemente grande. De ahí que se consiguieran grandes mejoras en los estudios de sigMatch. Sin embargo, no se considera tan claro que el rendimiento de Aho-Corasick esté por debajo del de sigMatch cuando se trate de una base de firmas pequeña.

Esta relación, entre rendimiento y cantidad de firmas usadas, se basa en el principio que establece que el rendimiento de un algoritmo de búsqueda de patrones queda condicionado por el tamaño que ocupe en memoria su estructura de búsqueda –cuanto mayor espacio ocupe dicha estructura, menor rendimiento se obtendrá–. De este concepto nació sigMatch: al construir una estructura de búsqueda de menor tamaño, se consigue que sea almacenada en memoria caché, logrando de esta forma una ejecución más rápida.

En la gráfica 7.4 se muestra la distribución del consumo de memoria de las estructuras de búsqueda de Aho-Corasick y sigMatch por cada instancia creada por Snort. Para Aho-Corasick, se ha usado la implementación *Full Matrix*, que es la que ofrece el mejor rendimiento, a pesar de contar con un



Gráfica 7.4: Distribución del consumo memoria de las estructuras de búsqueda de Aho-Corasick y sigMatch por cada instancia creada por Snort.

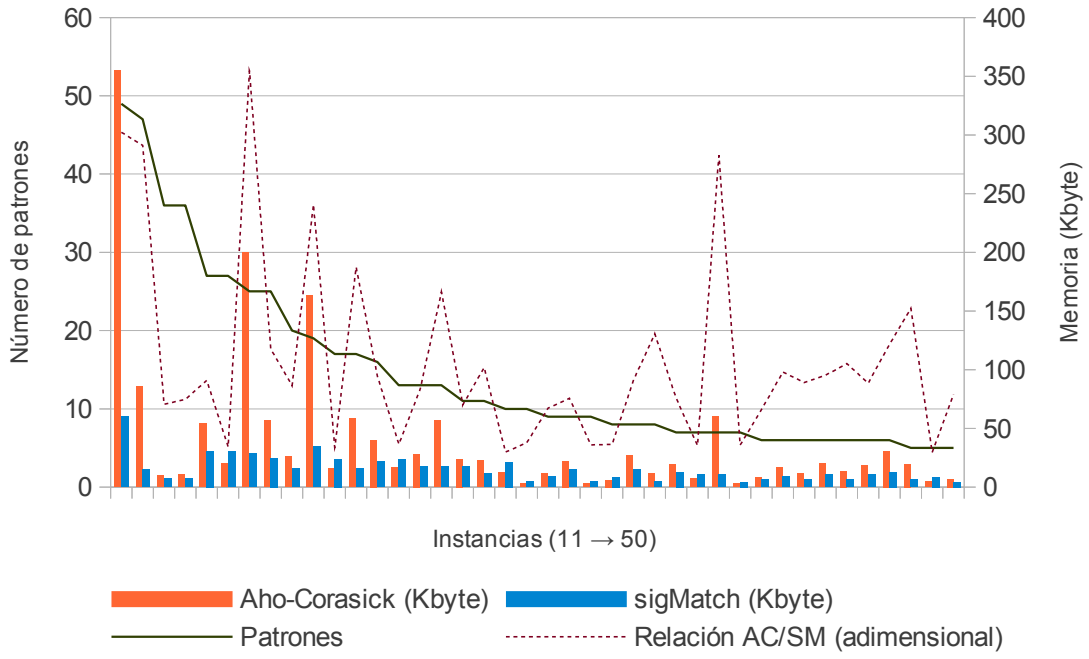
consumo de memoria mayor.

Se muestran las 40 primeras instancias ordenadas por el número de patrones que alberga cada una. La línea continua representa el número patrones de cada instancia, mientras que la línea discontinua representa una relación adimensional entre el tamaño de la estructura de Aho-Corasick (AC) y sigMatch (SM).

Se puede observar como la relación entre el tamaño de cada estructura es mayor cuanto mayor es el número de reglas que contiene una instancia. Esto es así debido a que el tamaño de la estructura de sigMatch no es tan sensible a la cantidad de firmas como sí lo es Aho-Corasick.

La gráfica 7.5 representa los mismos parámetros que la 7.4, salvo por el hecho de que se representan datos a partir de la undécima instancia con mayor número de patrones. En esta gráfica, se aprecia mejor que a medida que una instancia albergue menos patrones, el tamaño de la estructura de búsqueda de ambos algoritmos será más parecido. No obstante, siempre hay casos en los que se producen grandes diferencias de tamaño, de ahí que la línea discontinua ofrezca picos muy a menudo. Esto se debe a que, a pesar de que una instancia contenga pocos patrones, éstos pueden ser de gran longitud, lo que provoca que la estructura de Aho-Corasick crezca bastante, mientras que la estructura de sigMatch permanecería invariable, puesto que no es sensible a longitudes mayores, ya que siempre empleará resúmenes de igual longitud.

Aún con las diferencias que poseen Aho-Corasick y sigMatch en cuanto al consumo de memoria de sus estructuras, se puede comprobar como en ambos algoritmos el tamaño de la estructura de búsqueda es realmente pequeño a partir de la undécima instancia, superando raramente los 100 KB. Esta cantidad de memoria es suficiente para que una estructura de búsqueda pueda ser almacenada



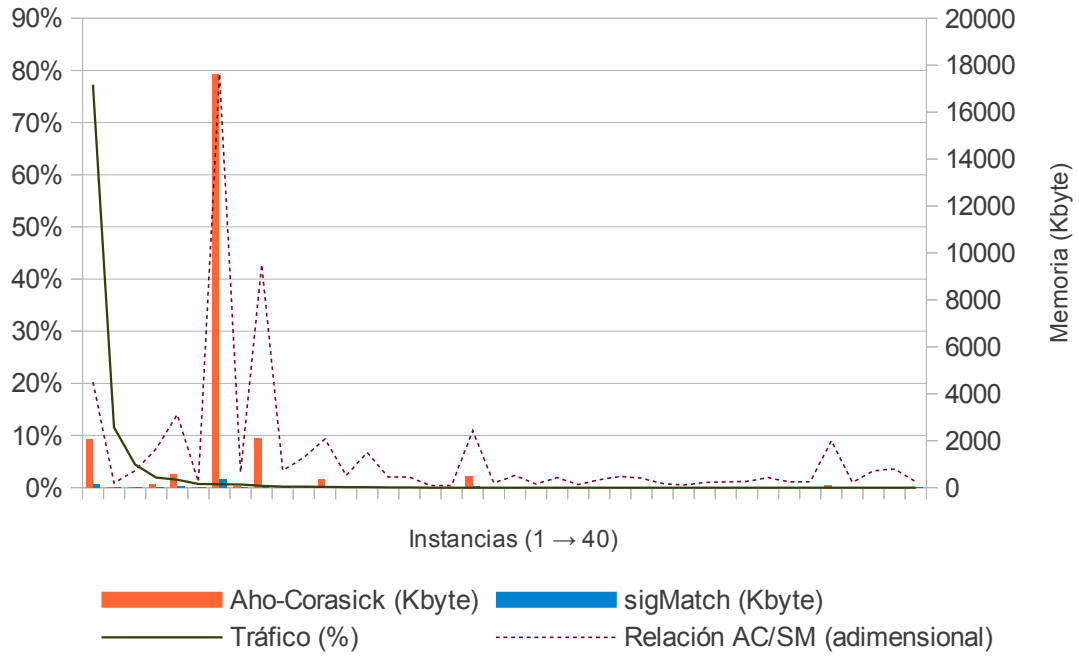
Gráfica 7.5: Distribución del consumo memoria de las estructuras de búsqueda de Aho-Corasick y sigMatch por cada instancia creada por Snort (desde la undécima instancia en adelante).

en memoria caché, por lo que tanto Aho-Corasick como sigMatch podrían funcionar a pleno rendimiento. Este hecho es bastante indicativo si se recuerda que Aho-Corasick cuenta con un rendimiento teórico superior al de sigMatch. Lo que hace pensar que, en estas instancias, no se obtendría ningún factor de mejora. Aunque en realidad, que se consiga o no obtener una mejora dependerá, como se ha explicado anteriormente, de qué grupos intervengan en el análisis de un determinado tráfico.

En la gráfica 7.6 se muestra también la distribución del consumo de memoria de las estructuras de búsqueda de ambos algoritmos, pero en este caso las instancias están ordenadas por la cantidad de tráfico que las atraviesa, habiendo usado la misma captura de datos empleada para la realización de la gráfica 7.3. La línea continua representa en este caso el porcentaje de tráfico que atraviesa dicha instancia. Los demás datos conservan el mismo significado que tenían en las gráficas 7.4 y 7.5.

Los resultados más significativos de esta gráfica son los siguientes:

- El 77,26% del tráfico atraviesa una sola instancia en la que la estructura de Aho-Corasick almacenada apenas supera los 2 MB, mientras que la estructura de sigMatch ocupa exactamente 162,56 KB.
- Algo más del 20% del tráfico atraviesa instancias con estructuras de Aho-Corasick de menos de 600 KB y estructuras de sigMatch inferiores a 70 KB.
- La instancia que alberga las mayores estructuras de Aho-Corasick y sigMatch es recorrida por tan sólo un 0,66% de tráfico. En concreto, la estructura de Aho-Corasick ocupa 17,23 MB, mientras que la estructura de sigMatch ocupa 352,54 KB.



Gráfica 7.6: Distribución del consumo memoria de las estructuras de búsqueda de Aho-Corasick y sigMatch por cada instancia creada por Snort (instancias ordenadas en función de la cantidad de tráfico que soportan).

De tales resultados se puede concluir que casi todo el tráfico atraviesa estructuras de Aho-Corasick de muy pequeño tamaño en memoria. Actualmente, no es raro encontrar procesadores con memorias de caché L2 de hasta 8 MB. Por lo que la cantidad de *cache misses*, que resultaba ser un problema según el estudio de sigMatch, debería reducirse considerablemente gracias al *splitting* empleado por Snort.

En el siguiente apartado se quiere dar un paso más en el estudio de la previsión de la integración de sigMatch en Snort. Después de haber estudiado en profundidad el funcionamiento de sigMatch y de Aho-Corasick, se intuye que la posible mejora aportada por la integración del filtro sigMatch en Snort no dependerá únicamente del tamaño de la base de firmas, ni siquiera del consumo de memoria de las estructuras de búsqueda de ambos algoritmos.

Se cree que jugará un papel aún más importante la correlación existente entre la base de firmas y el tipo de tráfico que se deba analizar; empeorando de esta forma el rendimiento de sigMatch a medida que esta correlación sea mayor. En otras palabras, que la correlación entre el conjunto de reglas y el tipo de tráfico sea muy alta significa que se producirán muchas ocurrencias en el análisis de datos de red.

Parece obvio que ha de ser así. Es decir, cuantas más ocurrencias se produzcan, el rendimiento de cualquier algoritmo de búsqueda de patrones será peor. No obstante, se puede demostrar que el rendimiento de sigMatch dependerá mucho más de este factor de lo que depende cualquier otro algoritmo como Aho-Corasick. En definitiva, se piensa que sigMatch tiene un muy buen rendimiento en entornos de pocas ocurrencias, pero cuando el nivel de ocurrencias es mayor presenta un rendimiento bastante pobre.

7.2.3. Rendimiento: coste computacional

Para reforzar las hipótesis planteadas en el punto anterior, acerca del rendimiento que se podría obtener de Aho-Corasick y sigMatch al situarlos en diversos escenarios, se debe realizar un estudio teórico del rendimiento de ambos algoritmos, en el que se tengan en cuenta los parámetros necesarios para poder ser evaluados en tales circunstancias.

Se deben, por tanto, obtener un conjunto de fórmulas matemáticas, que permitan evaluar diversas situaciones, y que devuelvan un valor indicativo del tiempo empleado en llevar a cabo la ejecución del algoritmo. Para ello, se cuenta con las rutinas de cada algoritmo que fueron implementadas en Snort. En el caso de Aho-Corasick, ya se explicó en el quinto capítulo que las pruebas de rendimiento de la integración del filtrado previo en Snort se realizarían seleccionando la implementación *Full Matrix*, puesto que esta era la modalidad que mejores resultados ofrecía, por lo que se usará la rutina de dicha variante. En el caso de sigMatch, se usará la propia rutina desarrollada para este proyecto y que fue explicada en el capítulo anterior.

Una vez conocida la rutina de programación de un algoritmo de búsqueda de patrones, se podría estimar un cierto rendimiento mediante el cálculo del consumo computacional de la rutina en cuestión (o *dicha rutina*). De esta forma se conseguirían obtener las ecuaciones matemáticas correspondientes que permitieran conocer el rendimiento de dicho algoritmo.

Partiendo del conocimiento de una rutina no es fácil obtener este tiempo de computación teórico. De forma práctica se pueden realizar pruebas de rendimiento en las que se mida el tiempo empleado en completar dicha rutina. Aún así, esto no resulta de mucha ayuda cuando se desea conocer qué partes de la rutina resultan menos eficientes, para así intentar mejorarlas. Ya que, medir el tiempo empleado por cada línea de ejecución introduciría mucho ruido en los resultados finales, puesto que esta medición temporal conlleva un consumo computacional propio que falsearía los tiempos obtenidos. Esta es la razón por la que se pretende desarrollar un método teórico que proporcione una previsión del rendimiento de una rutina cualquiera.

7.2.4. Cálculo del consumo computacional

Las operaciones implicadas en una rutina cualquiera, como comparación de variables, operaciones aritméticas, etc. no están asociadas a un tiempo de computación fijo. No se puede conocer el tiempo que empleará una rutina simplemente conociendo las operaciones que lleva a cabo. Esto es así debido a que este tiempo dependerá de muchos factores, como el tipo de procesador en el que se lleva a cabo, la cantidad y el tipo de memoria disponible, etc.

No obstante, se conoce que en la programación en lenguaje máquina existen instrucciones que realizan operaciones básicas y que emplean determinados ciclos de reloj para llevar a cabo una operación concreta. Este concepto es el primer impulso que se dio para desarrollar la idea del método que se propone para obtener el rendimiento teórico de rutinas de programación.

Se pretende obtener, de cada línea de ejecución de una rutina, el conjunto equivalente de instrucciones de lenguaje ensamblador que llevaría a cabo el procesador. De esta forma, conociendo los ciclos de reloj empleados por cada instrucción básica se podría, no sólo conocer los ciclos de reloj necesarios para ejecutar una rutina, sino también los ciclos que consumiría cada línea de ejecución de dicha rutina para, de esta forma, poder conocer qué partes de la rutina son las que más tiempo computacional emplean.

Casi con total seguridad, se podría asegurar que esta equivalencia no será exacta puesto que seguramente para cada procesador se compile de una forma diversa a la planteada por este método. Puesto que el campo de los compiladores no es el objetivo de estudio de este proyecto, no se profundizará más en este tema y se aceptarán como válidas las equivalencias propuestas, ya que para una previsión de los resultados esperados resulta más que suficiente, dejando para más adelante el estudio de los resultados obtenidos de las pruebas de rendimiento.

Casi todos los procesadores cuentan con un juego de instrucciones similar, en el que comparten las instrucciones más básicas, como pueden ser `MOV`, `ADD` o `INC`. No obstante, el número de ciclos de reloj requerido por cada instrucción cambiará de un procesador a otro. En este caso, se ha elegido como referencia el procesador Pentium 4, modelo 2, de Intel, aunque se procedería de forma idéntica si se eligiera otro procesador.

A continuación se muestran las instrucciones de ensamblador que se usarán para realizar la equivalencia de las rutinas. En la primera tabla se mostrará una descripción de la tarea llevada a cabo por cada instrucción básica, mientras que en la segunda tabla se mostrarán los CPI (ciclos de reloj por instrucción), la latencia y la latencia adicional de cada instrucción. Estos datos han sido

extraídos de la tabla de instrucciones[2] aportada por Agner Fog, en la que se muestra una lista de procesadores de Intel, AMD y VIA con información relacionada con sus instrucciones, latencias y valores de throughputs, entre otros.

Instrucción	*Operandos*	*Descripción*
`MOV`	`r,r`	Mueve el valor del segundo registro al primero.
`MOV`	`r,i`	Mueve el valor de un dato inmediato a un registro.
`MOV`	`r32,m`	Mueve un valor almacenado en memoria a un registro.
`MOV`	`m,r`	Mueve el valor de un registro a una posición de memoria.
`MOV`	`m,i`	Mueve el valor de un dato inmediato a una posición de memoria.
`ADD`	`r,r`	Realiza una suma entre dos registros, salvando el resultado en el primero de ellos.
`ADD`	`r,m`	Realiza una suma entre un registro y un valor almacenado en memoria, salvando el resultado en el registro.
`CMP`	`r,r`	Compara dos registros.
`CMP`	`r,m`	Compara un registro con un valor almacenado en memoria.
`DEC`	`r`	Decrementa el valor de un registro.
`INC`	`m`	Incrementa el valor almacenado en una posición de memoria.
`DIV`	`r32/m32`	Divide un valor almacenado en memoria entre el valor de un registro, salvando el resultado en este último.
`AND`	`r,r`	Realiza una operación AND entre dos registros, salvando el resultado en el primero de ellos.
`XOR`	`r,m`	Realiza una operación XOR entre un registro y un valor almacenado en memoria, salvando el resultado en el registro.

Tabla 7.1: Instrucciones de lenguaje ensamblador implicadas en el método de cálculo teórico de rendimiento: Descripción.

En computación, la latencia es el retraso que provoca una instrucción, siempre y cuando la siguiente instrucción se invoque en la misma unidad de ejecución que la anterior. En el caso en el que la siguiente instrucción dependiente se lleve acabo en una unidad de ejecución distinta, se añadirá una latencia adicional. Ambos valores, latencia y latencia adicional, se expresan en ciclos de reloj, al igual que el valor CPI.

Hay que tener en cuenta que los valores mostrados en la tabla 2 no siempre son exactos, sobre todo

2 http://www.agner.org/optimize/instruction_tables.pdf

Instrucción	*Operandos*	*CPI*	*Latencia*	*Latencia adicional*
`MOV`	`r,r`	0,25	0,5	0,5
`MOV`	`r,i`	0,25	0,5	0,5
`MOV`	`r32,m`	1	2	0
`MOV`	`m,r`	2	1	
`MOV`	`m,i`	2		
`ADD`	`r,r`	0,25	0,5	0,5
`ADD`	`r,m`	1	1	0,5
`CMP`	`r,r`	0,25	0,5	0,5
`CMP`	`r,m`	1	1	0,5
`DEC`	`r`	0,5	0,5	0,5
`INC`	`m`	4	4	
`DIV`	`r32/m32`	23	50	0
`AND`	`r,r`	0,5	0,5	0,5
`XOR`	`r,m`	1	0,5	1

Tabla 7.2: Instrucciones de lenguaje ensamblador implicadas en el método de cálculo teórico de rendimiento: Valores de CPI, Latencia y Latencia Adicional.

los relacionados con la latencia. El consumo de ciclos de reloj podría aumentar si se produjeran *cache misses* o datos mal posicionados en memoria, aunque también es cierto que podrían no darse latencias si las instrucciones se invocaran en un orden concreto para que no hubieran esperas en las unidades de ejecución. Todo esto, también se deja fuera del alcance de este apartado, y se considerará que cada instrucción irá asociada con unos valores mínimos de latencia y latencia adicional.

Una vez explicado todo lo anterior, sólo queda poner en práctica el método propuesto. Se mostrarán las rutinas de la fase *online* de Aho-Corasick y sigMatch. De cada rutina se extraerá una fórmula matemática que represente el número de ciclos de ejecución que requerirá. Esto se conseguirá estudiando cada línea de ejecución y formando un conjunto equivalente de instrucciones de lenguaje ensamblador. Cada línea extraída de una rutina podrá ejecutarse con una cierta probabilidad, debido a los varios caminos de ejecución que podrá ofrecer la rutina –estos caminos suelen estar determinados por sentencias de tipo `if`, `for` o `while`, entre otras–.

7.2.4.1. Aho-Corasick

La rutina de búsqueda de Aho-Corasick será la primera que se estudiará. Este algoritmo se basa en el uso de un autómata finito determinista (DFA). Como se explicó en el segundo capítulo, dedicado

a la algoritmia, los DFAs consisten en un árbol de decisión que transcurre de un estado a otro en base a las entradas que reciba. En el caso de Snort, las entradas serán los caracteres, o bytes, que se reciban del payload del paquete a analizar. Por cada byte recibido, Aho-Corasick deberá cambiar de un estado a otro, y para ello tendrá que decidir, de entre todas las transiciones posibles, cuál es la correcta, en función del byte recibido. Una vez encontrada la transición, ya sabrá cuál será el siguiente estado al que deberá saltar. Este proceso se repite para cada byte entrante.

A continuación, se muestra la rutina básica de búsqueda de Aho-Corasick *Full Matrix* implementada en Snort. Esta rutina, es la tarea que se ejecuta cada vez que se recibe un nuevo byte como entrada del autómata.

```
1  ps = NextState[state];                    INST1
2  sindex = xlatcase[T[i]];                  INST2
3  if (ps[1])                                INST3 → P1
4      if (MatchList[state])                 INST4 × P1
           tareas a realizar una vez
           alcanzado el estado
5  state = ps[2+sindex];                     INST5
```

Junto a cada línea de ejecución se muestra una etiqueta que corresponde con el conjunto equivalente de instrucciones que la representa. Como ya se ha explicado, este conjunto estará formado por instrucciones en lenguaje ensamblador.

No siempre se conocerá el camino que sigue la ejecución de una rutina. En este caso la rutina presenta dos puntos de bifurcación, el primero de ellos se produce en la tercera línea, en la que se comprueba si se ha alcanzado un estado de ocurrencias o estado *match* –al alcanzar un estado de este tipo se sabe que se ha encontrado algún patrón–. El hecho de que se produzca una ocurrencia se asociará a una probabilidad, a la que se ha denominado P_1. Por tanto, la línea cuarta se ejecutará con una probabilidad P_1. En adelante, para el resto de rutinas se procederá de forma similar en cuanto a la inclusión de probabilidades.

Al segundo punto de bifurcación, situado en la cuarta línea, no se le prestará importancia, puesto que, pase lo que pase, a partir de ese punto Aho-Corasick cederá el control de la ejecución a otra tarea que se encargará de verificar si la ocurrencia cumple con una regla y, si es el caso, lanzará la alerta correspondiente. Dicho esto, la fórmula que establece el consumo computacional de la rutina de Aho-Corasick será la siguiente:

$$AC = INST1 + INST2 + INST3 + P_1 \cdot INST4 + INST5 \qquad (1)$$

Seguidamente, se muestran las instrucciones en ensamblador que corresponden a cada conjunto equivalente de la rutina Aho-Corasick:

```
INST1 = 2*MOVr32m + ADDrm + MOVmr
INST2 = 3*MOVr32m + 2*ADDrm + MOVmr
INST3 = MOVri + MOVr32m + ADDrm + CMPrr
INST4 = 2*MOVr32m + ADDrm + CMPrr
INST5 = MOVri + MOVr32m + 2*ADDrm + MOVmr
```

Las suposiciones llevadas a cabo para llegar a estos resultados se muestran en el quinto anexo. En dicho anexo también se mostrarán las equivalencias correspondientes a las rutinas de sigMatch.

Aplicando las equivalencias anteriores, la fórmula 1 queda del siguiente modo:

$$AC = 7 \cdot MOVr32m + 6 \cdot ADDrm + 3 \cdot MOVmr + P_1 \cdot (2 \cdot MOVr32m + ADDrm + CMPrr) + CMPrr \qquad (2)$$

Y dando valores a cada instrucción según la tabla 2, en la que se indica el consumo computacional que requiere la ejecución de cada instrucción, se obtiene:

$$AC = 48{,}75\ CPI + P_1 \cdot 9{,}75\ CPI \qquad (3)$$

Queda por calcular el valor de P_1, que representa la probabilidad de que, al procesar un nuevo byte, se pase de une estado cualquiera a un estado con ocurrencias. Ésta probabilidad dependerá de varios factores, como son la distribución de probabilidad de los bytes recibidos y las características del autómata. Podría pensarse que un autómata con una alta proporción de estados con ocurrencias influiría en una P_1 mayor. No obstante, este supuesto no tiene porqué verificarse siempre, siendo más importante la correlación existente entre esta proporción de estados con ocurrencias y la distribución de probabilidad de los bytes recibidos; aumentando la probabilidad P_1 a medida que la correlación sea mayor. En otras palabras: conociendo la base de firmas y el tipo de datos presentes en la red podría obtenerse esta correlación, y por tanto calcular una P_1 que se ajustara a la realidad.

La distribución de los bytes recibidos podría obtenerse después de realizar un estudio estadístico de los datos recibidos en un entorno de red concreto. No obstante, los valores obtenidos no podrían aplicarse a otros entornos de red de diversas características, por lo que resulta difícil estimar una P_1 fiable que pueda proporcionar un promedio del consumo computacional de la rutina aplicable a cualquier situación.

Esta dificultad para calcular P_1 se puede extender a cada probabilidad presente en el resto de las rutinas que se estudiarán, puesto que todas ellas dependerán de los bytes provenientes de la red y de las características de la base de firmas. No obstante, al mostrar los resultados finales se retomará el estudio del valor promedio de cada rutina, calculando todas las probabilidades presentes en cada rutina, en función de cada entorno de red que se evalúe y del conjunto de firmas. De momento, únicamente se calcularán los valores máximos y mínimos del consumo computacional de cada rutina, como se muestran a continuación para la rutina de Aho-Corasick:

$$AC_{min} = 48{,}75\ CPI \qquad (4)$$
$$AC_{max} = 56\ CPI \qquad (5)$$

En el primer caso, se ha considerado que P_1 sería nula, mientras que en el caso del consumo máximo se toma un valor unitario para P_1, y por tanto siempre se ejecutará la cuarta línea de la rutina.

7.2.4.2. sigMatch

En cuanto a la estructura de búsqueda de sigMatch, denominada sigTree, ya se ha explicado que comienza con un NFA de altura fija *b*, relativamente pequeña, seguida de filtros Bloom y listas enlazadas de patrones cortos.

Se recuerda, del capítulo dos, que en los NFA se producía el fenómeno *backtracking*, por el cuál se debía retroceder en la lectura de los bytes para no perder ninguna posible coincidencia. En el caso de sigMatch, significará que por cada byte recibido, deberán comprobarse como máximo *b* bytes para llegar a un nodo final del NFA, mientras que como mínimo tan solo será necesario comprobar un byte.

Una vez que se haya alcanzado un nodo final, y esto ocurre tras haber realizado *b* comprobaciones, se tienen tres opciones: la primera es marcar el paquete como candidato, que sucederá cuando existan patrones muy cortos, cuya longitud sea menor o igual a *b*; la segunda opción consiste en pasar el control de la ejecución al filtro Bloom, para comprobar la existencia de los *ß* siguientes bytes; la tercera y última opción se producirá cuando existan patrones cortos de longitud mayor que *b* y menor que *ß*, en estos casos se comprueba una lista enlazada que contenga dichos patrones cortos. Todas estas operaciones se pueden llegar a realizar cada vez que se analice un byte.

Las operaciones, arriba explicadas, que pueden producirse con cada llegada de un byte se ilustran en la figura 7.1.

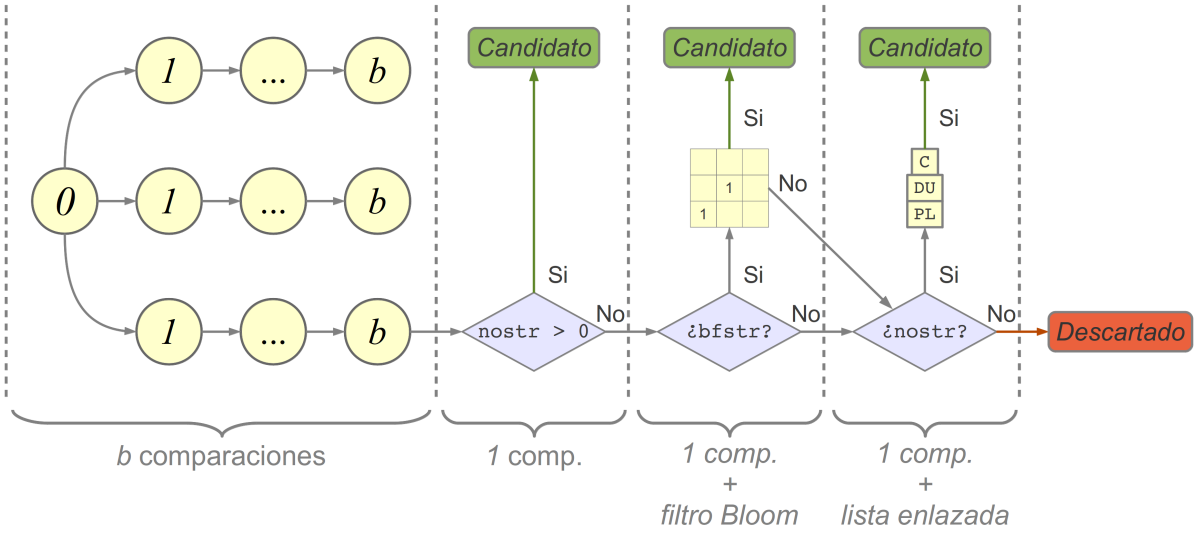


Figura 7.1: Operaciones realizadas en sigMatch.

Para simplificar los cálculos, se tomará un valor *b=2*, siendo además el valor recomendado y el que se tomará por defecto.

A continuación, se muestra la rutina encargada de gestionar todo el proceso anterior:

```
1  if (root->next[xlatcase[T[i]]])                  INST6 → P1
2      leaf = root->next[xlatcase[T[i]]];           INST7
3      if (leaf->next[xlatcase[T[i+1]]])            INST8 → P2
4          qgbf = leaf->next[xlatcase[T[i+1]]];     INST9
5          if (qgbf->numnostr)                      INST10 → Pnostr
              return 1;
6          if (qgbf->numbfstr)                      INST10 → PFB
7              Filtro Bloom                         CFB → PEFB
8          if (qgbf->numshortstr)                   INST10 → PL
9              Lista enlazada                       CL
```

así como la equivalencia de cada etiqueta:

```
INST6 = 5*MOVr32m + 3*ADDrm + ADDrr + CMPrr
INST7 = 4*MOVr32m + 3*ADDrm + ADDrr + MOVmr
INST8 = MOVri + 4*MOVr32m + 4*ADDrm + ADDrr + CMPrr
INST9 = MOVri + 3*MOVr32m + 4*ADDrm + ADDrr + MOVrr
INST10 = 2*MOVr32m + ADDrm + CMPrr
```

Las rutinas relativas a la operación del filtro Bloom (C_{FB}) y de la lista enlazada de patrones cortos (C_L) se estudiarán aparte, para así simplificar los cálculos.

Para extraer la fórmula del consumo de sigMatch partiendo de la rutina previa, se deberán gestionar adecuadamente las probabilidades. Las probabilidades P_1 y P_2 serán decisivas a la hora de finalizar la ejecución de la rutina, esto quiere decir que de no darse una veracidad en cada operación `if`, se terminará con la ejecución de la rutina, por tanto, cada instrucción que se ejecute con posterioridad deberá ser ponderada con las probabilidades P_1 y P_2.

En cuanto a la operación `if(qgbf->numnostr)`, la probabilidad en la decisión que se tome al ejecutarla se corresponderá con P_{nostr}, en caso de no cumplirse, se ejecutarán el resto de operaciones. Por tanto, para continuar con la ejecución del programa se deberán ponderar el resto de instrucciones con la probabilidad $1-P_{nostr}$. Razonando del mismo modo, se ejecutarán las operaciones relativas al filtro Bloom (C_{FB}) con probabilidad P_{FB}, mientras que las operaciones relativas a la lista enlazada (C_L) se llevarán a cabo con probabilidad $(1-P_{FB}\cdot PE_{FB})\cdot P_L$. Esto último se debe a que para que se realice una comprobación en la lista enlazada, debe darse el caso de que pasando por el filtro Bloom, no se produzca un resultado positivo. PE_{FB} corresponde a la probabilidad de éxito del filtro Bloom, es decir la probabilidad de que devuelva un resultado

positivo y por tanto se marque directamente el paquete como candidato, omitiendo de esta forma la comprobación posterior de la existencia de posibles patrones cortos en la lista enlazada.

En la siguiente fórmula se muestra el consumo computacional de la rutina de sigMatch, teniendo en cuenta cada una de las probabilidades mencionadas anteriormente.

$$\begin{aligned} SM = {} & INST6 + \\ & P_1 (INST7 + INST8) + \\ & P_1 P_2 (INST9 + INST10) + \\ & P_1 P_2 (1 - P_{nostr}) (INST10 + P_{FB} C_{FB}) + \\ & P_1 P_2 (1 - P_{nostr})(1 - P_{FB} \cdot PE_{FB}) (INST10 + P_L C_L) \end{aligned} \qquad (6)$$

Y al igual que en el caso de Aho-Corasick, dando valores a cada instrucción, según la tabla 2, la fórmula 6 queda de la siguiente manera:

$$\begin{aligned} SM = {} & 25\ CPI + \\ & P_1 (49{,}5\ CPI) + \\ & P_1 P_2 (34{,}25\ CPI) + \\ & P_1 P_2 (1 - P_{nostr}) (9{,}75\ CPI + P_{FB} C_{FB}) + \\ & P_1 P_2 (1 - P_{nostr})(1 - P_{FB} \cdot PE_{FB}) (9{,}75\ CPI + P_L C_L) \end{aligned} \qquad (7)$$

Al igual que en el caso de Aho-Corasick, el cálculo de las probabilidades se escapa de los objetivos del proyecto, por tanto, se considerarán únicamente los casos máximos y mínimos del consumo computacional de sigMatch. Obviamente, el consumo mínimo se dará para un valor nulo de P_1, mientras que para calcular el consumo máximo se asignará el valor 1 a las probabilidades P_1, P_2, P_{FB} y P_L, mientras que las probabilidades P_{nostr} y PE_{FB} tomarán un valor nulo. Con estas consideraciones, y partiendo de la fórmula 7, los consumos máximos y mínimos de sigMatch serán los siguientes:

$$SM_{min} = 25\ CPI \qquad (8)$$

$$SM_{max} = 128{,}25\ CPI + C_{FB} + C_L \qquad (9)$$

Quedando pendientes por calcular los consumos empleados por el filtro Bloom y la lista enlazada de patrones cortos.

El cálculo del consumo computacional del filtro Bloom se realiza de manera sencilla y sin incluir muchas probabilidades. Esto se debe a que, independientemente del número de patrones que hayan sido añadidos al filtro en la fase *offline*, siempre se ejecutará la misma operación. Sin más dilación, se presenta la rutina correspondiente al filtro Bloom seguida de sus instrucciones equivalentes:

```
1  for (j=0,p=i+RB_B; j<RB_BETA;          INST11+(RB_BETA)*INST12
       j++,p++)
2      bfpayload[j] =                     (RB_BETA)*INST13
           xlatcase[payload[p]];
3  bfpayload[RB_BETA] = '\0';             INST14
4  hashXOR = 0;                           INST15
5  for (bi = 0; bi<RB_BETA; bi++)         INST16+(RB_BETA)*INST17
6      hashXOR ^= *(T+bi);                (RB_BETA)*INST18
7  if (qgbf->bfmatrix &&                  INST19
      (qgbf->bfmatrix[(hashXOR%
      qgbf->bfsize)/SIZE_BYTE]
      & (1 << ((hashXOR%qgbf->bfsize)%SIZE_BYTE))))
       return 1;
```

```
INST11 = MOVmi + 2*MOVri + ADDrm + MOVmr + CMPrm
INST12 = 2*INCm + MOVri + CMPrm
INST13 = 4*MOVr32m + 3*ADDrm + MOVmr
INST14 = MOVri + ADDrm + MOVmi
INST15 = MOVmi
INST16 = MOVmi + MOVri + CMPrm
INST17 = 2*INCm + MOVri + CMPrm
INST18 = 2*MOVr32m + ADDrr + XORrm + MOVmr
INST19 = 4*MOVr32m + MOVri + 2*ADDrm + 3*CMPrr +
         2*DIVr32/m32 + 2*MOVrr + MOVmi + DECr + 2*ANDrr
```

Procediendo de igual forma que en las rutinas anteriores, se obtiene el consumo computacional del filtro Bloom:

$$C_{FB} = INST11 + \beta(INST12 + INST13) + INST14 + \\ = INST15 + INST16 + \beta(ISNT17 + INST18) + INST19 \qquad (10)$$

y asignando los valores indicados en la tabla 2 se obtiene:

$$C_{FB} = 203\ CPI + \beta\ 74{,}75\ CPI \qquad (11)$$

En el caso de las operaciones relacionadas con la lista enlazada se procede de manera idéntica. En primer lugar se muestra la rutina correspondiente junto con la equivalencia de las instrucciones:

```
1  for (k=0;                          INST20 +
       k<qgbf->numshortstr;           (qgbf->numshortstr)*INST21
       k++)
2      flag_detect = 1;               INST22
3  for (j=0,p=i+RB_B;                 INST23 +
       j<qgbf->lenshortstr[k];        (qgbf->lenshortstr[k])*INST24
       j++,p++)
4      if (xlatcase[payload[p]]       INST25
          != qgbf->shortstr[k][j])
5          flag_detect = 0;           INST26
           break;
6  if (flag_detect)                   INST27
       return 1;
```

```
INST20 = 2*MOVr32m + MOVri + ADDrm + CMPrm
INST21 = INCm + 2*MOVr32m + ADDrm + CMPrm
INST22 = MOVmi
INST23 = MOVmi + MOVri + 3*ADDrm + MOVmr + 2*MOVr32m + CMPrm
INST24 = 2*INCm + 2*MOVr32m + 2*ADDrm + CMPrm
INST25 = 6*MOVr32m + 5*ADDrm + CMPrr
INST26 = MOVmi
INST27 = CMPrm
```

Resultando el siguiente consumo computacional:

$$C_L = INST20 + num \cdot (INST21 + INST22 + INST23 + INST26 + INST27) + \\ = num \cdot len \cdot (INST24 + INST25) \qquad (12)$$

Al aplicar los valores de la tabla 2 se obtiene:

$$C_L = 12{,}25\ CPI + num \cdot (47{,}75 + len \cdot 61{,}25) \cdot CPI \qquad (13)$$

Para simplificar, se considerará *num=1* y *len=ß/2*. El primer supuesto tiene lógica puesto que la cantidad de patrones cortos será relativamente pequeña, por lo que no parece muy descabellado suponer que si se debe comprobar la existencia de patrones cortos sea porque normalmente sólo exista uno. En cuanto al segundo supuesto, supondremos el caso intermedio en el que la longitud está entre *1* y *ß-1*. De esta forma, la ecuación 11 quedará como se muestra a continuación:

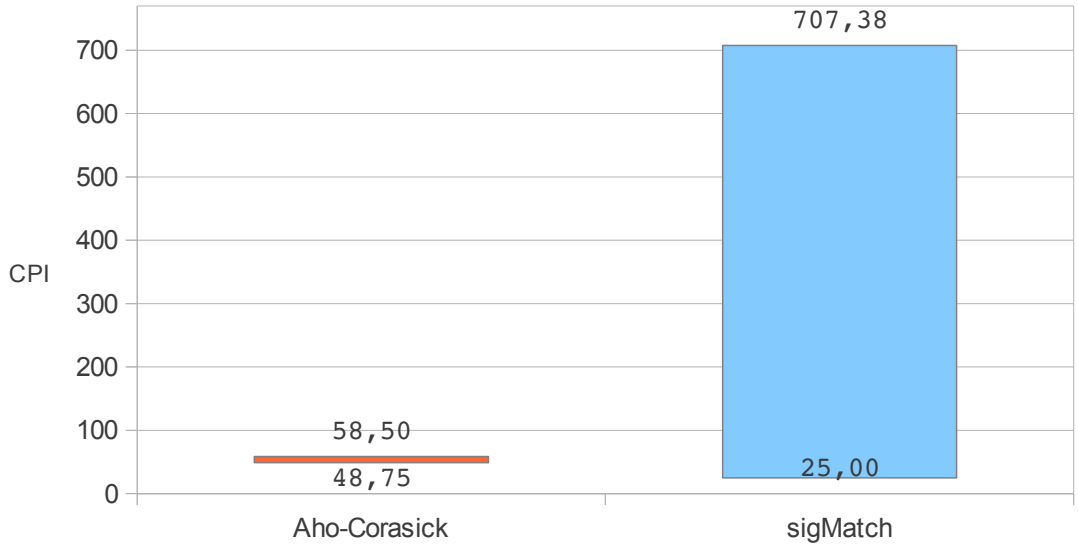
$$C_L = 60\ CPI + ß\ 30{,}625\ CPI \qquad (14)$$

Llegados a este punto, en el que se han conseguido calcular todas las ecuaciones de consumo relacionadas con sigMatch, se podrá obtener el valor máximo definitivo a partir de la ecuación 9, con ayuda de las fórmulas 11 y 14, y asignando a β su valor establecido por defecto ($\beta=3$). Se muestra el resultado a continuación:

$$SM_{max} = 391{,}25\ CPI + \beta\ 105{,}375\ CPI$$
$$= 707{,}375\ CPI \qquad (15)$$

7.2.4.3. Datos finales

Con esta última fórmula obtenida ya se puede realizar una comparación entre los valores máximos y mínimos de consumo computacional de cada algoritmo: Aho-Corasick y sigMatch. Esta comparación se ilustra en la gráfica 7.7:



Gráfica 7.7: Rango de consumo computacional de la implementación de los algoritmos Aho-Corasick y sigMatch: valores máximos y mínimos de cada uno.

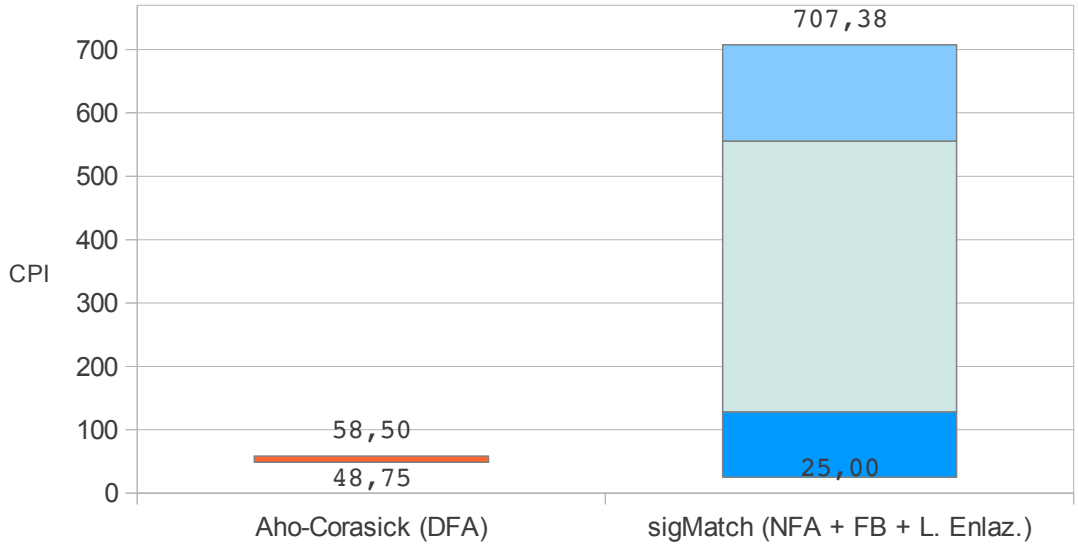
A primera vista, la comparación gráfica hace pensar que el consumo computacional de sigMatc será bastante mayor que el de Aho-Corasick, pero se deberían matizar algunos aspectos importantes para comprender mejor los resultados.

En primer lugar, un dato esperanzador es el hecho de que el consumo mínimo de sigMatch sea menor que el mínimo de Aho-Corasick. Por lo que en una situación extremadamente favorable para sigMatch, el consumo computacional, en comparación con Aho-Corasick, se podría reducir hasta casi la mitad –en concreto, desde 48,75 CPI hasta 25 CPI–. Por el contrario, el consumo máximo que se ha estimado en sigMatch puede llegar a ser hasta 12 veces superior al de Aho-Corasick. Estos resultados se acercan mucho a la hipótesis formulada en el apartado anterior, en el que se conjeturaba con el hecho de que sigMatch podría ser muy buen algoritmo en condiciones

favorables, mientras que en otras condiciones no tan ventajosas, el rendimiento podría verse notablemente reducido.

Otro aspecto a tener en cuenta es la presencia del *backtracking* en sigMatch, provocada por la estructura NFA inicial –se recuerda que la problemática del *backtracking* se presentaba en las estructuras NFA, mientras que en las de tipo DFA no aparecía este problema–. El hecho de que la estructura de Aho-Corasick esté basada en un DFA significa que realizará prácticamente el mismo número de operaciones por byte recibido, por lo que la diferencia entre el consumo máximo y mínimo es muy pequeña, como se ha podido comprobar en la gráfica anterior. En cambio, en las estructuras NFA, por cada byte recibido se realizará, como mínimo una operación, y como máximo todas las necesarias hasta llegar al último nodo del NFA. Esto provoca que el consumo máximo se dispare, aunque la probabilidad de que esto ocurra es bastante pequeña en un entorno de pocas ocurrencias. No acaba ahí, sino que en el caso de sigMatch, al llegar a un nodo final, se deberán hacer comprobaciones "extras", como son las relacionadas con el filtro Bloom y la lista enlazada de patrones cortos.

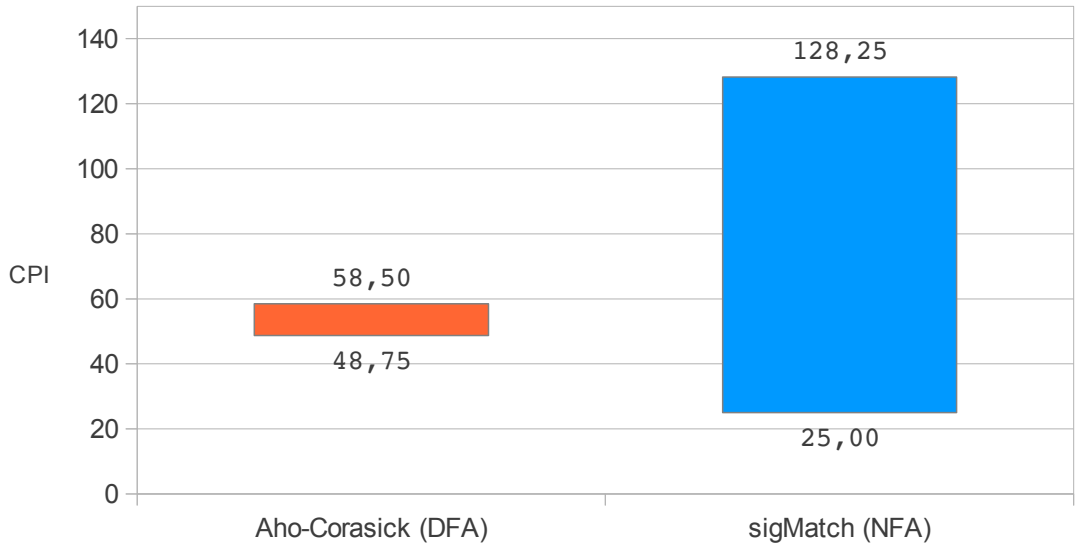
En la gráfica 7.8, se desglosa el consumo computacional de sigMatch en tres partes, representadas de abajo a arriba en el siguiente orden: NFA de altura $b=2$, Filtro Bloom de parámetro $\beta=3$ y lista enlazada de patrones cortos.



Gráfica 7.8: Rango de consumo computacional de la implementación de los algoritmos Aho-Corasick y sigMatch: valores máximos y mínimos de cada uno. Consumo de sigMatch desglosado.

Se puede comprobar que, en un caso de consumo máximo, el filtro Bloom es el componente de sigMatch que más tiempo empleará en su ejecución.

Realizando una comparación directa entre el DFA de Aho-Corasick y el NFA de sigMatch, como se muestra en la gráfica 7.9, se puede observar que la diferencia de consumo computacional entre ambas no es excesivamente grande. Siempre teniendo en cuenta que la probabilidad de alcanzar un consumo máximo en una estructura NFA es muy pequeña.



Gráfica 7.9: Rango de consumo computacional de la implementación de los algoritmos Aho-Corasick y sigMatch (sólo NFA de altura b): valores máximos y mínimos de cada uno.

Sería interesante realizar una estimación del consumo medio de cada algoritmo. De esta forma se podría demostrar todo lo dicho anteriormente acerca de los NFA –que aunque tengan un consumo máximo mayor, en realidad no se alcanza en la mayoría de las ocasiones–.

A pesar de que, a lo largo de este apartado, se ha descartado la posibilidad de realizar estimaciones acerca de las probabilidades involucradas, en este caso se considera interesante llevar a cabo una evaluación de dichas probabilidades. Para ello, se obviarán los posibles entornos de red, usando únicamente las características de una base de firmas VRT proporcionada por Sourcefire.

Esta base de firmas cuenta con 5.551 reglas, que incluyen 5.842 patrones en total. Al compilar estos patrones con sigMatch –siempre para $b=2$ y $\beta=3$– se crean 432 instancias o grupos de reglas, se obtienen 1.717 *q-grams* representativos, o lo que es lo mismo 1.717 nodos finales en el NFA inicial, 1.140 nodos intermedios, 3.650 patrones de filtro Bloom, 496 patrones cortos y 159 patrones muy cortos. Esto indica que, en promedio, por cada nodo final existirán 2,13 patrones normales, 0,29 patrones cortos y 0,09 patrones muy cortos.

El hecho de que los los 5.842 patrones estén divididos en 432 grupos, indica que cada grupo contará, en promedio, con 3,97 nodos finales, 2,64 nodos intermedios, 8,44 patrones de filtro Bloom, 1,15 patrones cortos y 0,37 patrones muy cortos. Hay que aclarar, que la distribución de patrones en los grupos no será uniforme –como se ha podido comprobar en apartados anteriores– por lo que el número de nodos y patrones en cada grupo podrá variar considerablemente. Del mismo modo, todos los datos analizados podrían pasar únicamente a través de un solo grupo, estando este grupo mayor o menor poblado de nodos y patrones. También en este caso, para simplificar, se considerará que todos los grupos estarán igualmente poblados de patrones, y que la compilación de sigMatch proporcionará el mismo número de nodos y patrones.

Con los datos anteriores, se podrán estimar, con mayor o menor acierto, todas las probabilidades involucradas en sigMatch:

- P_1 será la probabilidad de que, al recibir un nuevo byte, se alcance un nodo intermedio. Sabiendo que existen 2,64 nodos intermedios, de 256 posibles, esta probabilidad será de 2,64/256, lo que resulta una $\boldsymbol{P_1}$ **= 1,03%**.
- P_2 será la probabilidad de que, estando en un nodo intermedio, se alcance un nodo final. Por cada nodo intermedio hay 1,51 nodos finales, por lo que la probabilidad de pasar de un nodo intermedio a uno final será de 1,51/256, lo que resulta una $\boldsymbol{P_2}$ **= 0,59%**.
- P_{nostr} es la probabilidad de que al alcanzar un nodo final el paquete se marque como candidato de manera inmediata, debido a la existencia de un patrón muy corto correspondiente con el *q-gram*. Esta probabilidad se calcula como el cociente entre el número de patrones muy cortos y el número de *q-grams*, o nodos finales, resultando $\boldsymbol{P_{nostr}}$ **= 9,26%**.
- P_{FB} es la probabilidad de que, no habiendo sido todavía marcado un paquete como candidato, éste deba pasar por el filtro Bloom para ser analizado debido a la presencia de patrones normales. Habiendo una media de 2,13 patrones normales por cada nodo final, se considerará que al llegar a un nodo final, siempre existirán patrones que analizar. Por lo tanto, se estimará una probabilidad $\boldsymbol{P_{FB}}$ **= 100%**.
- PE_{FB}, sin embargo, es la probabilidad de que, habiendo analizado un paquete en el filtro Bloom, se marque dicho paquete como candidato, es decir que el filtro devuelva un resultado positivo; por tanto, se puede considerar como la probabilidad de éxito del filtro Bloom. Estableciendo un tamaño fijo de matriz de filtro Bloom en 2.048 posiciones (bits), y asumiendo que la función hash tendrá una frecuencia uniforme de salida –lo que significa que cada posición de la matriz podrá darse con igual probabilidad–, la probabilidad buscada se calcularía como cociente entre el número de patrones por *q-gram* (2,13) y el número de posiciones posibles (2.048), resultando $\boldsymbol{PE_{FB}}$ **= 0,10%**.
- P_L, en este caso, será la probabilidad de que, no habiendo sido todavía marcado el paquete como candidato, éste deba comprobarse en la lista enlazada de patrones cortos debido a la existencia de patrones de longitud mayor que *b* y menor que *ß*. Existen 496 patrones cortos y 1.717 nodos finales, por lo que la probabilidad se obtendrá al calcular el cociente entre estos dos valores, resultando $\boldsymbol{P_L}$ **= 28,89%**.

Se insiste en el hecho de que estas probabilidades han sido calculadas obviando la distribución estadística de los bytes recibidos, habiéndola considerado como uniforme. Por lo que se podría asegurar que estas probabilidades serán muy distintas a las que se calcularan al tener en cuenta una distribución estadística más real de los bytes recibidos –es lógico pensar que la probabilidad de recibir un byte correspondiente al carácter asterisco (`*`) no será igual a la de recibir el carácter `H`, sino que debería ser incluso menor–. No obstante, se vuelve a remarcar el concepto de previsión de resultados, por lo que no se emitirá ninguna conclusión final hasta no comprobar los resultados finales obtenidos de las pruebas de rendimiento.

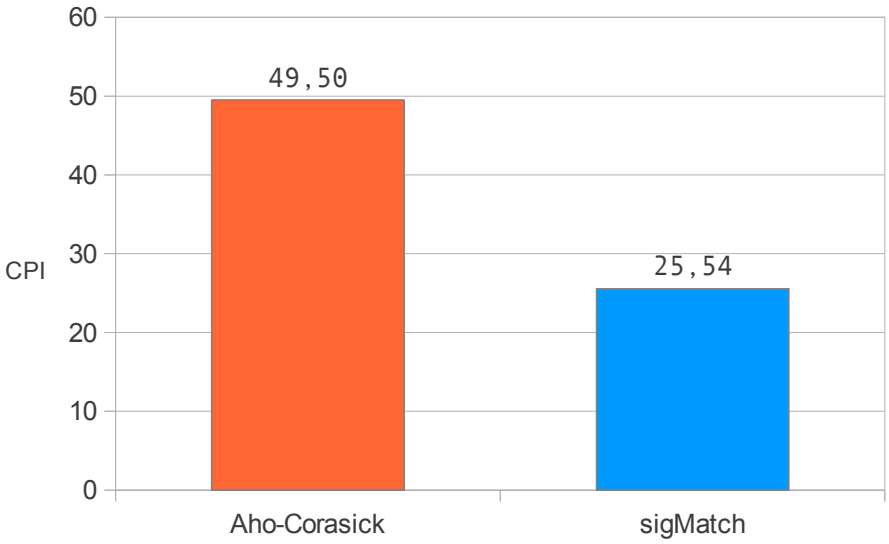
En el caso de Aho-Corasick, únicamente habría que calcular P_1, que indica la probabilidad de que, estando en un estado cualquiera, se avance hacia un estado con ocurrencias. Esta probabilidad se puede calcular fácilmente sabiendo que, al compilar la misma base de firmas descrita anteriormente, se crean 69.499 estados, de los cuales 5.348 serán estados de ocurrencias. Realizando el cociente de estos dos valores, resulta una $\boldsymbol{P_1}$ **= 7,70%**.

Habiendo estimado todas las probabilidades implicadas en las fórmulas del consumo computacional

de Aho-Corasick y sigMatch, se podrían estimar los consumos promedios de las rutinas de ambos algoritmos de búsqueda (AC_{avg} y SM_{avg}). Para ello, se rescatarán las ecuaciones 3 y 7 y se aplicarán los valores estimados de las probabilidades arriba mencionadas. Estos valores se ilustran en la gráfica 7.10.

$$AC_{avg} = 49{,}5\ CPI \qquad (16)$$

$$SM_{avg} = 25{,}54\ CPI \qquad (17)$$



Gráfica 7.10: Consumo computacional promedio de los algoritmos Aho-Corasick y sigMatch, usando probabilidades estimadas, teniendo en cuenta únicamente una base de firmas concreta.

Al parecer, la estimación realizada de las variables da una clara ventaja al algoritmo sigMatch. No obstante, se ha supuesto el caso de una distribución de probabilidad uniforme en cuanto a los bytes recibidos, algo bastante improbable en cualquier entorno de red, por lo que seguramente estos promedios no serán todo lo fiables que debieran.

Como se ha dicho anteriormente, estas probabilidades dependerán mucho de la correlación existente entre los bytes recibidos y los *q-grams* que se obtengan de la base de firmas. Para entenderlo mejor, se planteará un caso práctico más sencillo. Suponga el lector que, a partir de una secuencia de ADN, se desean encontrar varias parejas de bases nitrogenadas –adenina (A), timina (T), citosina (C) y guanina (G)–, en concreto CG, TA, AC y AT. Para encontrar tales parejas en la secuencia de ADN dada, se propone crear una estructura de búsqueda NFA, como la que se muestra en la figura 7.2.a.

Siguiendo el método ya aplicado de estimación de probabilidades basado en el número de nodos (figura 7.2.b), se supone que las cuatro bases aparecen en la secuencia de ADN con igual probabilidad. Por tanto, al tener tres nodos intermedios y un alfabeto de cuatro caracteres, se obtendrá una $P_1 = 75\%$. Procediendo de manera similar al caso de sigMatch, se obtendría una $P_2 = 33{,}33\%$, por lo que la probabilidad total resultaría ser el producto de P_1 y P_2, resultando $P_T = 25\%$.

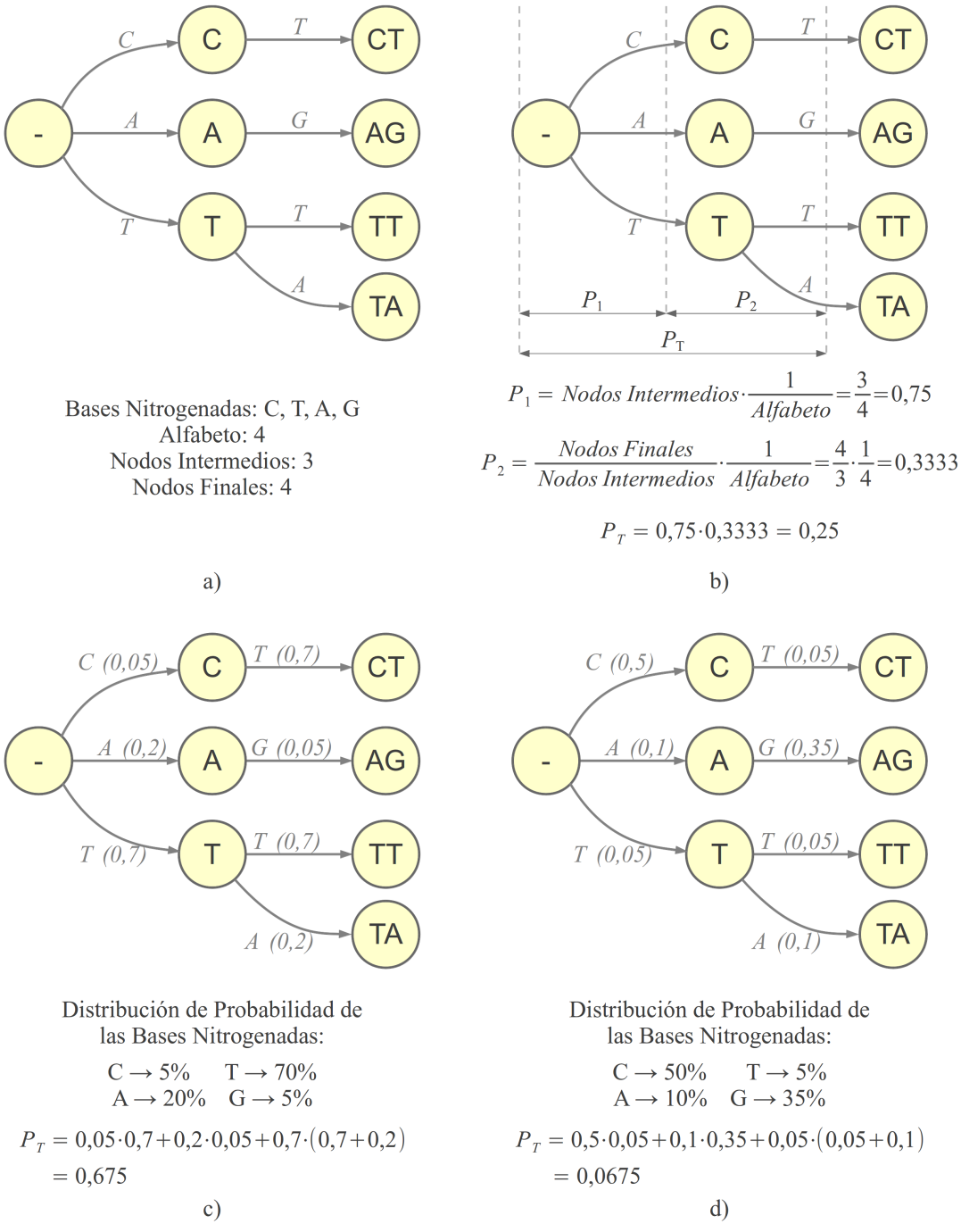


Figura 7.2: Ejemplo de estimación de probabilidades. a) planteamiento del NFA; b) estimación basada en el número de nodos; c) y d) estimaciones basadas en distintas distribuciones de probabilidad de las bases nitrogenadas.

Si se tuviera en cuenta la distribución de probabilidad de las bases nitrogenadas, conociendo el NFA, la probabilidad total se calcularía como se muestra en la figura 7.2.c, es decir sumando las probabilidades de cada rama, que a su vez se obtiene al realizar el producto de las probabilidades de cada transición de nodo. En este caso, se puede comprobar que la probabilidad obtenida resultará mayor (P_T = 67,5%). Normalmente, en un caso real siempre será así. Es decir, si al realizar la estimación de las probabilidades se tienen en cuenta la distribución de probabilidad del alfabeto, la

probabilidad resultante será mayor. Una probabilidad menor correspondería con el caso en que las firmas a buscar estuvieran formadas por caracteres con poca probabilidad de aparición –pero eso no suele suceder en el caso de Snort–.

Si en cambio, la distribución de probabilidad de las bases fuera la de la figura 7.2.d, significaría que la correlación entre el NFA y la probabilidad de aparición de las bases es muy baja, puesto que las bases más buscadas son A y T, mientras que las que más probabilidad de aparición presentan son C y G. Esto provoca una probabilidad resultante bastante menor, en concreto $P_T = 6{,}75\%$.

En casos como el del apartado *d*, sigMatch cosecharía resultados muy buenos, mientras que en el caso *c* el rendimiento se vería enormemente mermado.

Con la ayuda de este ejemplo, se puede afirmar con mayor seguridad que el rendimiento de sigMatch en Snort dependerá en gran medida del tipo de tráfico que deba analizar. Un tráfico que contenga muchas ocurrencias –o lo que es lo mismo, que genere muchas alertas– provocará que se recorra más distancia en la estructura de búsqueda de sigMatch, lo que se traduce en un mayor consumo computacional. Sin embargo, si el tráfico que se debe analizar casi no contiene ataques, no habrá que recorrer una gran distancia en la estructura de búsqueda de sigMatch, por lo que su rendimiento estará cerca del mínimo.

Merece una mención especial el hecho de que en cualquiera de los dos tipos de tráfico, el algoritmo Aho-Corasick no verá apenas alterado su rendimiento en comparación con sigMatch.

Antes de finalizar este apartado convendría aclarar que los consumos computacionales aquí calculados podrían variar debido a varios factores. Uno de ellos se corresponde con la obtención de un mayor número de *cache misses* en la ejecución de un algoritmo que en otro, provocado en parte por la cantidad de memoria usada por cada algoritmo. Pero éste factor no será el único.

Por esta razón, se insiste en el hecho de que todo lo anterior no deja de ser una previsión previa y un estudio realizado de ambos algoritmos de búsqueda de patrones. Será en el siguiente apartado cuando se comprobará realmente la eficiencia del filtrado introducido en Snort.

7.3. Pruebas de rendimiento

Una vez reflexionado acerca de las posibilidades que puede llegar a ofrecer la integración del filtrado sigMatch en Snort, se ha llegado al punto en el que se valorarán los resultados obtenidos al haber aplicado pruebas reales de rendimiento que serán expuestas a lo largo de este apartado.

Las pruebas de rendimiento se dividirán en dos fases. En la primera de ellas se realizarán varias pruebas cuyo objetivo será el de obtener información necesaria que permita tomar una decisión acerca de los valores que se habrán de asignar a los parámetros configurables. Una vez elegidos estos valores se procederá con la ejecución de la segunda fase, en la que se analizará de nuevo la implementación, pero esta vez usando varios entornos de red, así como distintas bases de firmas.

En ambas fases, las pruebas de rendimiento fueron ejecutadas en un equipo con las siguientes características:

- AMD Turion(tm) II Dual-Core Mobile M520 a 2.3GHz y 512KB de memoria caché L2 por núcleo.
- 4GB de memoria RAM.
- Sistema Operativo Linux.

7.3.1. Fase I: Estimación de los parámetros configurables

Como ya se explicó en el capítulo anterior, existen principalmente cuatro opciones configurables por el usuario en el fichero `redb.h`. Estas opciones son el tipo de implementación, los parámetros del filtro, el número y tipo de funciones hash empleadas y el tamaño de la matriz del filtro Bloom.

Para decidir qué parámetros usar en la segunda fase, se realizará una batería de determinadas pruebas que aporten los datos necesarios para poder elegir un valor concreto en cada caso. Tenga en cuenta el lector que, en esta primera fase, los ejes de ordenadas de las gráficas no comenzarán desde el valor 0, por lo que aparecerán algo distorsionados con respecto a la realidad. Esta medida se toma debido a que, en algunos casos, las diferencias son tan pequeñas que no se lograrían distinguir con facilidad.

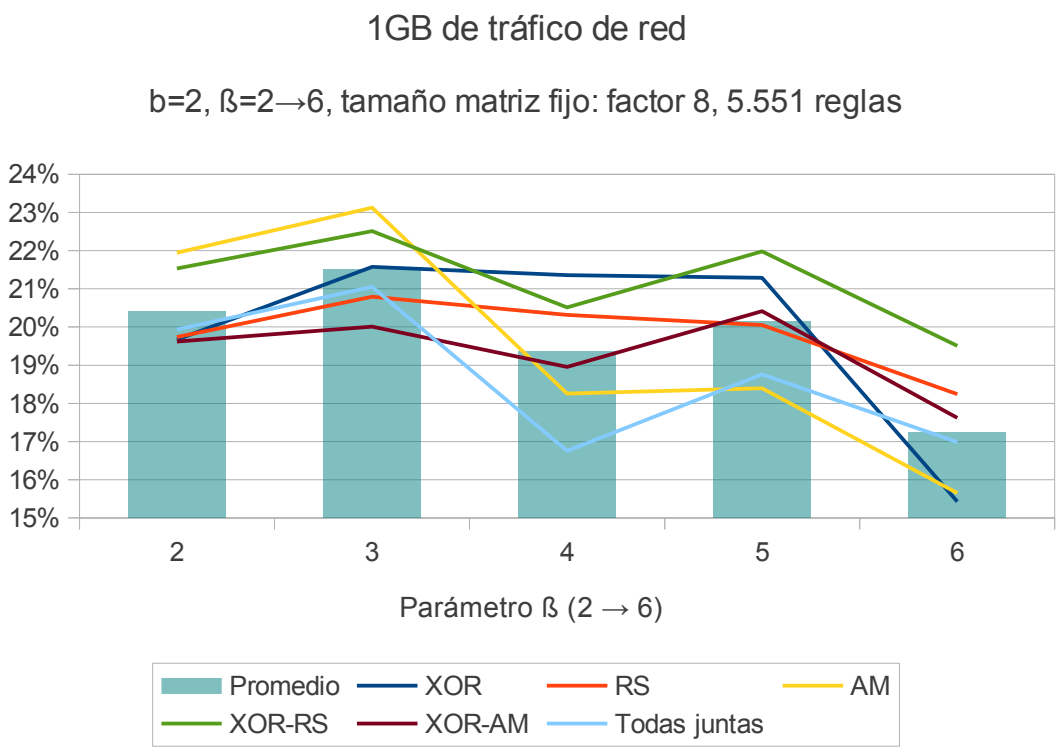
7.3.1.1. Estudio del parámetro *ß*

Para establecer un valor del parámetro *ß* que ofreciera el mejor rendimiento posible se realizaron pruebas asignando a *ß* un rango de valores de 2 a 6, ambos inclusive, estableciendo un tamaño de matriz fijo de factor 8 y usando varios tipos de funciones hash. Para la prueba se utilizó la base de 5.551 firmas, ya mencionada anteriormente para otros propósitos, y un fichero PCAP de 1GB con datos de tráfico de una red, también utilizado en apartados anteriores. El resultado se puede comprobar en la gráfica 7.11.

En la gráfica, se representa el porcentaje de mejora que se obtiene al utilizar el filtrado sigMatch en Snort. La comparación se realiza sobre el tiempo de ejecución, empleado por el motor de búsqueda de patrones, que se obtiene al ejecutar Snort empleando el filtro sigMatch y al ejecutarlo sin filtrado previo.

Para ser más rigurosos, el porcentaje de mejora que aparece, y aparecerá en el resto de estudios, es un promedio de la mejora obtenida al llevar a cabo cuatro ejecuciones idénticas entre sí.

Las líneas de la gráfica representan la evolución del rendimiento al cambiar el parámetro *ß*. Cada línea se corresponde con un tipo de función hash empleado. Se han utilizado las funciones hash XOR, RS y AM por separado, así como una combinación de dos funciones hash al mismo tiempo. También, la etiqueta *Todas juntas* indica que se han utilizado las seis funciones hash de forma contemporánea. Se recuerda que estas seis funciones eran RS, XOR, SAX, SDBM, AM y AA.



Gráfica 7.11: Estudio del parámetro ß.

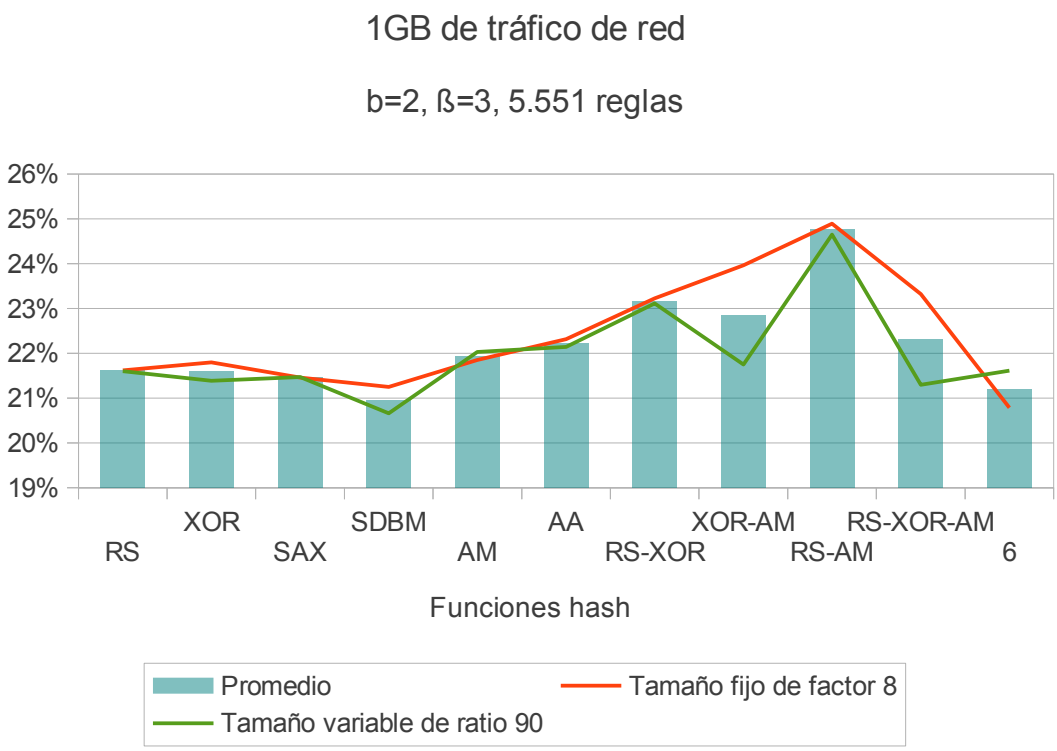
Las columnas, en cambio, representan el promedio de los factores de mejora obtenidos al usar las seis combinaciones anteriores de funciones hash. Dichas columnas proporcionan el dato interesante, que es el que establece *ß=3* como mejor opción.

7.3.1.2. Estudio de las funciones hash

Una vez seleccionado el valor de *ß* se sigue el mismo procedimiento para obtener información acerca del rendimiento de cada función hash. Se realizaron pruebas con un valor de *ß=3*, dos tipos de configuraciones acerca del tamaño de matriz, una de ellas la correspondiente a un tamaño fijo de factor 8 y la otra para un tamaño variable de relación 90. Se utilizó la misma base de firmas y la misma captura de datos que en el apartado anterior. Y se utilizaron las seis funciones hash por separado, así como varias combinaciones de algunas funciones. Se muestran los resultados en la gráfica 7.12.

En este caso, las líneas representan la evolución del factor de mejora al usar distintas combinaciones de funciones hash. Cada línea representa una configuración de la matriz del filtro Bloom distinta. Se han usado estas dos configuraciones para comprobar el grado de influencia de esta configuración en el factor de mejora de cada función hash. Es concreto, interesaba conocer si la función hash con mejor promedio también era la mejor para distintas configuraciones de tamaños de matriz.

La combinación que mejor promedio ofrece es la formada por las funciones RS y AM. La combinación de las seis funciones ofrece un resultado muy pobre, por lo que será descartada de futuras pruebas. Tampoco ofrece un buen resultado la combinación formada por RS, XOR y AM.



Gráfica 7.12: Estudio de las funciones hash.

En el ámbito individual, las funciones AM y AA son las que mejores resultados cosechan. Se comprueba también que no es determinante el uso de una configuración específica de tamaño de matriz, pudiéndose extraer idénticas conclusiones si se hubieran presentado únicamente los resultados correspondientes a una de las dos configuraciones utilizadas para el tamaño de la matriz del filtro Bloom.

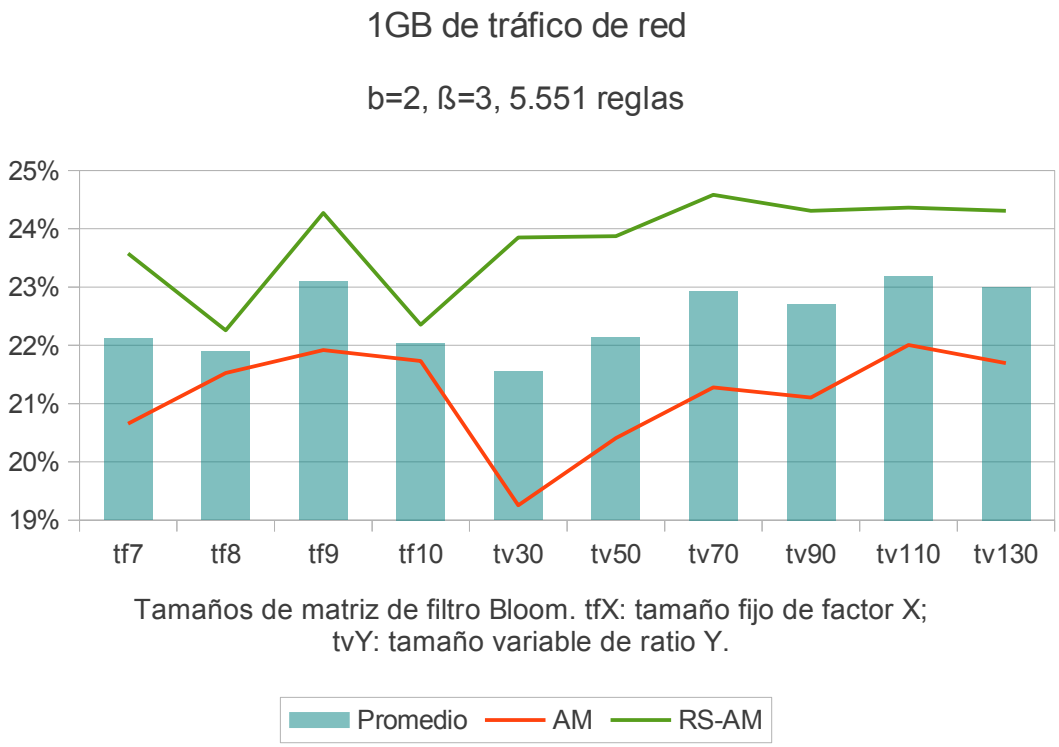
Por otra parte, se descartarán las funciones SDBM y SAX, por considerar que no están a la altura del rendimiento del resto. Las funciones RS y XOR se tendrán en consideración, ya que pueden resultar útiles al combinarlas con las funciones AA y AM. Se recuerda que estas dos últimas funciones compartían la misma distribución de frecuencia de resúmenes hash, por lo que sería interesante realizar pruebas usando una combinación de ambas.

Dicho esto, se ha considerado que seguirán teniéndose en cuenta las funciones AA, AM, RS y XOR. Por tanto, se realizarán pruebas utilizando cada una de ellas, tanto de manera individual como en parejas.

7.3.1.3. Estudio del tamaño de la matriz del filtro Bloom

Le toca ahora el turno a la configuración del tamaño de matriz. Se utilizaran dos tipos de configuraciones diversas: tamaño fijo y tamaño variable, con diferentes valores de factor y ratio, respectivamente. De momento, el objetivo no es decidir sobre el tipo de configuración empleada, sino descartar los valores de las configuraciones que no funcionen correctamente. Para ello se tomarán valores que vayan en torno a los utilizados ahora por defecto: factor 8 y ratio 90.

Ya se han decidido qué funciones hash se usarán en la segunda fase, por lo que en este estudio no tiene sentido que se incluyan una gran cantidad de ellas. Lo más interesante al respecto es conocer la evolución de una misma función hash a lo largo de las distintas configuraciones de tamaño de matriz. Para ello se escogerá una función que trabaje sola y una combinación de dos funciones. De esta forma se comprenderá mejor de qué manera afecta la variación del tamaño de la matriz al usar funciones solas y combinadas por separado. Los resultados se muestran en la gráfica 7.13.



Gráfica 7.13: Estudio del tamaño de la matriz del filtro Bloom.

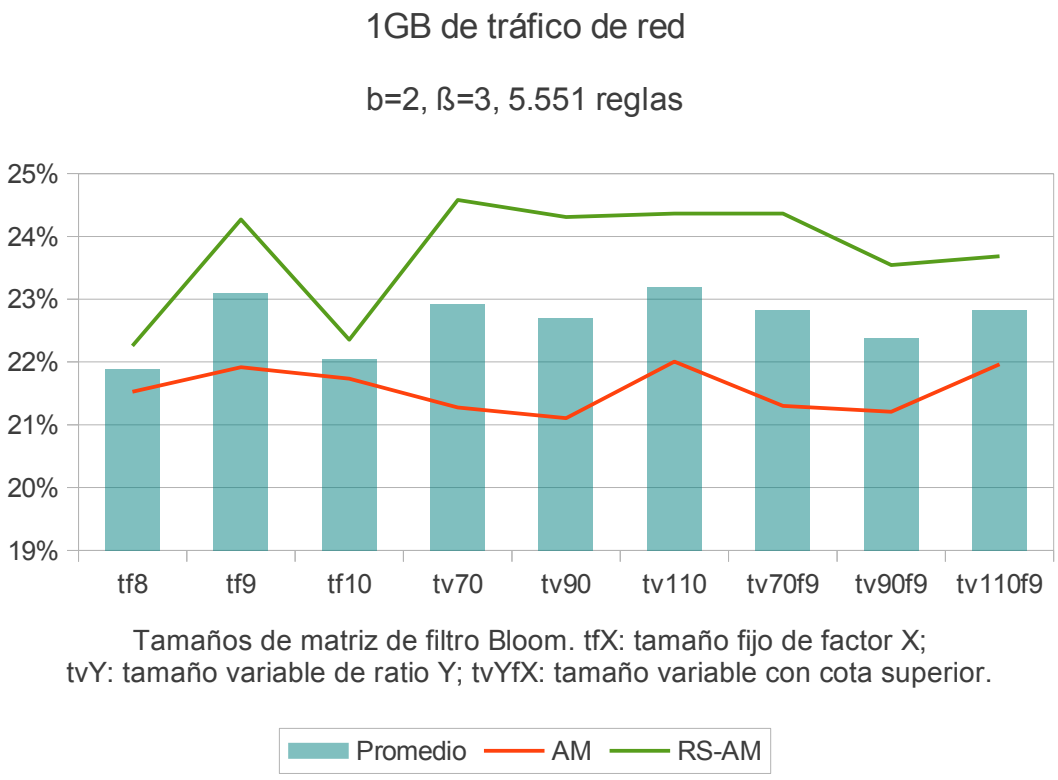
En la gráfica se podría realizar una división virtual entre la configuración de tamaño fijo y variable. De esta forma quedarían separados los cuatro primeros resultados de los seis últimos. En la primera parte, los valores asignados al factor de tamaño fijo van de 7 a 10 y en los cuatro casos se obtiene un mejor resultado en el valor 9, siendo muy superior a sus adyacentes en el caso de la combinación RS-AM.

A pesar de que el valor 7 ofrezca mejores resultados que 8 y 10 para la combinación RS-AM, este valor debería ser descartado puesto que es muy probable que en una matriz de estas características se produzcan demasiadas colisiones –tenga el lector en cuenta que Snort crea subgrupos de reglas en los que el número de patrones que se añaden al filtro Bloom supera con creces las 128 posiciones de esta matriz–.

Concluyendo de igual forma en la segunda parte de la gráfica se descartarán los valores 30 y 50 por ofrecer, en ambos casos, los peores resultados. El valor 130 también se descartará puesto que no supera el rendimiento del valor anterior 110. De esta forma se ahorraría en consumo de memoria

obteniendo el mismo resultado.

En definitiva, se mantienen los valores 8, 9 y 10 para el tamaño fijo y 70, 90 y 110 para el tamaño variable. Se realizará una segunda prueba que incluirá la configuración híbrida resultante de la combinación de ambos parámetros, obteniendo tamaños de matriz variables acotados superiormente. Se establecerá el límite en un factor 9, por ser el valor que mejores resultados ha tenido, y los tamaños variables vendrán determinados por los ratios 70, 90 y 110. En la gráfica 7.14 se muestran dichos resultados.



Gráfica 7.14: Estudio del tamaño de la matriz del filtro Bloom (segunda parte).

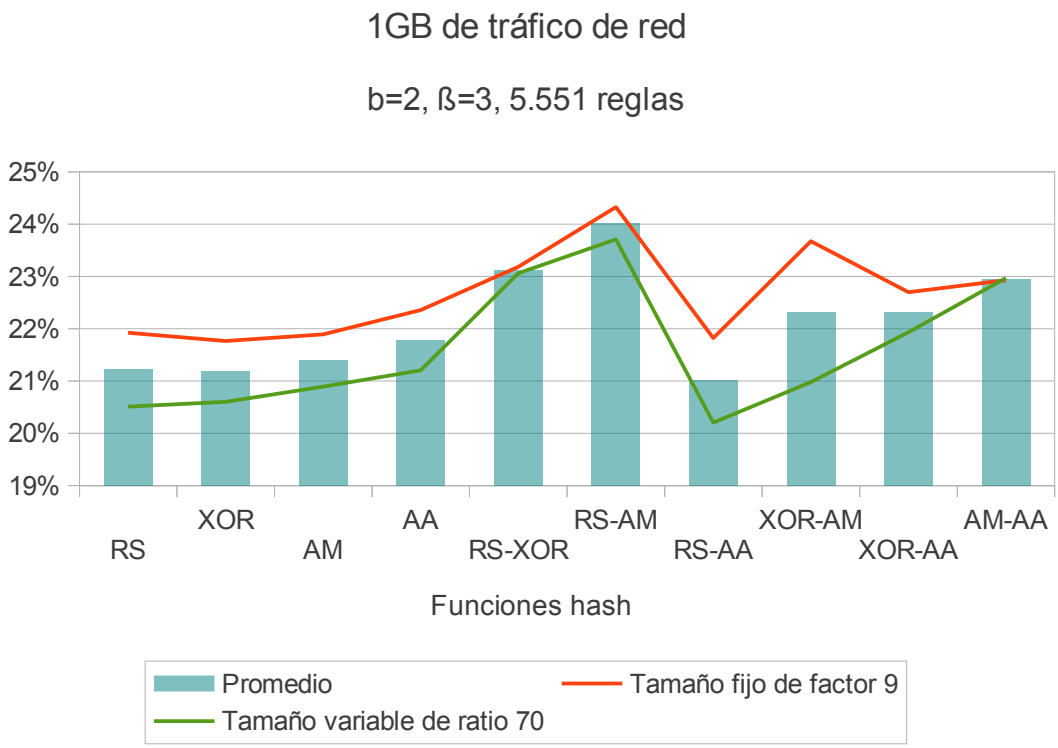
En esta representación, conviene comparar los tres valores centrales con los tres últimos. Ambos cuentan con tamaño variable, pero los últimos están acotados superiormente a un tamaño de matriz de factor 9. En ningún caso, una configuración de tamaño variable con cota superior cuenta con un mejor rendimiento que una configuración variable sin cota. En el caso de la combinación RS-AM esta afirmación puede apreciarse a simple vista sobre la gráfica. Menos evidente es en el caso de una única función hash. No obstante, este hecho se puede achacar al pequeño rango de salida que ofrecían las funciones AM y AA –visto en el capítulo seis–.

Obviamente se dará más peso a los resultados que se obtengan de las funciones con mayor rendimiento. Por lo que la opción de la configuración de tamaño de matriz híbrida será descartada para la realización de las siguientes pruebas. Del mismo modo, se escogerán únicamente la configuración de tamaño fijo de factor 9 y la de tamaño variable de ratio 70, descartando el resto.

7.3.1.4. Estudio final

En este punto, contamos con un valor seleccionado del parámetro *ß*, mientras que aún no se ha llegado a una decisión final acerca de las funciones hash a utilizar, ni tampoco del tamaño de la matriz del filtro Bloom. De este último estudio saldrá una única configuración que será usada en la segunda fase de las pruebas de rendimiento.

Las posibilidades respecto a la configuración del tamaño de la matriz quedaron reducidas a un tamaño fijo de factor 9 y un tamaño variable de ratio 70. Por otro lado, en cuanto a las funciones hash, aún se tenían dudas en cuanto a usar una única función hash o una combinación de dos funciones, por lo que se probarán todas las combinaciones posibles de las funciones RS, XOR, AM y AA. En la gráfica 7.15 se muestra el rendimiento de cada una de las 20 posibles configuraciones que restan.



Gráfica 7.15: Estudio final.

Se comprueba claramente que en cualquier tipo de configuración de función hash, es el tamaño fijo de factor 9 el que obtiene un mayor rendimiento. Siendo la opción RS-AM la que mejor rendimiento ofrece. Queda por tanto decidida la configuración final, que contará con una combinación de las funciones hash RS y AM, y un tamaño de matriz fijo de factor 9 (línea roja).

Más allá de extraer un dato obvio de la gráfica, sería interesante aprovechar el hecho de que se están empezando a obtener datos reales de rendimiento, por lo que podrían compararse estos datos con algunas suposiciones que se enunciaron en capítulos anteriores.

En primer lugar, ha quedado claro que en estas condiciones específicas, de tipo de tráfico de red y base de firmas, conviene el uso de funciones hash en pareja, en lugar del uso individual. Por lo que el compromiso entre velocidad de ejecución de sigMatch y tasa de candidatos se decanta en este caso a favor de la combinación de funciones hash.

También, habría que recalcarse el hecho de que las dos funciones de creación propia, AM y AA, se han hecho un hueco entre funciones más elaboradas y reconocidas. A veces, sucede que en la simplicidad reside la eficiencia.

Esto puede servir para aprender que no vale adaptar cualquier desarrollo a cualquier entorno. Cada creación, tiene un entorno específico en el que su funcionamiento es óptimo. Posiblemente, el resto de funciones hash sean muy eficientes en distintos ámbitos, como la realización de resúmenes de textos de grandes dimensiones. Pero quizás no sean las más indicadas para llevar a cabo operaciones sobre cadenas de caracteres muy cortas, como las de este caso.

Sin más dilación, se continúa con la segunda fase de las pruebas de rendimiento, en la que se utilizarán los parámetros ya elegidos, y con ellos se evaluará el rendimiento del filtro en varias situaciones.

7.3.2. Fase II: Rendimiento en diversos entornos

Con la configuración seleccionada, en base a los resultados obtenidos en la primera fase, se realizarán pruebas de rendimiento en entornos de diferentes características.

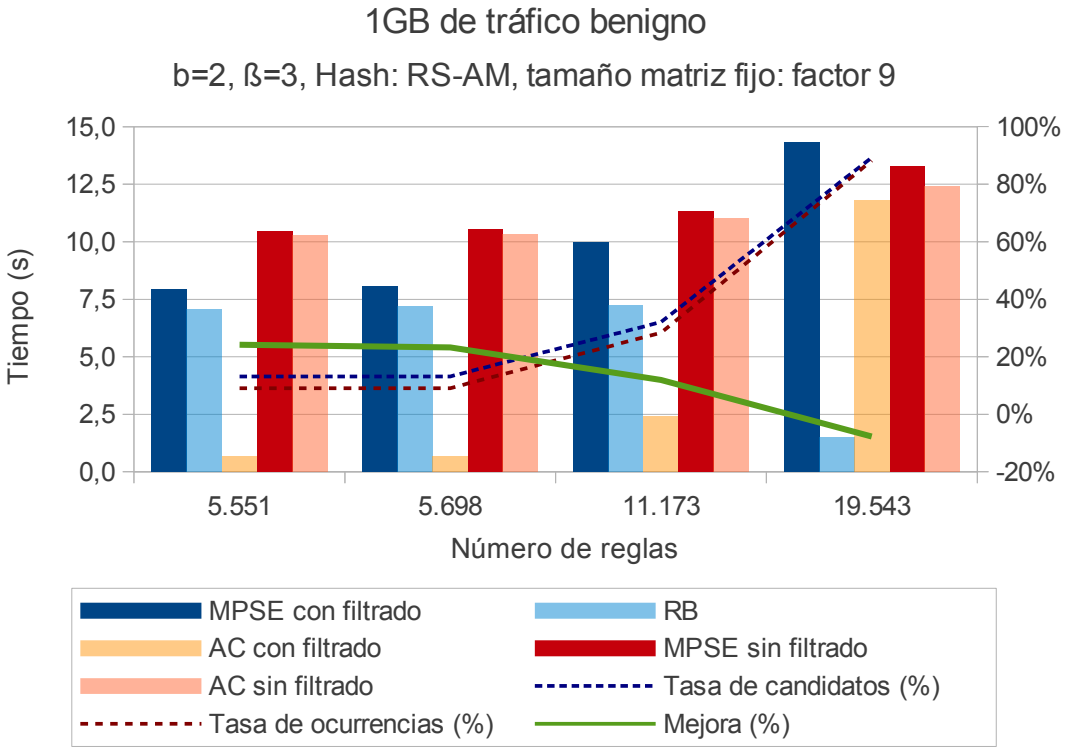
Se recuerda al lector que en este mismo capítulo se definieron cuatro grupos de reglas diversos. Estos grupos corresponden al mismo juego de reglas proporcionado en la versión 2.9.1.0 de Snort.

Se explicó que un juego de reglas de estas características contienen varios conjuntos de reglas –o *rulesets*–, y que dentro de ellos existe un listado de reglas. No todos los *rulesets* están habilitados por defecto ni todas las reglas de su interior están habilitadas. Aprovechando esta situación se obtuvieron cuatro grupos de reglas. A continuación se detalla cómo se obtuvo cada grupo.

1. Se incluyen los *rulesets* habilitados por defecto, y únicamente las reglas activadas por defecto. Resultado: 5.551 reglas.
2. Se incluyen todos los *rulesets*, y únicamente las reglas activadas por defecto. Resultado: 5.698 reglas.
3. Se incluyen los *rulesets* habilitados pro defecto, y se activan todas las reglas. Resultado: 11.173 reglas.
4. Se incluyen todos los *rulesets*, y se activan todas las reglas. Resultado: 19.543 reglas.

También, se disponen de cinco ficheros PCAP con tráfico de red capturado. Dos de ellos correspondientes al tráfico de una red interna, por lo que resulta tener un porcentaje de ataques muy bajo, de ahí que se le denominen como "capturas benignos". Los tres restantes contienen tráfico más agresivo. De esta forma, se pueden estimar hasta 20 entornos diversos al combinar diferentes conjuntos de reglas con las cinco capturas de tráfico.

En las gráficas 7.16 y 7.17 se muestran los resultados correspondientes a las dos capturas de tráfico benigno. Cada paquete ha sido analizado usando los cuatro grupos de reglas.



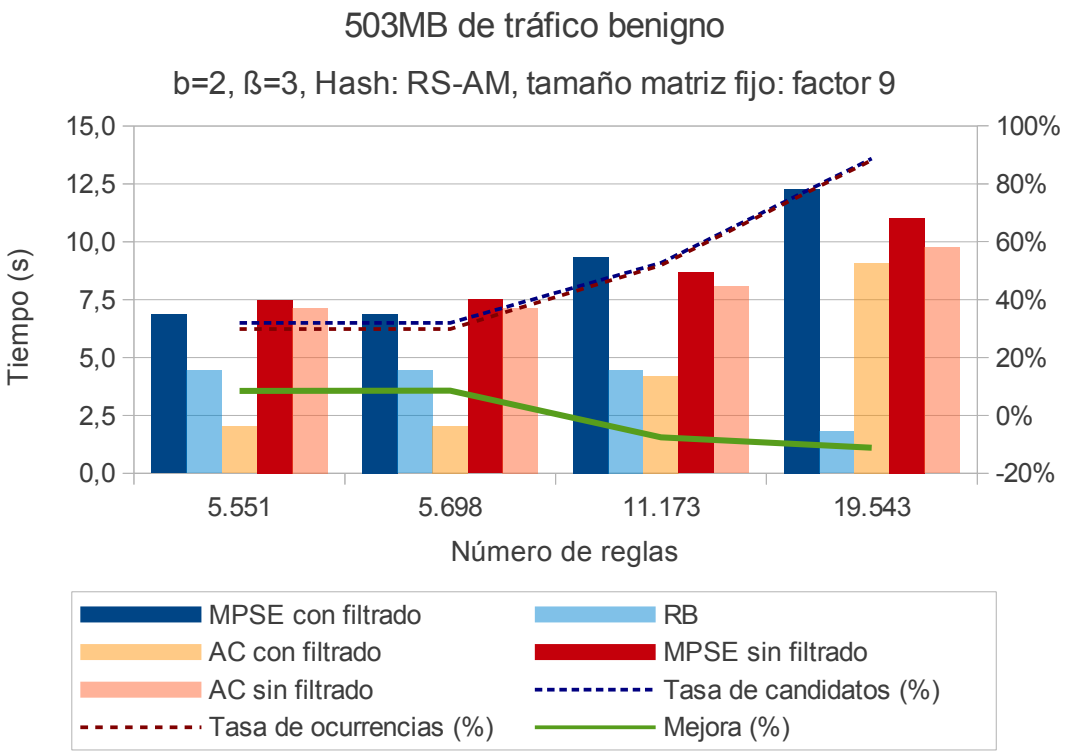
Gráfica 7.16: Rendimiento: 1GB de tráfico benigno.

En ambos casos, la mejora que se obtiene es mayor para los dos primeros grupos de reglas, que casualmente tienen un número de reglas menor. De hecho, para el paquete de 503MB, en los dos últimos grupos no se obtiene ninguna mejora, sino que el sistema formado por sigMatch y Snort ofrece peores resultados que al ejecutar Snort sin filtrado. Esto no quiere decir que a mayor número de reglas el rendimiento del filtro empeore, afirmarlo sería ir en contra de los principios de sigMatch, y no es el caso.

No se ponen en entre dicho las conclusiones postuladas, en las que se afirmaban que un número de reglas mayor produciría una mejora aún mayor. Simplemente se desea añadir que influye en mayor medida el tipo de reglas que la cantidad.

Ya se ha explicado que el rendimiento que pueda ofrecer la integración de sigMatch en un sistema de búsqueda de patrones dependerá de muchos factores. Y sobre todo se ha insistido en que será determinante la correlación existente entre el tráfico analizado y la base de firmas, o más concretamente la estructura de búsqueda construida en base a la base de firmas.

Por lo que este descenso en el rendimiento se puede deber a que las reglas que se hayan activado en el tercer y cuarto grupo –que por defecto estaban desactivadas–, provoquen un gran cambio en la correlación ya citada, viéndose a su vez influido el rendimiento del conjunto.



Gráfica 7.17: Rendimiento: 503MB de tráfico benigno.

Se recuerda que una correlación mayor provocaría un peor rendimiento, puesto que el coste computacional del filtro sigMatch se vería incrementado en mayor medida.

Convendría hablar sobre las tasas de candidatos y de ocurrencias. La primera, indica el porcentaje paquetes de red que el filtro envía a Aho-Corasick. Mientras que la segunda se corresponde con el porcentaje de paquetes que Aho-Corasick detecta como ocurrencias, y por tanto como posibles alerta; será una función posterior –conocida como función *Match*– la que comprobará si verdaderamente se trata de una alerta –en una regla existen más opciones que indicarán si se ha de emitir una alerta, no siendo el contenido el único factor determinante–.

En ambos casos, una tasa de candidatos menor –correspondiente con un menor tráfico candidato– repercute en un mejor rendimiento. Tiene lógica, puesto que la tasa de candidatos estará ligada a la correlación existente entre el tipo de tráfico y la base de firmas. Una tasa de candidatos menor significa también una menor correlación, lo que implica un menor coste computacional de sigMatch, que a su vez implica un mejor rendimiento.

Es cierto que podrá darse el caso de que ante una tasa de candidatos pequeña se tenga un rendimiento muy bajo. Esto se podría producir puesto que la relación entre la tasa de candidatos y la correlación mencionada no es tan directa. Puede ser que la tasa de candidatos sea baja pero que la correlación sea algo superior. Esto produciría que se tuvieran que ejecutar muchas veces la rutina completa de sigMatch, para así comprobar si se debe marcar un paquete como candidato, o no. Esta situación será muy poco probable, por lo que se podrá afirmar con gran seguridad que ante una tasa de candidatos pequeña, también se obtendrá un mejor rendimiento, puesto que la correlación entre

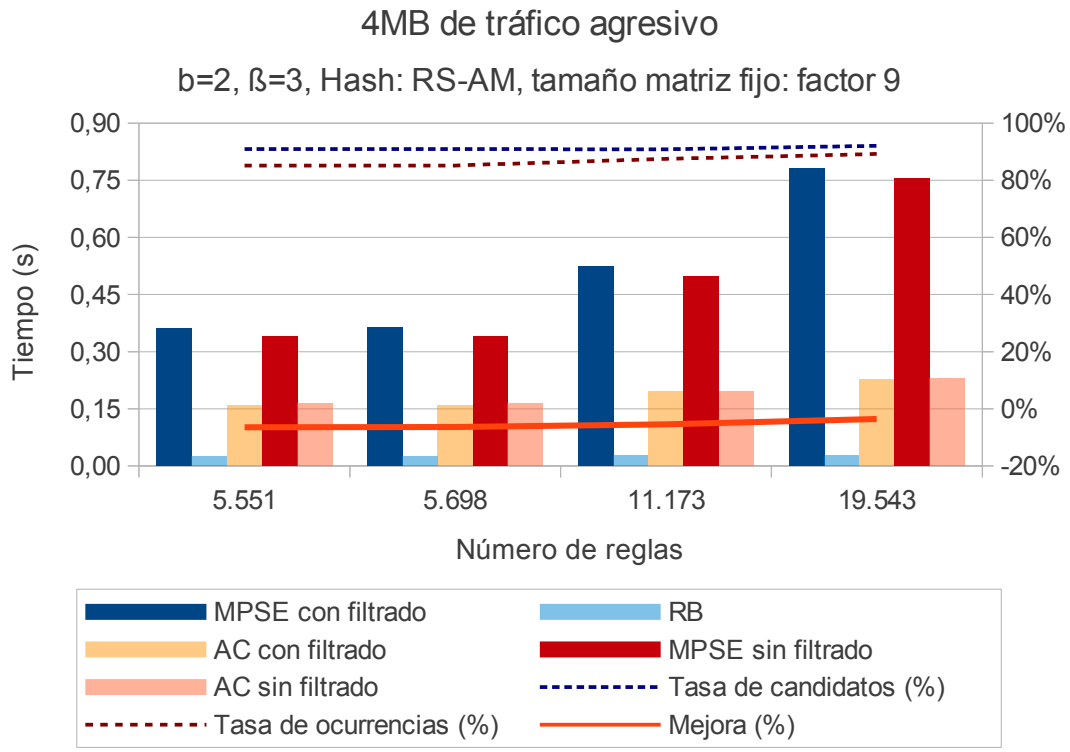
tráfico y firmas será baja.

Un último apunte acerca de la tasa de candidatos: note el lector que cuanto más juntas estén las tasas de candidatos y de ocurrencias, significará que la cantidad de falsos positivos es menor. Este hecho también influye notablemente en el rendimiento. De hecho, en la fase I fue decisivo para elegir una combinación de dos funciones hash en lugar de una sola, debido a que se reducían notablemente la cantidad de falsos positivos provocados por el filtro Bloom.

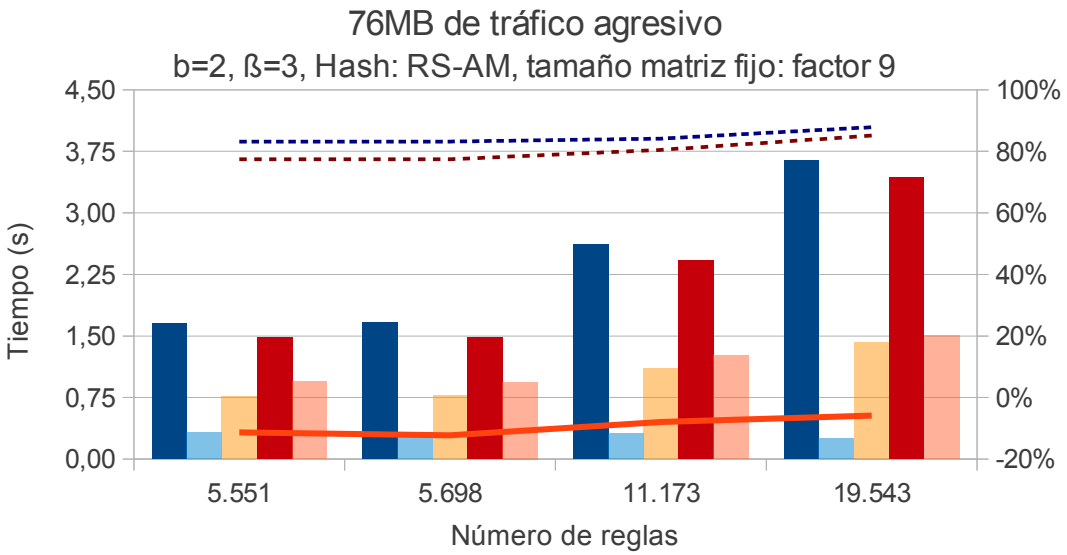
En las gráficas 7.18, 7.19 y 7.20 se presentan los resultados correspondientes a las capturas de tráfico agresivo. En ninguno de los tres casos se consigue una mejora al integrar sigMatch en Snort. Se demuestra una vez más la importancia de la tasa de candidatos en el rendimiento de sigMatch.

Sin embargo, es curioso comprobar como, en general, se obtienen resultados algo mejores en los grupos de reglas 3 y 4. Esto puede resultar contradictorio si se observa que también aumenta la tasa de candidatos. Algunos párrafos atrás se afirmó que la tasa de candidatos es un indicativo importante del rendimiento, de manera que cuando dicha tasa disminuía, el rendimiento mejoraba, y viceversa. En este caso no sucede así, puesto que cuando la tasa aumenta en los grupos de reglas 3 y 4, el rendimiento también lo hace. Esto no quiere decir que se tenga que descartar la conclusión que se obtuvo, sino que se debe ampliar.

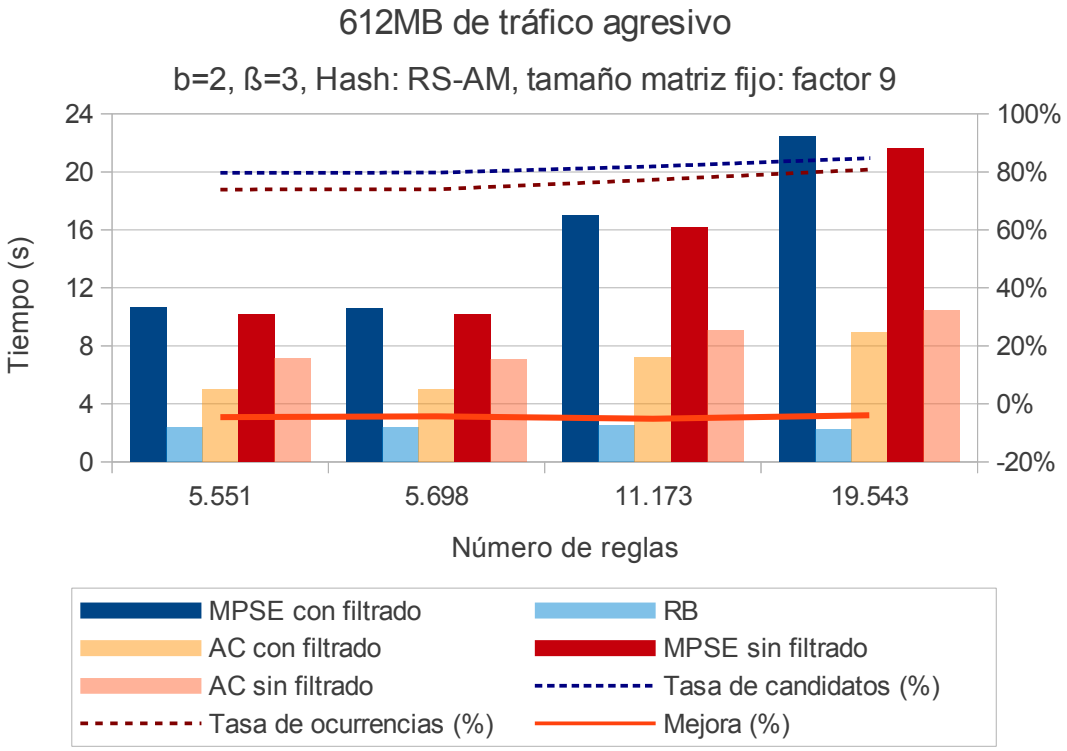
También en párrafos anteriores, se ha hablado de la relación entre las tasas de candidatos y de ocurrencias, explicando que cuanto menos diferencia haya entre ellas, significará que el filtro lanza menos falsos positivos. A su vez, se dedujo que ello provocaría un mejor rendimiento. Por tanto, se podría afirmar que cuanto más próximas se sitúen ambas tasas, el rendimiento será mayor.



Gráfica 7.18: Rendimiento: 4MB de tráfico agresivo.



Gráfica 7.19: Rendimiento: 76MB de tráfico agresivo.

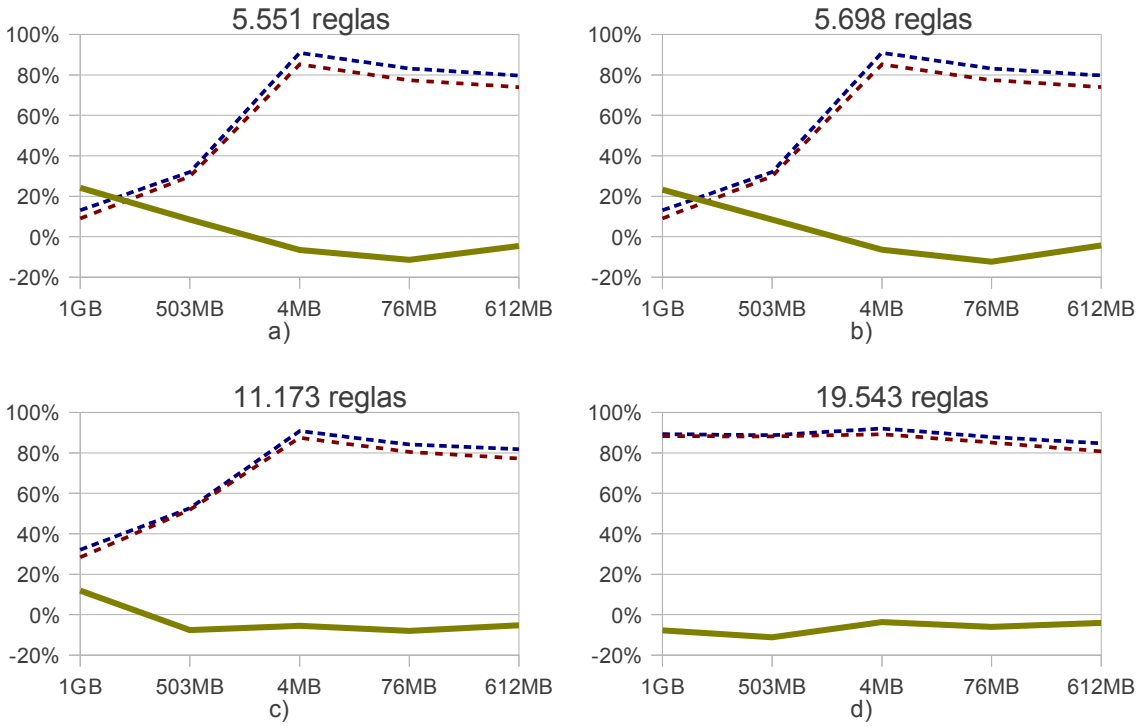


Gráfica 7.20: Rendimiento: 612MB de tráfico agresivo.

Dicho esto, se podrá afirmar que si el incremento de la tasa de candidatos no es muy elevado, la diferencia entre ambas tasas tendrán mucho peso en el rendimiento final. Es por ello por lo que el rendimiento aumenta para los grupos de reglas 4 y 5 de las tres gráficas anteriores. Si se repasan las

gráficas 7.16 y 7.17 se comprobará como a pesar de que las tasas de candidatos y de ocurrencias están más unidas en los grupos 4 y 5, el rendimiento es peor, y esto se debe a que el aumento que ha sufrido la tasa de candidatos es bastante elevado.

Respecto a las tasas de candidatos, a continuación se muestran cuatro gráficas en las que se comparan los factores de mejora con las tasas de candidatos. Cada gráfica se corresponde con un grupo de reglas determinada y contendrán los valores de cada captura de tráfico utilizada hasta ahora.



Gráfica 7.21: Comparativa de las tasas de candidatos y de ocurrencias con los factores de mejora usando las cinco capturas de red disponibles hasta ahora. Línea azul discontinua: tasa de candidatos, Línea roja discontinua: tasa de ocurrencias. Línea verde continua: factor de mejora.

Se puede comprobar como la relación existente entre la tasa de candidatos y el factor de mejora es evidente, a mayor tasa de candidatos, se reduce el factor de mejora, normalmente hasta una cota inferior. Esto se debe a que una tasa de candidatos alta provoca que se devuelva el control de la ejecución a Aho-Corasick en multitud de ocasiones, por lo que no es posible que se produzca un empeoramiento excesivo de la aplicación.

También, en la tabla 7.3 se muestran los valores correspondientes a las gráficas 7.21.

Si de la tabla 7.3 se extraen las relaciones entre las tasas de candidatos y los respectivos factores de mejora, se podría realizar una estimación de la relación entre ambos valores. Esto se muestra en la gráfica 7.22.

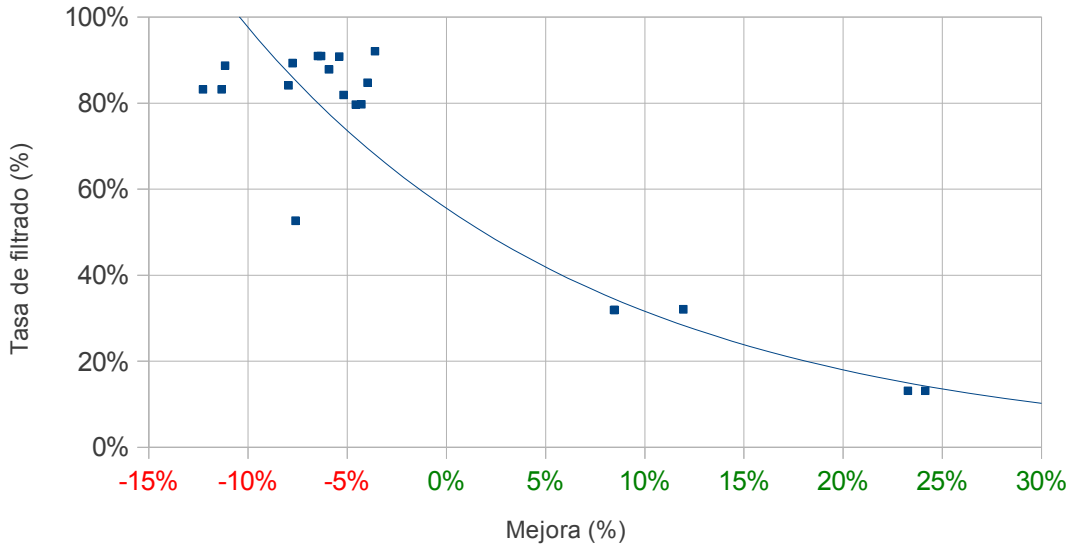
Se puede comprobar como, por mucho que se alcance el máximo valor de tasa de candidatos no se alcanza un rendimiento extremadamente pobre. Esto se debe a que, como se ha explicado

	1GB	503MB	4MB	73MB	612MB	Base de firmas
Tasa de candidatos	13,2%	31,9%	90,9%	83,2%	79,7%	*5.551 reglas*
Tasa de ocurrencias	9,1%	29,8%	85,2%	77,4%	73,9%	
Factor de mejora	24,1%	8,4%	-6,5%	-11,3%	-4,6%	
Tasa de candidatos	13,1%	31,9%	90,9%	83,2%	79,7%	*5.698 reglas*
Tasa de ocurrencias	9,1%	29,9%	85,2%	77,4%	74,0%	
Factor de mejora	23,3%	8,5%	-6,3%	-12,3%	-4,3%	
Tasa de candidatos	32,1%	52,7%	90,4%	84,1%	81,9%	*11.173 reglas*
Tasa de ocurrencias	28,4%	51,9%	87,5%	80,5%	77,3%	
Factor de mejora	11,9%	-7,6%	-5,4%	-8,0%	5,2%	
Tasa de candidatos	89,3%	1,1%	92,1%	87,9%	84,7%	*19.543 reglas*
Tasa de ocurrencias	88,2%	88,2%	89,2%	85,1%	80,8%	
Factor de mejora	-7,8%	-11,1%	-3,6%	-5,9%	-4,0%	

Tabla 7.3: Valores de tasa de candidatos, tasa de ocurrencias y factor de mejora para los cuatro grupos de reglas disponibles y las cinco capturas de tráfico.

anteriormente, al resultar muchos candidatos, se pasa continuamente el control a Aho-Corasick, lo que provoca que sigMatch se ejecute en menos ocasiones, que en este caso es el algoritmo que provoca un peor rendimiento.

También, se verifica que existe una relación coherente entre tasa de candidatos y factor de mejora, por lo que se ha podido demostrar la dependencia de ambos valores.



Gráfica 7.22: Relación obtenida entre tasa de candidatos y factor de mejora.

Antes de terminar con este apartado, sería interesante contrastar algunas ideas en cuanto al coste computacional calculado anteriormente en este mismo capítulo. Se calcularon valores máximos y mínimos del consumo computacional de las rutinas de búsqueda de Aho-Corasick y sigMatch. Así mismo, se estimó un promedio del consumo computacional de cada algoritmo, en el que salía bastante favorecido sigMatch. Aún así, se aclaró que las probabilidades usadas para obtener tal promedio se calcularon teniendo en cuenta distribuciones uniformes de datos entrantes, por lo que dichos resultados no se corresponderían con la realidad.

Se dijo, que para una mejor estimación se deberían calcular las probabilidades teniendo en cuenta un entorno real, en el que existiera un determinado tráfico y una base de firmas específica. En este punto nos encontramos en dicha situación.

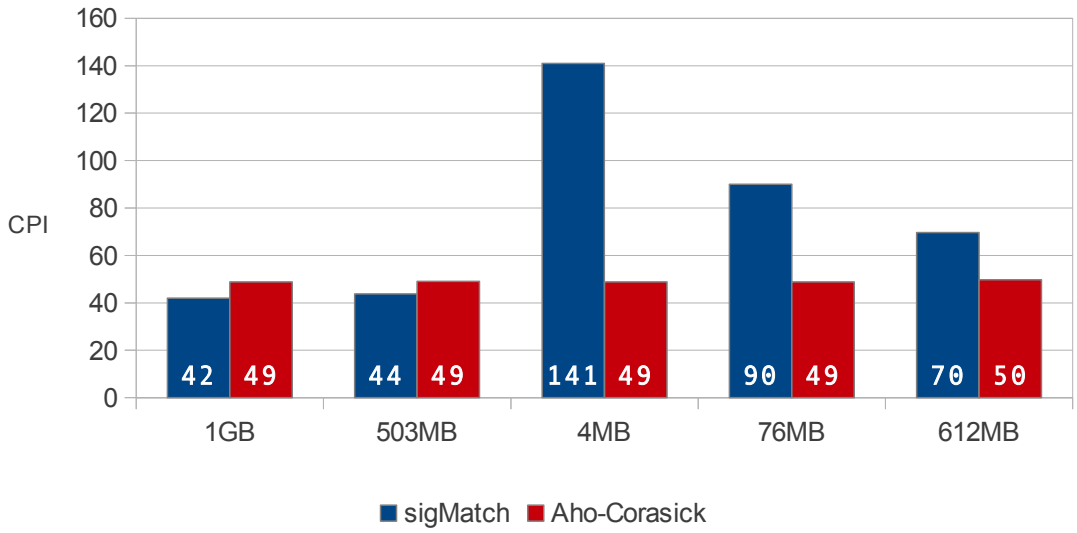
Se ha obtenido información que permite calcular dichas probabilidades para un entorno de tráfico benigno –en concreto la captura de datos de 1GB– y una base de 5.551 reglas –la misma que la utilizada en las pruebas realizadas hasta ahora–. En la tabla 7.4 se muestran las probabilidades calculadas en un primer momento, junto con las calculadas en este estudio y teniendo en cuenta un entorno específico.

	sigMatch						*A-C*
	P_{NFA1}	P_{NFA2}	P_{nostr}	P_{FB}	PE_{FB}	P_{L}	P_{DFA1}
Sin tener en cuenta el entorno	1,03%	0,59%	9,26%	100%	0,10%	28,89%	7,70%
Entorno: 1GB & 5.551 reglas	28,31%	2,19%	0,50%	89,37%	1,22%	26,32%	0,66%
Entorno: 503MB & 5.551 reglas	27,40%	3,96%	2,12%	83,09%	2,77%	36,49%	3,03%
Entorno: 4MB & 5.551 reglas	70,48%	22,79%	0,68%	96,58%	10,35%	20,74%	1,05%
Entorno: 76MB & 5.551 reglas	49,05%	16,68%	1,85%	89,42%	8,47%	25,44%	1,01%
Entorno: 612MB & 5.551 reglas	45,36%	9,84%	1,01%	94,17%	8,05%	23,59%	10,39%

Tabla 7.4: Probabilidades para el cálculo del promedio del coste computacional.

Estas nuevas probabilidades corroboran el hecho de que la previsión que se hizo del consumo computacional era demasiado optimista. Aplicando las nuevas probabilidades a la fórmula de dicho coste se obtienen los resultados mostrados en la gráfica 7.23.

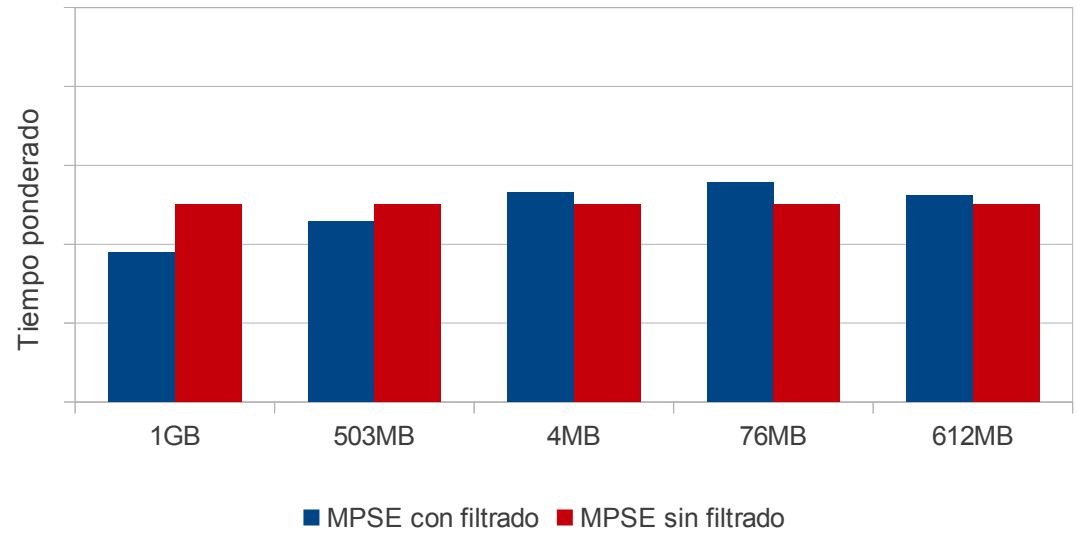
Se puede comprobar que en los dos primeros entornos, la estimación del coste computacional de sigMatch es menor que la de Aho-Corasick. Mientras que en entornos más agresivos, en los que se supone una correlación mayor entre tráfico y reglas, el consumo de sigMatch es bastante mayor que el de Aho-Corasick.



Gráfica 7.23: Consumo computacional promedio de los algoritmos Aho-Corasick y sigMatch, usando probabilidades calculadas teniendo en cuenta el entorno.

En su momento, se explicó que este comportamiento se debía al método de búsqueda de cada algoritmo. Mientras que Aho-Corasick usa una estructura basada en DFA, caracterizada por tener un consumo muy uniforme ante diversos entornos, sigMatch usa una estructura basada en NFA y filtros Bloom, que resulta ser muy buena en entornos favorables, pero que en entornos algo desfavorables se vuelve tremendamente ineficiente.

Con la gráfica 7.24 se pretenden comparar los resultados reales obtenidos en las pruebas de rendimiento con las estimaciones del coste computacional mostradas en la figura 7.23. Puesto que cada fichero de captura de datos proporciona tiempos muy distintos, se han ponderado todos los



Gráfica 7.24: Tiempo ponderado empleado por los motores de búsqueda, usando y sin usar el filtrado.

tiempos para facilitar la comparación a simple vista, tomando como referencia el tiempo empleado por el motor de búsqueda, cuando no incluye filtrado, de cada uno.

Las gráficas 7.23 y 7.24 muestran casos distintos. En la primera, se muestra el consumo computacional de cada algoritmo por separado. En cambio, en la segunda se muestra el tiempo ponderado empleado por el motor de búsqueda de patrones al usar filtrado y sin usarlo. No obstante, en este último caso, se pueden extraer algunas conclusiones triviales.

Cuando el menor tiempo empleado por el motor de búsqueda se dé al usar el filtrado previo, significará que en ese entorno sigMatch es más rápido que Aho-Corasick. Si en cambio, el menor tiempo corresponde con los casos en los que no se usa ningún filtrado, querrá decir que en ese entorno Aho-Corasick es más rápido que sigMatch.

En definitiva, de la gráfica 7.24 se podría deducir en los dos primeros casos que sigMatch es más rápido que Aho-Corasick, mientras que en el resto de situaciones es Aho-Corasick quien consigue un mayor rendimiento.

A pesar de que en su momento los cálculos del coste computacional no eran más que estimaciones, después de haberlos comparado con las pruebas reales realizadas en este apartado, se podría decir que estas estimaciones no iban mal encaminadas, lo que ratifica la hipótesis lanzada en su momento en la que se afirmaba de que sigMatch era un algoritmo muy dependiente del entorno, mientras que Aho-Corasick era más estable en este aspecto.

7.4. Conclusiones

Después de haber evaluado el rendimiento mediante pruebas en diferentes entornos, se puede llegar a la conclusión de que los factores que están más relacionados con el rendimiento obtenido son las tasas de candidatos y de ocurrencias. La tasa de candidatos, en concreto, dependerá de otros factores, como la correlación existente entre los datos del tráfico de red y la base de firmas, el número de falsos positivos, etc.

En el estudio de sigMatch, se llegó a la conclusión de que la cantidad de firmas era el principal factor que podía provocar un mejor o peor rendimiento. No se pone en duda que esto se cumpla al integrar sigMatch en determinadas aplicaciones, pero desde luego no sucede así en Snort. Desde este estudio se puede asegurar que el rendimiento dependerá principalmente del entorno que se dé.

Por supuesto, existen puntos en común entre ambos estudios. Imagínese un estudio de la integración de sigMatch en Snort en el que se tienen N bases de firmas con cantidades y longitudes de firmas distintas entre sí; imagínese también que para el estudio se emplea cierto tráfico para su análisis; supóngase que la conjunción de dicho entorno de red con cada una de las N bases proporcionan una correlación muy similar, y que ésta es tan pequeña que hace que en el estudio se observe un buen factor de mejora en todos los casos. Habiendo supuesto todo lo anterior, se podrá afirmar que el mejor factor de rendimiento lo conseguirá la base de firmas con mayor número de reglas.

He aquí el punto en el que confluye el estudio de sigMatch con el realizado en este documento, puesto que se cumplen los corolarios vertidos en este documento –la correlación provocada por el

entorno de red y las bases de firmas serán un factor determinante en cuanto a rendimiento– y en el de sigMatch –mayor número de reglas implica una mejora también mayor–.

Los autores de sigMatch no trataron la tasa de candidatos como un factor determinante, sino que consideraron como factor más decisivo el tamaño de la base de firmas. Aún así, publicaron datos de dicha tasa para ClamAV y DBLife. El primero de ellos obtuvo una tasa de candidatos de un 5,3%, mientras que el segundo consiguió un 3%. Ambos cosecharon unos factores de mejora de 11,7X y 15X, respectivamente. Hubiera sido muy interesante conocer también la tasa de candidatos obtenida en Bro, cuyo factor de mejora fue de 4,4X. No obstante, la relación entre las tasas de candidatos y los factores de mejora en ClamAV y DBLife es bastante indicativa y corroboran todo lo expuesto anteriormente.