Universidad Nacional del Litoral

Proyecto Final de Carrera

Diseño de un sistema de detección de anomalías en redes de computadoras.

Informe de avance 2

Pineda Leandro
Santa Fe
2 de junio de 2017

Resumen

1. Introducción

La detección en tiempo real de ciertos patrones en una red de datos tales cómo escaneo de puertos, ataques de denegación de servicio, expansión de *malware*, entre otros, es de vital importancia para salvaguardar la integridad de la infraestructura de datos de cualquier organización. Estas anomalías pueden encontrarse analizando los flujos de datos y llevando cuenta de todos los paquetes que atraviesan los puntos de acceso a la red. En la capa de transporte, el protocolo TCP (que provee conexión host a host sobre IP) provee información suficiente para identificar estos patrones. Para llevar a cabo el análisis en tiempo real de los segmentos TCP que son transportados, utilizaremos el modelo de *data streaming*: este se basa esencialmente en algoritmos que procesan una única vez los datos, dado que transferirlos y almacenarlos no es posible en la práctica, y permiten determinar cuales son los elementos más comunes en el stream de datos, así como algunos estadísticos (como la media, mediana, histograma), entre otros.

2. El modelo de data streaming

En este modelo, las entradas que van a ser procesadas no están disponibles para ser accedidas aleatoriamente desde disco o memoria, sino que llegan como uno o mas flujos continuos de datos. Los streams de datos son diferentes de los modelos relacionales convencionales en varios aspectos:

- Los elementos del stream deben ser procesados de manera *online*.
- Los sistemas que procesan los datos no tienen control sobre el orden en los elementos de la entrada.
- Los stream de datos pueden ser infinitos.
- Una vez que un elemento es procesado, este se descarta.[1]

Otra característica que es conveniente remarcar se relaciona con las consultas a los datos procesados: podemos hacer una distinción entre *consultas únicas* y *consultas continuas*[1]. Las consultas únicas (como aquellas que se hacen mediante un DBMS tradicional) son evaluadas una vez sobre un *snapshot* de un conjunto de datos. La consultas continuas, por otro lado, son evaluadas continuamente mientras el stream de datos esta siendo procesado. Las respuestas a estas consultas pueden ser almacenadas y actualizadas en la medida que llegan nuevos flujos de datos, o pueden ser origen de otro stream de datos.

Podemos pensar a los segmentos TCP que pasan por un *gateway* como los eventos o *keys* a ser procesados. Aunque estos ocurren de manera secuencial, pertenecen a diferentes sesiones que están activas al mismo tiempo. Las *keys* pueden ser agrupadas mediante la 5-tupla dirección de IP de origen, destino, número de puerto de origen y destino y protocolo. De esta manera se puede analizar el comportamiento del tráfico de red de cada una de las sesiones en busca de anomalías. Sin embargo, el espacio de las *keys* es tan grande que llevar registro de todos los eventos es imposible: consideremos el problema de determinar la frecuencia de ocurrencia de cierto evento, perteneciente a algún universo de eventos posibles U. Para obtenerla basta con llevar registro de la frecuencia f_i por cada elemento $i \in U$: dado $U_0 = \{a, b, c\}$ y una serie o stream de eventos $S = \{a, a, b, a, c, b, a\}$, la frecuencia de ocurrencia de cada elemento de U_0 es $f_a = 4$, $f_b = 2$ y $f_c = 1$. A pesar de su simpleza, el costo de memoria de este algoritmo crece exponencialmente cuando la cantidad de eventos posibles $|U|$ aumenta. En términos de implementación, para representar un universo de posibles elementos U tal que $|U| = 2^{40} \approx 10^{12}$ tenemos que se necesita almacenar en memoria $2^{40} * 32\ bits \equiv 4096$ GB en contadores, uno por cada evento posible de U. Este tipo de problemas y similares llevaron al desarrollo de los llamados *modelos de streaming* y la utilización de diferentes técnicas de conteo: bajo esta abstracción, los algoritmos procesan la entrada una única vez y deben calcular de manera precisa varios resultados usando recursos (espacio y tiempo por elemento) de forma estrictamente sublineal al tamaño de la entrada[2]. Existen diferentes algoritmos para procesar y obtener información acerca de los eventos usando estructuras de datos que utilizan el espacio de memoria eficientemente. Sin

[1] En algunas aplicaciones los elementos pueden ser almanacenados para ser procesados posteriormente, pero en un modelo de streaming "puro"los elementos son procesados una única vez.

embargo, estos métodos no calculan la frecuencia exacta de cada evento sino que la estiman: en general, para cantidades masivas de eventos basta con tener una buena aproximación de las frecuencias para identificar anomalías.

El problema de los eventos frecuentes consiste en procesar una serie consecutiva de elementos y encontrar aquellos que ocurren más frecuentemente en un período de tiempo. Es un problema muy estudiado en la minería en streams de datos debido a que la resolución de muchos problemas se basan directa o indirectamente en la identificación de eventos frecuentes.

Los algoritmos para encontrar elementos frecuentes pueden dividirse en dos clases. Aquellos basados en técnicas de conteo llevan registro de un subconjunto de elementos, y monitorean los contadores asociados con los mismos. Por cada entrada nueva, el algoritmo decide si guardar el elemento o no, y de hacerlo, con que valor lo inicializa. Por otro lado, los algoritmos basados en sketchs realizan proyecciones lineales aleatorias de las entradas[3] (vistas como vectores de características) y por lo tanto, no almacenan explícitamente los elementos de entrada. Estos últimos tienen ciertas propiedades que son de gran utilidad para procesamiento de múltiples streams de datos.

3. Detección de anomalías

Cómo se mencionó anteriormente, existen muchos indicadores de escenarios que pueden tener impacto negativos en una infraestructura de red. Una práctica muy común en la etapa reconocimiento[2], por donde comienzan todos los ataques, es el escaneo de puertos; uno o varios atacantes envían paquetes a un rango de puertos para determinar que servicios están activos, generando así grandes volúmenes de tráfico en la red. Otro escenario crítico es el de *command and control (C2)* en donde el atacante tiene control de un conjunto de equipos ya infectados y utiliza sus recursos para atacar otro objetivo. Esto también genera tráfico anómalo y es un indicador crítico ante el cual se deben tomar medidas inmediatamente. Otro indicador son las fluctuaciones repentinas en los flujos de datos, que pueden indicar ataques de denegación de servicio (DoS).

La lista de ataques podría extenderse, pero de forma general pueden definirse dos tipos de comportamientos anómalos que son útiles para identificar amenazas. Para esto es necesario modelar el problema bajo el modelo de *data streaming*.

3.1. Modelado del problema

Consideremos los segmento TCP que atraviesa un punto de acceso; estos pueden ser representados como un *stream* de eventos, donde cada elemento es una tupla (x, v_x). El elemento x pertenece a un dominio $T = \{0, 1, 2, \ldots, n-1\}$ con $|T| = n$, y v_x es un valor asociado a x. Para el caso de detección de eventos en tráfico de red, cada *key* x identifica un segmento TCP y está formado por la 5-tupla IP de origen, IP de destino, puerto de origen, puerto de destino y protocolo. El valor $v_x = 1$ representa cantidad de paquetes identificados por x. Dado un *stream* de eventos o *keys*, definimos ***heavy hitters*** como aquellos elementos que aparece más frecuentemente en el *stream* de eventos. Los ***heavy changers*** son aquellos elementos que presentan inconsistencias significativas entre el comportamiento observado y el comportamiento normal del flujo de datos (el cual se basa en lo ocurrido en el pasado) en un período de tiempo acotado[5]. En un algoritmo de detección de ***heavy keys***[3] típicamente se realizan dos procedimientos: en el primero, llamado de actualización, el valor de cada elemento es procesado y se almacenan los resultados en una estructura de datos; en el segundo, de detección, se examina la estructura de datos en cada época y se determinan los *heavy keys*.

Para detectar *heavy keys* se realizan estimaciones de frecuencias en ventanas de tiempo o épocas. En cada época, sea $S(x)$ la suma de los valores v_x del elemento x. Sea $D(x)$ la diferencia (en valor absoluto) de $S(x)$ en la época actual y la anterior. Además, sea $U = \sum_{x \in T} S(x)$ la suma total de todos los elementos x en una época. El problema de detectar *heavy keys* consiste en encontrar aquellos elementos cuya suma o diferencias excedan, en valor absoluto, al parámetro ϕ en una época. Formalmente, definimos cómo *heavy hitters* a aquellos elementos x con $S(x) \geq \phi_1$,

[2] El modelo de seguridad informática llamado *cyber kill chain* describe las 7 etapas que todo atacante ejecuta para lograr su objetivo.[4]

[3] Este término suele utilizase para referirse a ambos tipos de anomalías

y *heavy changers* a los elementos x con $D(x) \geq \phi_2$. En adelante, referiremos indistintamente a los parámetros ϕ_1 y ϕ_2 como ϕ, teniendo en cuenta que pueden ser diferentes.

3.2. Elementos frecuentes en data streams

El problema de los elementos frecuentes es uno de los problemas más estudiados debido a su valor y a la simplicidad de su enunciado. Es importante en si mismo y como subrutina en procesos más complejos sobre *streams* de datos. Antes de describir los algoritmos para encontrar elementos frecuentes es necesario enunciar formalmente el problema.

Elementos frecuentes

Dado un stream S de n elementos $t_1, t_2, \ldots, t_n$, la frecuencia del elemento i es $f_i = |\{j|t_j = i\}|$ (es decir, la cantidad de índices j donde el jth elemento es i). Los ϕ elementos frecuentes están dados por $\{i|f_i > \phi n\}$.

Ejemplo: El stream $S = \{a, a, b, a, c, b, a\}$ tiene $f_a = 4$, $f_b = 2$ y $f_c = 1$. Para $\phi = 0{,}2$ los elementos frecuentes son a y b.

Encontrar exactamente los ϕ elementos frecuentes puede ser costoso en términos de recursos: un algoritmo que resuelve el problema de los elementos frecuentes debe usar una cantidad lineal de espacio[6]. Para relajar el requerimiento de recursos se utiliza una aproximación a la solución del problema.

ϵ-aproximación de elementos frecuentes

Dado un stream S de n elementos una ϵ-aproximación de los elementos frecuentes esta dada por el conjunto F tal que todos los elementos $i \in F$ tengan frecuencia $f_i > (\phi - \epsilon)n$, y que no exista $i \notin F$ con $f_i > \phi n$. Dicho de otra manera, una ϵ-aproximación de los elementos frecuentes consiste en encontrar todos los elementos con frecuencia mayor o igual a ϕn, de los cuales ninguno tiene frecuencia menor que $(\phi - \epsilon)n$.

Estimación de la frecuencia de un elemento

Un problema relacionado a los anteriores consiste en estimar la frecuencia de los elementos al momento que se están procesando los datos. Dado un stream S de n elementos con frecuencias f_i, el problema de estimación de frecuencia consiste en procesar el stream de forma que para cualquier i se pude obtener un $\hat{f}_i$ tal que $\hat{f}_i \leq f_i \leq \hat{f}_i + \epsilon n$.

Podemos encontrar los *heavy hitters* resolviendo el problema de los elementos frecuentes. Los *heavy changers* pueden ser identificados realizando estimaciones de frecuencia de los diferentes elementos para ventanas de tiempo adyacentes. Dada la naturaleza del problema es necesario implementar soluciones eficientes en el costo de cómputo como en costo de memoria.

4. Métodos para determinar elementos frecuentes

4.1. Técnicas basadas en sketches

La forma clásica de procesar grandes volúmenes de datos y obtener información relevante asume que los datos están almacenados en algún soporte. Sin embargo, para ciertas aplicaciones este enfoque no es viable. Esta restricción es la que más importancia tuvo en relación al nacimiento de los métodos de *data streaming* e impulso el desarrollo de algoritmos livianos sumamente eficientes (sublineales en el uso de memoria), y estructura de datos probabilistas que pueden usarse para responder ciertas preguntas sobre los datos, de manera precisa y con una probabilidad razonablemente alta. Es decir, en lugar de almacenar la totalidad de los datos, se utiliza una representación comprimida (con cierta pérdida) de los mismos.

Los *sketch* son estructuras de datos que pueden ser pensadas como proyecciones lineales de los datos de entrada. Si representamos las entradas como vectores, estos pueden ser multiplicados por

una *matriz sketch*. El *vector sketch* resultado contiene la información suficiente para responder de forma aproximada ciertas preguntas sobre los datos. Por ejemplo, si codificamos los elementos del *stream* de datos como vectores cuya i-ésima entrada es su frecuencia f_i, el *sketch* es el producto de este vector y una matriz[4].

Los algoritmos basados en *sketches* resuelven el problema de estimación de frecuencia, pero necesitan información adicional para resolver el problema de los elementos frecuentes. Por esto, se suele aumentar la estructura de datos de los *sketch* con algún método de conteo para encontrar elementos frecuentes de manera eficiente (ver 4.2).

Antes de describir los algoritmos basados en *sketches* es conveniente introducir los filtros *Bloom* que son una versión simplificada de los primeros, de forma de ganar intuición acerca del funcionamientos de estas estructuras de datos probabilistas.

Filtros Bloom

El término *filtro Bloom* refiere a una estructura de datos compacta que se usa para representar conjuntos, pero que puede reportar falsos positivos cuando se consulta por la existencia de cierto elemento al conjunto[8]. La ventana principal de esta estructura de datos sobre las tradicionales es la eficiencia en el uso de memoria.

Para representar un conjunto de a lo sumo n elementos, un *filtro Bloom* clásico hace uso de un vector de m bits. Al comienzo todos los bits están en 0. Para insertar un elemento e en el filtro, k bits son actualizados en base a la evaluación de k funciones de hash independientes $h_1(e), h_2(e), \ldots, h_k(e)$. Para consultar si un elemento es parte del conjunto se deben calcular las k funciones de hash. Si todos los bits que indican las funciones de hash están en 1 es probable que el elemento sea parte del conjunto, de lo contrario es seguro que el elemento no forma parte del conjunto. Sea $c = m/n$, la elección óptima para k está dada por $k = int(\ln 2\ c) = int(\ln 2\ m/n)$ y la probabilidad de obtener un falso positivo (en promedio) está dada por $f_{std}(m, n, k) \approx (1 - e^{-kn/m})^k$[7]. Por ejemplo, para un universo de $n = 65536$ elementos y usando $c = 4$ bits por elemento tenemos que $f_{std} = 0{,}0174$.

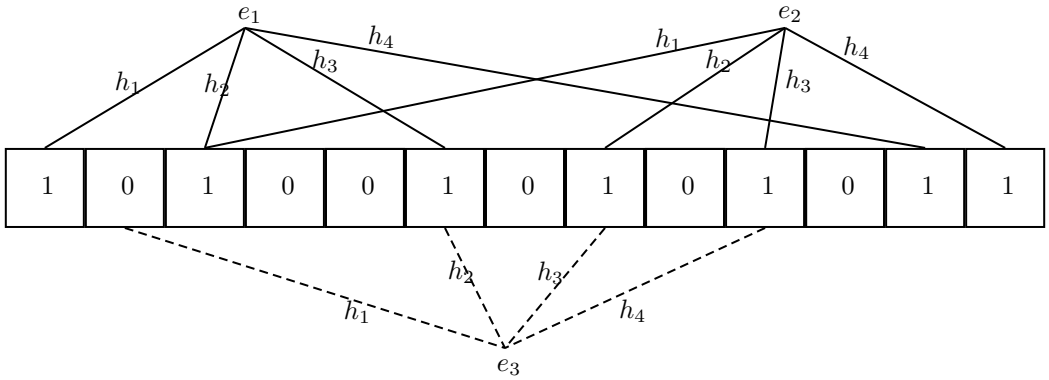

Figura 1: Filtro Bloom

La imagen muestra un filtro de Bloom luego que los elementos e_1 y e_2 fueron insertados. El elemento e_3 no se encuentra en el conjunto dado que al menos uno de los bits a los que apuntan las funciones de hash es 0. La pertenencia de un elemento e' al conjunto puede ser reportada erróneamente si las k funciones de hash $h_i(e')$ apuntan a bits del filtro con valor 1.

CountMin Sketch

Los *sketches* son menos conocidos que los filtros Bloom pero comparten ciertas similitudes. Son estructuras de datos que permiten sumarizar un *stream* de datos y pueden utilizarse para resolver el problema de los elementos frecuentes. Esto puede ser llevado a cabo usando menos

[4]Existen diferentes formas de definir la *matriz sketch*, cada una con aplicaciones particulares. Para conteo de eventos se usan familias de funciones de *hash* con ciertas propiedades para definir la proyección lineal.

espacio del que se utilizaría almacenando un contador por elemento, pero permitiendo que los contadores tengan cierto error en algunas ocasiones.

Sea $\mathbf{a}$ un vector de dimensión n cuyo estado en el tiempo t es $\mathbf{a}(t) = [a_1(t), a_2(t), \ldots, a_n(t)]$. Inicialmente $\mathbf{a}$ es el vector $\mathbf{0}$, es decir $a_i(0) = 0\ \forall i$. Las actualizaciones a los elementos de $\mathbf{a}$ se representan mediante un *stream* de tuplas. De forma general, la tupla (i_t, c_t) representa el t-ésimo elemento procesado:[5]

$$a_{i_t}(t) = a_{i_t}(t-1) + c_t$$
$$a_{i_{t'}}(t) = a_{i_{t'}}(t-1)\ \forall t' \neq t$$

El *sketch* COUNTMINT[9] se utiliza para sumarizar *streams* de datos. Consiste en un arreglo de $d \times w$ contadores y d funciones de hash independientes h_j con $1 < j < d$ que mapean un elemento i a un entero $z \in \{0, 1, \ldots w-1\}$. Por cada elemento procesado se calcula la posición del contador en el arreglo se mediante la tupla $(j, h_j(i))$ y actualizan d contadores (uno por cada fila) incrementando su contador en 1.

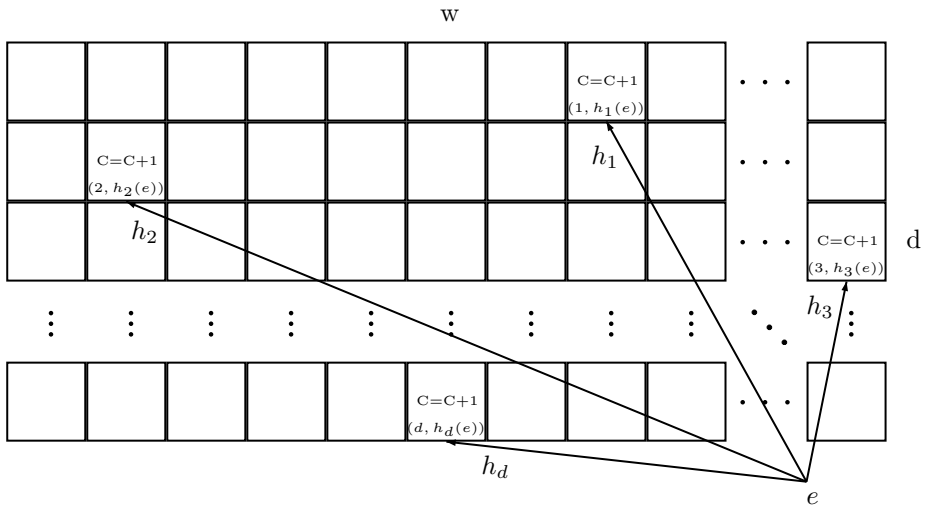

Figura 2: CountMin Sketch

Podemos estimar la frecuencia de ocurrencia a_i de un elemento i usando el *sketch* descrito anteriormente mediante la fórmula $\hat{f}_i = min_{1 \leq j \leq d}\{C[j, h_j(i)]\}$. Se puede demostrar que $a_i \leq \hat{f}_i$ y que $\hat{f}_i \leq a_i + \varepsilon||\mathbf{a}||_1$ con probabilidad mayor o igual a $1 - \delta$. Los parámetros ε y δ definen las dimensiones del *sketch* como $w = \lceil \frac{e}{\varepsilon} \rceil$ y $d = \lceil \ln \frac{1}{\delta} \rceil$.

Ejemplo Para procesar 10 millones de eventos y producir estimaciones con un error hacia arriba menor o igual a 100 con una probabilidad de al menos 0,99 necesitamos $\varepsilon||\mathbf{a}||_1 = 100$ y $\delta = 0{,}01$. En términos las dimensiones del *sketch* $w = 271829$ y $d = 5$

Finalmente, COUNTMIN es $O(\frac{1}{\epsilon} \log \frac{1}{\delta})$ en términos de memoria y para realizar operaciones de actualización es $O(\log \frac{1}{\delta})$

4.2. Técnicas basadas en conteo

Los algoritmos basados en técnicas de conteo mantienen contadores para un subconjunto del universo de elementos posibles T (ver 3.1). A continuación se describen algunos algoritmos de conteo existentes y sus características.

[5] Para conteo de elementos $c_t = 1$.

Majority

El problema de los elementos frecuentes fue estudiado por primera vez en la década del 80, y fue enunciado de esta manera:

> Supongamos una lista de n números, representando los "votos"de n procesadores en el resultado de cierto cálculo. Queremos determinar si hay un voto mayoritario y cual es ese voto.[10]

Para solucionar el problema, los autores desarrollaron un algoritmo de una pasada llamado MAJORITY. El pseudocódigo de MAJORITY se describe a continuación: se almacena el primer elemento y se inicializa un contador en 1. Por cada elemento subsecuente, si es el mismo que el elemento almacenado, se incrementa el contador en 1. Si el elemento es diferente y el contador es 0, entonces se reemplaza el elemento y se incrementa el contador en 1. De lo contrario, se decrementa el contador. Luego de procesar todos los elementos, el algoritmo garantiza que si hay un voto mayoritario, entonces este debe ser el almacenado por el algoritmo. En el peor caso, el algoritmo realiza $2n$ comparaciones. Usando un argumento de paridad podemos concluir que el resultado del algoritmo es el correcto: si por cada elemento que no es el mayoritario tomamos uno de los mayoritarios, al final van a quedar solo elementos del conjunto mayoritario.

Frequent

Una generalización del algoritmo MAJORITY es desarrolada en [11] la cual permite encontrar todos los elementos en el stream de datos que exceda $1/k$-ésimo del total de los elementos procesados. En lugar de guardar solo un elemento y un contador, FREQUENT almacena $k-1$ tuplas elemento-contador. Cada nuevo elemento es comparado contra todos los elementos almacenados por el algoritmo, y se incrementa el contador correspondiente si corresponde. Sino, si algún contador está en cero, se reemplaza el elemento y se inicializa el contador en 1. Si los contadores de los $k-1$ elementos están siendo utilizados, entonces todos son decrementados en 1. Este algoritmo no es exacto: al terminar de procesar el *stream* de datos, el contador asociado con cada elemento esta a lo sumo ϵn unidades por debajo del valor real si $k = 1/\epsilon$.[12]

FREQUENT puede resolver el problema de estimación de frecuencia desarrollado en la sección 3.2 con $\epsilon = 1/k$. En el peor caso, el algoritmo es $O(k)$ en memoria y realiza nk comparaciones y nk asignaciones a variables.

Dado el parámetro l que define el tamaño del subconjunto de elementos, se define un diccionario A de a lo sumo $l-1$ elementos. Supongamos un stream de datos $\{(x,1)\}$ (que puede ser infinito). Si el elemento x está presente en A entonces su valor es actualizado con v_x: $A[x]+=1$. Por el contrario, si la cantidad de elementos de A es menor a $l-1$ se agrega el contador $A[x]=v_x$. En caso que la cantidad de elementos de A sea igual a $l-1$, todos los valores de A son decrementados en 1: $A[x]-=1\ \forall x \in A$

Referencias

[1] D. Terry, D. Goldberg, D. Nichols y B. Oki, «Continuous Queries over Append-only Databases», *SIGMOD Rec.*, vol. 21, n.º 2, págs. 321-330, jun. de 1992, ISSN: 0163-5808. DOI: 10.1145/141484.130333. dirección: http://doi.acm.org/10.1145/141484.130333.

[2] S. Muthukrishnan, «Data Streams: Algorithms and Applications», *Found. Trends Theor. Comput. Sci.*, vol. 1, n.º 2, págs. 117-236, ago. de 2005, ISSN: 1551-305X. DOI: 10.1561/0400000002. dirección: http://dx.doi.org/10.1561/0400000002.

[3] G. Cormode y M. Hadjieleftheriou, «Finding Frequent Items in Data Streams», *Proc. VLDB Endow.*, vol. 1, n.º 2, págs. 1530-1541, ago. de 2008, ISSN: 2150-8097. DOI: 10.14778/1454159.1454225. dirección: http://dx.doi.org/10.14778/1454159.1454225.

[4] E. M. Hutchins, M. J. Cloppert y R. M. Amin, «Intelligence-driven computer network defense informed by analysis of adversary campaigns and intrusion kill chains», *Leading Issues in Information Warfare & Security Research*, vol. 1, pág. 80, 2011.

[5] D. Tong y V. Prasanna, «High Throughput Sketch Based Online Heavy Hitter Detection on FPGA», *SIGARCH Comput. Archit. News*, vol. 43, n.º 4, págs. 70-75, abr. de 2016, ISSN: 0163-5964. DOI: 10.1145/2927964.2927977. dirección: http://doi.acm.org/10.1145/2927964.2927977.

[6] G. Cormode y M. Hadjieleftheriou, «Methods for Finding Frequent Items in Data Streams», *The VLDB Journal*, vol. 19, n.º 1, págs. 3-20, feb. de 2010, ISSN: 1066-8888. DOI: 10.1007/s00778-009-0172-z. dirección: http://dx.doi.org/10.1007/s00778-009-0172-z.

[7] B. H. Bloom, «Space/Time Trade-offs in Hash Coding with Allowable Errors», *Commun. ACM*, vol. 13, n.º 7, págs. 422-426, jul. de 1970, ISSN: 0001-0782. DOI: 10.1145/362686.362692. dirección: http://doi.acm.org/10.1145/362686.362692.

[8] F. Putze, P. Sanders y J. Singler, «Cache-, Hash-, and Space-efficient Bloom Filters», *J. Exp. Algorithmics*, vol. 14, 4:4.4-4:4.18, ene. de 2010, ISSN: 1084-6654. DOI: 10.1145/1498698.1594230. dirección: http://doi.acm.org/10.1145/1498698.1594230.

[9] G. Cormode y S. Muthukrishnan, «An Improved Data Stream Summary: The Count-min Sketch and Its Applications», *J. Algorithms*, vol. 55, n.º 1, págs. 58-75, abr. de 2005, ISSN: 0196-6774. DOI: 10.1016/j.jalgor.2003.12.001. dirección: http://dx.doi.org/10.1016/j.jalgor.2003.12.001.

[10] L. J. Guibas, «Problems», *Journal of Algorithms*, vol. 2, n.º 2, págs. 208 -210, 1981, ISSN: 0196-6774. DOI: http://dx.doi.org/10.1016/0196-6774(81)90022-5. dirección: http://www.sciencedirect.com/science/article/pii/0196677481900225.

[11] R. M. Karp, S. Shenker y C. H. Papadimitriou, «A Simple Algorithm for Finding Frequent Elements in Streams and Bags», *ACM Trans. Database Syst.*, vol. 28, n.º 1, págs. 51-55, mar. de 2003, ISSN: 0362-5915. DOI: 10.1145/762471.762473. dirección: http://doi.acm.org/10.1145/762471.762473.

[12] E. Kranakis, P. Morin e Y. Tang, «Bounds for Frequency Estimation of Packet Streams», en *In SIROCCO*, 2003, págs. 33-42.

[13] A. Metwally, D. Agrawal y A. El Abbadi, «Efficient Computation of Frequent and Top-k Elements in Data Streams», en *Proceedings of the 10th International Conference on Database Theory*, ép. ICDT'05, Edinburgh, UK: Springer-Verlag, 2005, págs. 398-412, ISBN: 3-540-24288-0, 978-3-540-24288-8. DOI: 10.1007/978-3-540-30570-5_27. dirección: http://dx.doi.org/10.1007/978-3-540-30570-5_27.