МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ
РОССИЙСКОЙ ФЕДЕРАЦИИ
ТОМСКИЙ ГОСУДАРСТВЕННЫЙ УНИВЕРСИТЕТ

МАТЕРИАЛЫ

IX-й Международной научной конференции

«МАТЕМАТИЧЕСКОЕ И ПРОГРАММНОЕ ОБЕСПЕЧЕНИЕ ИНФОРМАЦИОННЫХ, ТЕХНИЧЕСКИХ И ЭКОНОМИЧЕСКИХ СИСТЕМ»

Томск, 26–28 мая 2022 г.

Под общей редакцией
кандидата технических наук И.С. Шмырина

Томск
Издательство Томского государственного университета
2022

III. ТЕОРИЯ МАССОВОГО ОБСЛУЖИВАНИЯ И ЕЕ ПРИЛОЖЕНИЯ

DOI: 10.17223/978-5-907572-27-0-2022-23

REVIEW OF MODERN METHODS OF NETWORK TRAFFIC ANALYSIS

Djeguede A.M.A.E., Salpagarov S.I.
Peoples Friendship University of Russia (RUDN University)
1042205202@rudn.ru, salpagarov-si@rudn.ru

Introduction

Nowadays information technologies gain more and more place in economics of all the countries. With the growing up of the percentage of digitalization of the almost all human activities in example education, banking system, energy providing, healthcare, national security and more, the need to ensure a high level of security and reliance also increases. The main way to reach that goal is to effectively monitor the network traffic. This work brings a brief overview of state-of-art methods and algorithms allowing to monitor, classify and also identify the harmfulness in network flow. In this sense, we explore the probabilistic data structures and string/pattern matching algorithms, that is widely use in monitoring of open networks. Analysis of encrypted traffic requires clairly another approach based on statistical processing of the network flow. We then expose the statistic-based method of representation the network in time series and the spectral methods associated to their analysis.

1. Probabilistic data structure

In order to save time searching for signatures and above all to minimize the volume of memory consumed, the use of probabilistic data structures has become widespread in the field of network traffic analysis. However, this expansion was subordinated to the phenomenon of false positives that we will elucidate in this section.

a. Bloom Filter

The Bloom Filter (BF) is a probabilistic data structure based on hash functions and a bit array initialized to 0. It was proposed by Burton Howard Bloom in 1970 in [1]. This filter is very efficient in terms of the amount of memory it occupies and is focused on the search of the occurrence of an element in a certain set. The absence of an element in the set is given unambiguously, while the occurrence of an element in the set is established with some probability. In this regard, a false-positive operation of the bloom filter is possible. False-negative operation is completely excluded. BF supports only two actions – insert and search.

Let m – the size of the bit array, k – the number of hash functions. Each hash function $h_i(x)$ generates a unique identifier in the interval $[1, m]$. $h_i(x)$ has a uniform distribution, which means $P(h_i(x) = p_i) = \frac{1}{m}$, $i = \overline{1, m}$. The probability that after adding an element x some p-th cell of the bit array will remain equal to 0, taking into account the independence of random variables $h_i(x)$

$$P(h_1(x) \neq p_1 \cap \ldots \cap h_k(x) \neq p_k) = P(h_i(x) \neq p_i)^k = \left(1 - \frac{1}{m}\right)^k .$$

After adding n elements we get

$$P\left(\forall i \in \{1 \ldots k\}, \forall j \in \{1 \ldots n\}, h_i\left(x_j\right) \neq p_i\right) = \left(1 - \frac{1}{m}\right)^{kn} \approx e^{-kn/m} .$$

A false positive is that after adding n elements, while searching for an element that is not included in the set of these elements, all positions $h_1(y), h_2(y), ..., h_k(y)$ in the bit array are set to 1. The probability of this event can be calculated by the following formula:

$$P = \left(1 - \left[1 - \frac{1}{m}\right]^{kn}\right)^k = \left(1 - e^{-kn/m}\right)^k .$$

Having determined the desired probability of a false positive, we can calculate the optimal size of the bit array as $m = -\frac{n \ln P}{(\ln 2)^2}$, and the optimal number of hash functions that minimize P can be calculated by the formula $k = \frac{m}{n} \ln 2$.

The Counting Bloom Filter (CBF) is a generalized version of the BF proposed in [2]. CBF is based on an array of counters. Each counter consists of groups of bits, which allows this data structure to support a delete operation in addition to insertion and search operations. When adding an element, the corresponding counters are incremented by one, and when removing, they are decremented by one. Repeating the same reasoning as in the previous case, we can derive the probability b that some arbitrary counter CBF had values l like $b\left(l, kn, \frac{1}{m}\right) = \binom{kn}{l}\left(\frac{1}{m}\right)^l \left(1 - \frac{1}{m}\right)^{kn-l}$. If you use CBF to determine the minimum number θ of occurrences of an element in a set, then you can consider the probability that a certain counter has a value $l < \theta$ as a sum $b\left(l, kn, \frac{1}{m}\right)$, $l = \overline{0, \theta - 1}$. Then the false positive probability formula for CBF can be calculated as follows $p_{\text{fp}}(\theta, k, n, m) = \left(1 - \sum_{l<\theta} b\left(l, kn, \frac{1}{m}\right)\right)^k$. The analysis of the probabilistic formula for false positive CBF carried out in the work [Analysis of CBF Used ....] and allowed us to derive the tabulated values of the optimal parameters, as well as their formula. Thus, for $\theta > 30$ it is possible to calculate the optimal number of hash functions $k_{\text{opt}} = \frac{m}{n}(0.20370 + 0.9176)$.

Since the introduction of the ability to remove elements from the counting bloom filter, the probability of a false negative increases as one removes from the set of false positive elements. This probability was estimated in [3]. The authors of this paper also proposed a variant of the CBF, called the MCBF multiselect bloom filter with a counter, which reduces the likelihood of false negatives of the CBF. The main idea is to use c groups of hash functions, each of which consists of k hash functions. When adding a new element x to the set X, first c groups of counter indices are calculated, from which only one group of indices will be selected to increment the corresponding counters. The final choice of an index group is based on minimizing the number of remaining bits in the c groups. This greedy approach was first proposed in [4] and [5].

b. Cuckoo Filter

The Cuckoo filter, like the Bloom filter, is a probabilistic data structure that allows you to check whether some element x is in the set X. In this case, the answer to this question is expressed in the format "possibly included / no, not included". This means there are a possibility of false positives. Cuckoo filter is based on a hash table using the Cuckoo hash function. Each element of the set has two potential indices i and j in CF. You can calculate these indices using the following formulas: $i = h_1(x) = \text{hash}(x)$, $j = h_2(x) = h_1(x) \oplus \text{hash}(\text{fingerprint}(x))$. This method is also called partial key hashing. To optimize the amount of memory occupied in practice, the fingerprint function is often used. It allows to replace *x* by the element that is significantly less in terms of the memory space occupancy.

Adding an element to the filter begins with calculating its signature using $f = \text{fingerprint}(x)$, then the index *i* is calculated using the formula $i = \text{hash}(x)$. The value *f* is added in the cell. If the cell *i* previously contained some value $f' = \text{fingerprint}(y)$ of the element *y*, then it is relocated to its alternative cell, the index of which is calculated by $j' = i \oplus \text{hash}(f')$. The value of cell j', if exists, is also relocated to an alternative cell, and so on until it is found free cell or the maximum allowed number of relocations is not reached, in which case the insertion algorithm fails. According to the above description, you will notice a decrease of insertion speed as the load coefficient filter is growing.

Searching and deleting algorithm is done in a simple way. Values *f*, *i* and *j* are calculated for the element *x*. If in any of the cells $i; j$ there is a signature value equal to *f*, then the search is completed successfully and, if necessary, this element can be deleted.

An analysis of the reasons for the appearance of false positive responses for the CF filter was made in [6]. The example considered in this work shows that if two different elements *x* and *y* have a double collation so that $f = \text{fingerprint}(x) = \text{fingerprint}(y)$ and $i = \text{hash}(x) = \text{hash}(y)$, it is obvious that these elements will have the same alternative addresses $j = i \oplus \text{hash}(f)$. Not hard to see, even after removing any of these elements from filter, the search for the removed element will continue to give a positive answer, which is actually a false positive answer.

An analysis of the collision probability In CF was also made in [6]. Let $\in$ – be the targeted percentage of false positives, *f* – the length of the fingerprint signature, α – the load factor of the hash table, *b* – the number of elements in each cell, *m* – the size of the hash table, *n* – the number of elements, and *C* – the average number of bits in each element. Let be given a set consisting of *q* elements and *x* – one of them such as $i_x = \text{hash}(x)$ and $t_x = \text{fingerprint}(x)$. The probability that, of the remaining $q-1$ elements, any *y* element had $t_x = \text{fingerprint}(y)$ equal $\frac{1}{2^f}$ and the probability that the first index of the element *y* had a value of either i_x or $i_x \oplus \text{hash}(t_x)$ equal to $\frac{1}{2^f}$, you can then calculate the collision probability with the following formula $\left(\frac{2}{m} \cdot \frac{1}{2^f}\right)^{q-1}$.

The aforementioned insertion algorithm will fail if the alternative cells overflow for some element *x*. Given the capacity of each cell, we can then say that failure will occur at $2b+1$ collisions. Of the *n* elements, the number of such collisions is C_n^{2b+1}. The expected probability of failure of the insertion algorithm is $C_n^{2b+1}\left(\frac{2}{m} \cdot \frac{1}{2^f}\right)^{(2b+1)-1} = \binom{n}{2b+1}\left(\frac{2}{2^f \cdot cn}\right)^{2b} = \Omega\left(\frac{n}{4^{bf}}\right)$.

Optimizing the amount of memory occupied by the CF filter is an important task in its practical implementation. You can estimate the memory footprint in terms of the average number used to represent each element of the filter. Let's call *C* this metric measured in bits. You can then define

$$C = \frac{\text{hash table size}}{\text{maximum number of stored elements}} = \frac{f \cdot m}{\alpha \cdot m} = \frac{f}{\alpha}\text{bits}.$$

The CF filter found its practical application in [7] in integration with the network traffic monitoring system Bro. Comparison of various CF filter implementations with traditional HashTable and HashSet data structures. Showed more than a hundredfold decrease in memory capacity. Methodological considerations for the selection of CF filter parameters, given by the authors Jan Grashofer and al. led to a decrease in the rate of false negatives. In this regard, the optimal size of the hash table was established $m = 2^k$, leading to the described decrease in the percentage of false positives.

c. Quotient filter

The quotient filter, is a probabilistic data structure that has come to replace the Bloom filter. As in the case of the Bloom filter, the Private filter is used to check for the occurrence of some element *x* in the set *X*. QF also supports insertion, search and deletion operations. The innovation of QF is the ability to combine with other filters and resize the filter without having to rehash the original keys. The quotient filter, like the Bloom and Cuckoo filters, has inherent false positive responses as a result of search activity.

The quotient filter data structure is based on a variant of the hash table containing only part of the hash key and three additional meta bits. The hash function generates a fingerprint of length *p*. The least significant *r* bits are called the remainder and the most significant $q = p - r$ bits are called quotient. The hash table then take 2^q places.

Let the key *K* have a hash $h(K)$ with quotient $h_q(K)$ and remainder $h_r(K)$. The cell at index h_q is canonical for remainder h_r. When you try to insert h_r into a cell with an index h_q, two collision situations are possible. Full collision and partial collision. Full collision – coincidences of quotient and remainders of two different ones, partial collision is coincidence of only quotients. The main reason for false positives of the filter is a complete collision, but when a partial collision occurs, it is possible to avoid false positives using additional metabits.

A run is a sequence of cells with the same quotients. The run whose first element occupies its canonical slot is the start of the cluster. Cluster – a sequence of runs ending in an empty cell or the beginning of another cluster.

Additional meta-bits are used to control collision in the hash table and have the following meanings. The busy bit is one if the cell is canonical to some key in the filter, optionally stored in that cell. Continuation bit – equal to one if the cell is occupied, but not the first elements in the run. The shift bit is one if the run is shifted relative to the canonical slot.

The search for any element *x* in the filter starts by calculating its hash $h(x)$ and separating the quotient h_q and the remainder h_r. The cell at index h_q is canonical for h_r. If all three additional bits had the value 0, we can unambiguously conclude that there is no element in the filter. Otherwise, you need to set the run and cluster to which the element belongs. To establish the beginning of the cluster, we will scan the left area from the canonical cell h_q, while starting the counter. Each time the busy bit is set to 1, the counter incremented by 1. When the shift bit set to 0 is encountered, the scan ends, the cluster start is found. Next, scan in the opposite direction. On encountering a continue bit set to 0, the counter is decremented by 1. The counter will have a value of 0 at the start of the quotient h_q run. One can then compare each remainder in this run with h_r. If such comparisons succeed, then we can conclude that the element is probably in the filter. Otherwise, we can unambiguously say that there is no such element in the filter.

The filter insertion algorithm starts by looking for an element in the filter. If there is such an element in the filter, then the insertion is completed, otherwise we insert the rest of the element in the current run in such a way as to keep this run sorted and shift all the rests to the right by one cell, starting from the occupied position by the inserted element. This updates all shift bits of the shifted elements. If a new element was added at the start of the run, then the offset at the start of the run is shifted to the right and its continue bit is set to 1. Shifting the remainder of the cell does not affect the busy bit of the cell, because it refers to the slot, not the remainder, contained in the cell.

An estimate of the probability of a false positive filter operation was carried out in [8]. Based on the observation that a false positive is only possible in QF if for any two different elements such that $x' \notin S$ and $x \in S$ (S – set of n elements, h – hash function), which is a complete collision, the authors Michel A. Bender et al. came up with a formula.

For a quotient filter, the average length of clusters plays an important role, since the insertion, search, and deletion algorithms have a certain time complexity, since these actions are performed within one cluster without affecting the entire filter. The cluster size can be estimated using the Chernoff estimate. Let $\alpha \in [0,1)$ and $K = (1+\varepsilon)\dfrac{\ln(m)}{\alpha - \ln(\alpha) - 1}$. The probability of existence of a cluster with a size greater than K, $\Pr(\text{the existence of a cluster with size} \geq K) < m^{-\varepsilon} \xrightarrow{m \to \infty} 0$, where $m = 2^q$ – is the number of cells in the filter, α – is the load factor. The mathematical expectation of the size of the clusters is then less than $1 \big/ 1 - \alpha e^{1-\alpha}$.

2. Pattern (String) matching algorithms

a. Aho – Corasick algorithm

In 1975 Alfred V. Aho and Magaret J. Corasick presented in their work [9] an algorithm for searching for a pattern in a text. This algorithm is based on building a state machine from a given set of keywords and then using this machine to search for occurrences of keywords in the incoming text in one pass.

The Aho – Corasick algorithm is divided into two parts: the FSM algorithm and the text search algorithm. The presented algorithms for constructing a finite automaton are partly inspired by the idea of the Knuth – Morris – Pratt algorithm and classical methods for constructing finite automata.

Let $K = \{\{y_1, y_2, \dots\ y_k\}$ – be a finite set of keywords and x – any text. The task is to find all occurrences of the keywords y_i in the text x, taking into account that the keywords may overlap. The program that implements the automaton proposed by the authors has three functions: transition function g, failure function on f, output function Output. The automaton consists of numbered states combined into a directed graph. The state with the number "0" is considered initial. The transition from state to state is carried out by a function g that takes two parameters – the current state and the input symbol. This function can be defined as $g(\text{state}_k, a_i) = \begin{Bmatrix} \text{state}_l \\ \text{fail} \end{Bmatrix}$. fail – means no transition from the state state_k by symbol a_i. The function $f(s) = s'$ allows you to go from state s to state s' in the absence of a transition from s by character a_i, that is, $g(s, a_i) = \text{fail}$. Some states can be final, which means that the function returns a list of keywords for which these states are final.

The transition function g can be built on the basis of a directed graph. The graph starts from state "0". Each keyword y_i is added to the graph from the initial state so that the sequence of arcs forming a new path in the graph is marked with the keyword symbols y_i. It is also important to note that it is added only if there is no path in advance in the column containing this word y_i. For the state that completes the new path in the graph, a list of recognized keywords is formed using the function output. It is important to note that for any symbols, the transition from the state "0" always exists and is different from fail, therefore, for all symbols with which the key does not start, transitions from "0" to "0" are automatically added.

A function f that provides an alternate transition when a failure occurs can be computed based on the function g. The depth $\text{depth}(s)$ of a certain state s is an integer number of transitions that must be made from the initial state "0" to reach s. For any states with depth 1, it is obvious that the alternative transition is the state "0", which is equivalent to $f(s) = 0$. The calculation of the function f value for states with depth d occurs iteratively using the values of the functions *f* and *g* for states with a smaller depth. Let *r* – any state with depth $d-1$, *f* can then be calculated for the *d* depth states as follows:

• If $g(r,a)=\text{fail}$ for all *a* – then stop.

• If $g(r,a)=s\neq\text{fail}$ for any character *a*:

– Announce state $\text{state}=f(r)$.

– Run zero or more times $\text{state}\leftarrow f(\text{state})$ until you get $g(\text{state},a)\neq\text{fail}$.

– You can then declare $f(s)=g(\text{state},a)=s'$.

– You can update the function $\text{output}(s)=\text{output}(s)\cup\text{output}(s')$.

You can optimize the number of failed transitions by introducing a function δ to replace *g* and *f*. The function δ then defines a deterministic finite automaton.

b. Booyer – Moore Algorithm

Robert S. Boyer and J. Strother Moore in their work [10] presented an algorithm for searching for a pattern in a text. The main idea of the BM algorithm is that when scanning the text from left to right, the comparison of characters occurs from right to left, that is, from the end of the searched pattern. The BM algorithm, through the use of two heuristics and tables associated with them, conducts jumps in the text. These heuristics set the rules for calculating the shift length, which eliminates the need to check all the characters in the text.

Let *T* – be the text and *P* – be the searched pattern. The stop character heuristic is based on finding the first non-matching character *T*[*i*] when comparing characters between text and pattern from right to left. If the symbol *T*[*i*] is found on the left at the position *P*[*j*] in the pattern, then the searched pattern is shifted to the right so that *T*[*i*] and *P*[*j*] match. Otherwise, if there is no such position on the left so that $T[i]=P[j]$, then the whole pattern is shifted to the right after the symbol *T*[*i*]. If the stop character is behind a character such that $T[i]=T[i+1]$, then the good suffix heuristic is applied.

The matching (good) suffix heuristic is based on finding a pattern suffix *S* that matches some substring *t* of text *T* when comparing characters from right to left. Let *P*[*j*] the character before the suffix *S* and *T*[*i*] be the character before the substring *t*. If there is a substring S' on the left in the pattern that matches *S* such, that the character *P*[*k*] in front of it differs from *P*[*j*], then the heuristic suggests shifting to the right by the minimum distance so that *P*[*k*] matches *T*[*i*]. Otherwise, if there are no more substrings on the left matching the substring *S*, then *P* is shifted by the minimum distance so that the prefix of the shifted pattern *P* matches the suffix of the substring *t*. And if such a shift is not possible, then shift the pattern *P* to the right by *m* positions, where *m* is the length *P*.

When found *P* in the text *T*, then the pattern *P* is shifted by the minimum distance so that the prefix of the shifted *P* matches the suffix of the found occurrence *P* in the text *T*. And if such a shift is not possible, then shift the pattern *P* to the right by *m* positions.

3. Statistical processing and spectral analysis of network flow

a. Entropy

Entropy is a major notion drawn from physics and mathematical statistics and information theory. According to information theory, entropy allow us to compute the amount of information in the system. In that case, this measure helps to process the network flow characteristic such as IP addresses, Port numbers into a sequence of numbers. Entropy is then computed on constant interval of time for instance every 5 or 10 minutes. p_i has to be interpreted as the probability of each unique characteristic in the aforementioned time interval. $H=-\sum_{i=1}^{n}p_i\log p_i$. We can then normalize the entropy using the following formula

$H_0 = \frac{H}{\log(n)} \to H_0 \in [0,1]$. Following is the algorithm generally used to detect anomaly in time series:

- Choose which attributes will be used to build entropy time series.
- Based on accumulated statistics during a large period, build time series for normalized Shannon entropy.
- For each time series on the its reference interval compute the variance.
- Then we are able to detect abrupt changes making short-time forecast and compare real traffic to the forecasted one. (Forecast can be done using ARIMA models, LSMT or more complex models).

b. Wavelet transform

Another way to process traffic into time series for its following analysis is perform wavelet transform. Traditional wavelet transform is based on Haar basis functions $\psi(x) = \begin{cases} 1, & 0 \le x < 1/2, \\ -1, & 1/2 \le x < 1 \end{cases}$ and scaling function $\phi(x) = 1$, $0 \le x \le 1$ to decompose input signal into piecewise constant signal. Novakov in [11] presents a spectral analysis technique for anomaly detection in network traffic.

Conclusion

This article after having briefly evoked the reasons, objectives and goal of the analysis of the network traffic, has reviewed a great number of techniques of analysis of the flows networks according to whether one is in an encrypted network or not. For each category of methods we recalled the theoretical foundations. For example, for probabilistic structures, we had underlined the negative impact of false positives, hence the need to be able to determine the factors minimizing the probability of false positives. Also it has been seen the contribution that classical text search algorithms can have in the detection of signatures. Then, we presented some methods for analyzing encrypted networks. As such, the role of the concept of entropy in the transformation of traffic with a view to its processing was presented. Finally, the contribution of spectral analysis in the detection of anomalies has been noted. Far from being a complete review, this article can however be improved and extended to other types of methods not listed in this work.

BIBLIOGRAPHY

1. *Bloom B.H.* Space/Time Trade-Offs in Hash Coding with Allowable Errors // Commun. ACM. 1970. – 7. – V. 13. – P. 422–426.
2. *Fan L., Cao P., Almeida J., Broder A.Z.* Summary cache: a scalable wide-area web cache sharing protocol // IEEE/ACM transactions on networking. – 2000. – V. 8. – P. 281–293.
3. *Guo D., Liu Y., Li X., Yang P.* False Negative Problem of Counting Bloom Filter // IEEE Transactions on Knowledge and Data Engineering. – 2010. – V. 22. – P. 651–664.
4. *Lumetta S., Mitzenmacher M.* Using the power of two choices to improve bloom filters. – Citeseer, 2006.
5. *Jimeno M., Christensen K., Roginsky A.* A power management proxy with a new best-of-n bloom filter design to reduce false positives // 2007 IEEE International Performance, Computing, and Communications Conference. – 2007.
6. *Fan B., Andersen D.G., Kaminsky M., Mitzenmacher M.D.* Cuckoo filter: Practically better than bloom // Proceedings of the 10th ACM International on Conference on emerging Networking Experiments and Technologies. – 2014.
7. *Grashofer J., Jacob F., Hartenstein H.* Towards application of cuckoo filters in network security monitoring // 14th International Conference on Network and Service Management (CNSM)/ – 2018.
8. Don't thrash: how to cache your hash on flash // 3rd Workshop on Hot Topics in Storage and File Systems (HotStorage 11), 201.
9. *Aho A.V., Corasick M.J.* Efficient string matching: an aid to bibliographic search // Communications of the ACM. – 1975. – V. 18. – P. 333–340.
10. *Boyer R.S., Moore J.S.* A Fast String Searching Algorithm // Commun. ACM. – 1977. – V. 20. – 10. – P. 762–772.
11. *Novakov S., Lung C.-H., Lambadaris I., Seddigh N.* Combining statistical and spectral analysis techniques in network traffic anomaly detection // Next Generation Networks and Services (NGNS). – 2012.