Computers, Materials & Continua CMC, vol.64, no.3, pp.1819-1844, 2020

High Accuracy Network Cardinalities Estimation by Step Sampling Revision on GPU

Jie Xu[1, *], Qun Wang[1], Yifan Wang[1] and Khan Asif [2]

Abstract: Host cardinality estimation is an important research field in network management and network security. The host cardinality estimation algorithm based on the linear estimator array is a common method. Existing algorithms do not take memory footprint into account when selecting the number of estimators used by each host. This paper analyzes the relationship between memory occupancy and estimation accuracy and compares the effects of different parameters on algorithm accuracy. The cardinality estimating algorithm is a kind of random algorithm, and there is a deviation between the estimated results and the actual cardinalities. The deviation is affected by some systematical factors, such as the random parameters inherent in linear estimator and the random functions used to map a host to different linear estimators. These random factors cannot be reduced by merging multiple estimators, and existing algorithms cannot remove the deviation caused by such factors. In this paper, we regard the estimation deviation as a random variable and proposed a sampling method, recorded as the linear estimator array step sampling algorithm (*L2S*), to reduce the influence of the random deviation. *L2S* improves the accuracy of the estimated cardinalities by evaluating and remove the expected value of random deviation. The cardinality estimation algorithm based on the estimator array is a computationally intensive algorithm, which takes a lot of time when processing high-speed network data in a serial environment. To solve this problem, a method is proposed to port the cardinality estimating algorithm based on the estimator array to the Graphics Processing Unit (GPU). Experiments on real-world high-speed network traffic show that *L2S* can reduce the absolute bias by more than 22% on average, and the extra time is less than 61 milliseconds on average.

Keywords: Network security, cardinality estimating, parallel computing, sampling revision.

1 Introduction

The Internet is one of the important infrastructures of the modern information society, and its security has become an important prerequisite for the national economy and social development [Bhuyan, Bhattacharyya and Kalita (2014)]. With the rapid development of

[1] Jiangsu Police Institute, Nanjing, 210031, China.

[2] Department of Computer Application, Crescent Institutes of Science and Technology, Chennai, 600048, India.

[*] Corresponding Author: Jie Xu. Email: xujieip@yeah.net.

Received: 24 March 2020; Accepted: 29 April 2020.

China's economy, the number of Internet users in China has increased rapidly. According to the statistics from the China Internet Network Information Center (CNNIC) [CNNIC (2019)], as of June 2019, the number of Internet users in China has reached 854 million. These network users will generate massive network traffic every day [Pacifici, Lehrieder and Dán (2016)]. Managing and analyzing such a high-bandwidth network is a daunting task. The rapid development of the Internet has also brought network security into a new era. Network and information security events, such as botnet [Vormayr, Zseby and Fabini (2017)], information leakage [Ciriani, Vimercati, Foresti et al. (2010)], industry network attacks [Xu, Ren, Wang et al. (2019)], are threatening the network's security.

Lots of researchers propose excellent and efficient algorithms to protect network security. Calculation and analysis of network traffic properties is an important foundation of network security and management. The cardinality of a host in the network [Xiao, Chen, Zhou et al. (2017)] is one of the important network attributes which has received much attention. Suppose there are two networks (*ANet* and *BNet*), and they communicate with each other through an edge router (*R*). Let *aip* represent a host in *ANet*, *bip* represent a host in *BNet.* When taking *R* as the traffic observation point, the cardinality of an *aip* in *ANe*t is the number of hosts in *BNet* which communicating with *aip* in a period through *R*; the cardinality of a *bip* in *BNe*t is the number of hosts in *ANet* which communicating with *bip* in a period through *R*. Without losing generality, the cardinality in this paper refers to the cardinality of *aip* in *ANet*. An IP address pair like <*aip*, *bip*> can be extracted from each packet passing through *R*, and cardinalities of different hosts could be calculated by scanning these IP address pairs stream extracted from packets stream. When *ANet* and *BNet* are high-speed networks, such as the Internet between two cities [Bianco, Bonald, Cuda et al. (2013)], it is difficult to accurately calculate the cardinality of each *aip* in real-time. To ensure the real-time performance of the calculation, estimating algorithms are proposed.

The cardinality estimation algorithm based on the estimator array is a common cardinality calculation method. This method calculates the cardinality of different hosts by using a fixed number of estimators. Each estimator is used by several *aip* to reduce memory consumption, and each *aip* uses several estimators to improve estimation accuracy. Existing algorithms do not take memory footprint into account when selecting the number of estimators used by each host. Based on the relationship between memory occupancy and estimated accuracy, this paper analyzes the influence of different row numbers of estimator array. The cardinality estimation algorithm based on the estimator array is a computationally intensive algorithm, which takes a lot of computational time to run in real-time when processing high-speed network data in a serial environment. In this paper, a method is proposed to port the cardinality estimation algorithm based on the estimator array to the Graphics Processing Unit (GPU) [Mittal (2017)], and the real-time cardinality estimation on the high-speed network is realized. GPU was originally designed for image processing, and now it is widely used in the field of parallel computing and artificial intelligence. The main contributions of this article are as follows:

(1) Analyzed the influence of the number of linear estimator array's rows on the estimation results under the condition of fixed memory.

(2) Proposed a novel algorithm, the linear estimator array step sampling algorithm (*L2S*),

to improve the accuracy of estimated cardinalities by deviation sampling.

(3) A framework for deploying a cardinality estimation algorithm based on a linear estimator array on the GPU is proposed, and real-time cardinality estimation of the high-speed network is realized.

(4) Experiments on real-world high-speed network traffic are presented to demonstrate that *L2S* can improve the accuracy greatly with little time consumption.

This paper is organized as follows. In Section 2, the existing works are introduced. In Section 3, the algorithm of cardinality estimation based on the array of estimators is introduced. The method to analyze the parameters of estimator array is given by calculating the relationship between the number of estimators used by each host and the number of distinct IP address pairs. Section 4 describes how the linear estimator array step sampling algorithm (*L2S*) works to improve accuracy by cardinality sampling. In Section 5 of this paper, from the perspective of parallel computing, a method of transplanting the cardinality estimation algorithm based on estimator array to GPU is proposed. Section 6 gives the results and analysis of experiments, and Section 7 summarizes this paper.

2 Related work

Host cardinality is an important network attribute and used in many applications, such as Distributed Denial of Service (DDoS) detection [Zargar, Joshi and Tipper (2013)], Peer to Peer (P2P) management [Zhuang and Chang (2019)], network security [Xie, Yan, Yao et al. (2019)] and so on. For low-speed network or without real-time processing requirement, the red-black tree can be used to store all IP address pairs, and accurately calculate the cardinality of each host. This method can accurately get the cardinality of each host, and there is no error in the results. This algorithm is called the precise algorithm. For high-speed networks or embedded devices, the precise algorithm has the following disadvantages. First, large memory consumption. For each corresponding *bip* of each *aip*, the precise algorithm needs to save them in memory. Because the high-speed network contains a large number of hosts, the precise algorithm needs a large amount of memory to store this information. If there is a network attack, such as the flood attack which forges the source address, the number of IP address pairs of some *aip* will grow rapidly and occupy too much memory. Second, memory access is frequent. For each IP address pair, the precise algorithm needs to find out whether the corresponding *bip* has ever appeared in memory. It is an I/O-intensive algorithm, and the speed of memory access will limit the speed of the precise algorithm. The traffic of high-speed network is a kind of big data [Wang, Wang, Sherratt et al. (2020)], and it is inefficient to store all traffic when running in real-time. Hence, each IP address pair can only be scanned once when running online.

Therefore, the precise algorithm is not suitable for high-speed networks. To reduce memory usage and improve the speed of the algorithm, many scholars proposed the host cardinality estimation algorithm based on data sharing structures. This kind of algorithm uses fixed-size memory to estimate different hosts' cardinalities. In this paper, the algorithm that can estimate cardinalities of different hosts at the same time is called the

cardinality estimation algorithm. The algorithm that can only estimate the cardinality of a single host is an estimator.

The estimator is the basis of the cardinality estimation algorithm. Each estimator can be regarded as a set of counters. Commonly used estimators include PCSA [Qian, Ngan, Liu et al. (2011)], HyperLogLog [Cohen and Nezri (2019)] and Linear Estimator (*LE*) [Tarkoma, Rothenberg and Lagerspetz (2012)]. Among them, the *LE* algorithm, also called linear estimator, is widely used in various cardinality estimation algorithms because of its high accuracy and simple operation. *LE* uses g counters to record and estimate the cardinality, and every counter in *LE* is a bit. Let LE^i represent the *i*th bit in *LE*. When initialized, each bit is 0. When scanning the corresponding *bip*, each *bip* is randomly mapped to a bit and the value of that bit is set to 1. After scanning all corresponding *bip*, *LE* uses the following formula to estimate the cardinality. Where n_0 represents the number of bits with a value of 0.

$$\text{Est} = -g * \log\left(\frac{n_0}{g}\right)$$

The cardinality estimation algorithm uses the estimator to estimate the cardinality of each host. From the use of the estimator, the cardinality estimation algorithm can be divided into two categories, namely, the cardinality estimation algorithm based on counter sharing [Shin, Im and Yoon (2014)] and the cardinality estimation algorithm based on estimator sharing [Liu, Qu, Gong et al. (2016)]. The cardinality estimation algorithm based on counter sharing assigns a virtual estimator to each host. Each counter of the virtual estimator is mapped to a counter in the counter pool, i.e., each counter will be shared by virtual estimators of multiple hosts for cardinality estimation. The cardinality estimation algorithm based on counter sharing uses a fixed number of estimators to estimate cardinalities of different hosts. In the cardinality estimation algorithm based on estimator sharing, each estimator will be used for cardinality estimation of multiple hosts, and each host will also use multiple estimators to estimate cardinality. The cardinality estimation algorithm based on estimator sharing can not only reduce the estimation error by using multiple estimators jointly but also recover the candidate super points by reasonably setting the mapping mode from each host to the estimator. In this paper, the cardinality estimation algorithm based on estimator sharing using *LE* as the estimator is studied, a novel high accuracy algorithm by cardinality sampling is proposed, and a real-time operation mode of transplanting the algorithm to GPU is given.

3 LE array cardinality estimating

The *LE* can be used to estimate the cardinality of a host. The number of bits contained in *LE* determines the accuracy of its estimation results and the maximum cardinality it can estimate. For example, when the number of bits in *LE* is 2^9, according to the estimation formula of *LE*, the maximum estimation value that *LE* can give is 3194. When the cardinality of a host is greater than 3194, *LE* needs to use more bits to accurately estimate the cardinality of the host. To get high estimation accuracy, we need to allocate enough bits for *LE*, because we don't know the exact value of a host before estimating its cardinality. The more bits *LE* contains, the more memory it takes up.

Although it is better to use *LE* with a large number of bits for the high-cardinality host, it will waste a lot of memory to estimate the cardinality of small cardinality hosts using *LE* with the same number of bits. At the same time, the high-speed network will contain a large number of hosts, and most of them have small cardinalities. Therefore, allocating a single *LE* to each host will waste a lot of memory and reduce the efficiency of the algorithm.

To improve memory usage, each *LE* is used for cardinality estimation of multiple hosts. However, this will lead to an overestimation of the cardinality. To reduce the overestimation caused by *LE* sharing, each host also uses multiple *LE* for cardinality estimation at the same time. Using *LE* array to estimate cardinality is proposed under this principle.

3.1 Process of cardinality estimating

The *LE* matrix composed of *r* row and *c* column is called *LE* array, which is recorded as Linear estimator Array (*LA*). Fig. 1 shows the structure of *LA*. Each *LE* in *LA* will be shared by multiple hosts. Among them, *r* determines how many *LE* each host will use for cardinality estimation, and *c* determines how many hosts each *LE* will be used for cardinality estimation on average. The algorithm that uses *LA* for cardinality estimation is called Linear estimator Array Algorithm (*LAA*). In other words, *LA* is the main data structure used in *LAA*.

For high-speed networks, *LAA* can estimate the cardinality of each host in the network. Make *aip* a host in a high-speed network. *LAA* takes a *LE* from each line of *LA* to record and estimate the cardinality of *aip*. Therefore, for *aip*, *r* different *LE* are corresponding to it in *LA*. These *LE* are called the association *LE* set of *aip* and are recorded as *RLE(aip)*. *RLE(aip)* is defined as follows.

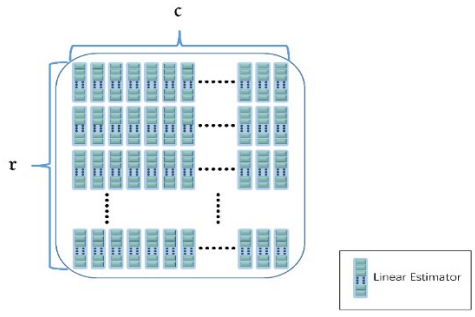

Figure 1: Linear Estimator array used to estimate different hosts' cardinalities

Definition 1. (Association *LE*). For a given host *aip*, the *LE* set used to record and estimate its cardinality in all rows of *LA* is the association *LE* set of *aip*, which is recorded as $RLE(aip) = \{LE_0(aip), LE_1(aip), LE_2(aip), \ldots, LE_i(aip), \ldots, LE_{r-1}(aip)\}$. $LE_i(aip)$ is the *LE* used to estimate the cardinality of *aip* in the *i*th row of *LA*.

LAA consists of two processes: scanning IP address pairs and estimating host cardinality. For each IP address pair in a time window, *LAA* will update its *LE*.

Before scanning IP pair, *LAA* first allocates the space needed by *LA* in memory and then initializes *LA*, i.e., all bits of each *LE* in *LA* are reset to 0. After *LA* initialization, *LAA* starts to scan each IP address pair. For IP address pair<*aip*, *bip*>, *LAA* selects *r LE* from *LA* according to *aip* to form *RLE(aip)*, and updates these *LE* with *bip*, to record the cardinality of *aip*. After scanning all IP address pairs in the time window, the cardinality information of each host is saved in *LA*. For a given host *aip*, *LAA* can estimate its cardinality according to *LA*.

Because *LE* in *LA* will be shared by multiple hosts, to reduce the impact of *LE* sharing, *LAA* uses the result of merging multiple *LE* to estimate the cardinality. Let *ULE(aip)* represent the union of all *LE* in *RLE(aip)*. For the host *aip*, before estimating its cardinality, *LAA* first sets each bit in the *ULE(aip)* to 1 and then merges the *ULE(aip)* with each *LE* in the *RLE(aip)* in the way of "bit AND" ("&"). *LAA* estimates the cardinality of *aip* based on *ULE(aip)*.

By merging *LE* in *RLE(aip)*, we can reduce the impact of *LE* sharing. The larger the *c* is, the fewer hosts are corresponding to each *LE*. The larger the *r* is, the fewer bits set by other hosts will be included in the union *LE*. That is to say, the larger *c* and *r*, the higher the accuracy of estimation. But the larger the *c* and *r*, the larger the memory occupied by *LA*. When the memory size of *LA* is determined, will the changes of *r* and *c* affect the accuracy of cardinality estimation? These are discussed in the next section.

3.2 Influence of row number

The number of rows in *LA* will affect the accuracy of cardinality estimation. The actual calculation is not that the bigger the *r*, the better. The larger the *r* is, the more *LE* will be updated when each IP address pair is processed. In addition to increasing the calculation time, it may also reduce the accuracy of the algorithm.

When the memory size occupied by *LA* is constant, reasonably setting of *r* and *c* can improve the accuracy of the algorithm. To facilitate the discussion, the relevant symbols and parameters are given first. Let *V* represent the total number of *LE* in *LA*, i.e., $V=r\times c$; *N* represent the number of different IP address pairs in a time window. For a stable network, the fluctuation range of *N* should be relatively stable in a certain period. This value can be obtained from the statistical observation of the past flow and can be reset when the observed value changes beyond the acceptable range. Therefore, *N* is assumed to be constant in the following discussion.

Let *RLE* represent the set composed of one *LE* selected from each row of *LA* randomly, and *ULE* represent the union *LE* of *RLE* by bitwise "*AND*" operation. Let *PSU* indicate the probability that a bit in the *ULE* is set to 1. The smaller the *PSU*, the closer the union *LE* of each *aip* is to the *LE* it uses exclusively. Therefore, *PSU* can be used as an analysis measure. It is proved that, with certain *V*, reasonably setting *r* can reduce *PSU* and reduce the error caused by *LE* sharing.

Lemma 1 When there are *N* different IP address pairs in a time window, the probability that a bit in the *ULE* is set to 1 is $\boldsymbol{PSU} = \left(\mathbf{1} - \left(\mathbf{1} - \frac{\mathbf{1}}{\boldsymbol{g}}\right)^{\frac{\boldsymbol{N}}{\boldsymbol{c}}}\right)^{\boldsymbol{r}}$.

Proof. Let n_1 be the number of IP address pairs corresponding to each *LE* in *LA*. In *LA*, each row is updated by *N* different IP address pairs. When the IP address pair is uniformly mapped to different *LE* by hash function [Huang, Yang and Zheng (2018)], $n_1 = \frac{N}{c}$. The probability that each bit in *LE* is set by an IP address pair is $\frac{1}{g}$, then the probability that a bit is not set by n_1 IP address pairs is $\left(1 - \frac{1}{g}\right)^{n_1}$. In the merged *ULE*, only when the bits of the corresponding position in the row are all 1, the merged bits will be 1. Hence,

$$PSU = \left(1 - \left(1 - \frac{1}{g}\right)^{\frac{N}{c}}\right)^{r}. \tag{1}$$

Theorem 1 If the number of *LE* in *LA* is *V*, i.e., $r \times c = V$, and there are *N* different IP address pairs in a time window, then *PSU* gets the minimum value when $\boldsymbol{r} = \frac{\boldsymbol{-V * log\ (2)}}{\boldsymbol{N * log\ (1 - \frac{1}{g})}}$.

Proof. When *V* and *N* is fixed, $PSU = (1 - \left(1 - \frac{1}{g}\right)^{\frac{r*N}{V}})^{r}$. Let $\mathrm{L} = \frac{N}{V}$, $\mathrm{G} = 1 - \frac{1}{g}$, the derivative of *PSU* is $\frac{dPSU}{dr} = (1 - G^{r*L})^{r} * (\log(1 - G^{r*L}) - r * L * G^{r*L} * \log(G) * (1 - G^{r*L})^{-1})$.

When $\frac{dPSU}{dr} = 0$, *PSU* reaches its minimal value. Because of $1 - G^{r*L} > 0$, when $\frac{dPSU}{dr} = 0$, we have the following equation.

$$\log(1 - G^{r*L}) = r * L * G^{r*L} * \log(G) * (1 - G^{r*L})^{-1} \tag{2}$$

Solving the Eq. (2), we have $r = \frac{-V * \log(2)}{N * \ln(1 - \frac{1}{g})}$.

For *LAA*, setting the *r* reasonably according to the number of IP address pairs can improve its accuracy. It can be seen from Theorem 1 that when the total number of *LE* is fixed, *PSU* does not always decrease with the increase of *r*. When the *r* is larger than a threshold, *PSU* will increase with the increase of *r*. At the same time, the larger *r* is, the more *LE* are needed to be merged when calculating the *ULE*. This increases the time taken by the algorithm. Hence, we should consider *PSU* and calculation time when setting *r*.

4 Step sampling revision

LAA uses several *LE* to calculate the cardinality of each *aip* to reduce the error caused by *LE* sharing. However, the error of *LAA* is also affected by some other factors, such as the random parameters inherent in *LE* and the random functions used to map *aip* to *LE* in *LA*. These random factors cannot be reduced by merging multiple *LE*. Therefore, we need to

use other methods to eliminate the impact of these random factors, to further improve the accuracy of cardinality estimation. To eliminate these errors, we need to estimate the errors first. The method of sampling is one of the best ways to estimate a random variable. According to the characteristics of the cardinality estimation process, a sampling method called step sampling is proposed to improve the accuracy of *LAA* estimation results. To distinguish the sampling revision algorithm from *LAA*, this paper calls the algorithm, which uses step sampling to adjust *LAA* estimation results, *L2S* algorithm (*LAA* with Step Sampling). Compared with *LAA*, *L2S* has two additional processes: cardinality sampling and estimation revision. Let's introduce the two processes respectively.

4.1 Cardinality sampling

After scanning all IP address pairs in a time window, *LAA* will estimate the cardinality of each *aip*. Let aip^n denote a host with cardinality n in the current time window in *ANet*, and AIP^n denote a set of *aip* with cardinality n in the current time window. For an *aip* with a cardinality of n, there will be some deviation between its estimated value n' and n due to the systematic error of *LE* and the random functions used by *LAA*. Let Δ_n represent this deviation and $\Delta_n = n' - n$.

Different aip^n have different n' and Δ_n. Therefore, n' and Δ_n could be regarded as two random variables. If we can estimate the expected value of Δ_n ($E(\Delta_n)$), and use $n' - E(\Delta_n)$ to rectify n', we can get an estimated value closer to n than n'. However, *LAA* itself cannot estimate $E(\Delta_n)$. This is because *LAA* doesn't know the actual cardinality of *aip* when running in real-time, and there may be too little hosts in AIP^n to estimate $E(\Delta_n)$ correctly.

Therefore, this paper presents a method to estimate $E(\Delta_n)$ by sampling. When estimating $E(\Delta_n)$, it is necessary to obtain enough values of Δ_n. When n is determined, the calculation of Δ_n is to calculate n'. Hence, the sampling method in this paper will first construct a sample set (SIP^n) consisting of several hosts whose cardinalities are n, $SIP^n = \{sip_1^n, sip_2^n, sip_3^n, \cdots\}$. Each sip_i^n in SIP^n has n different *bip*. The set of these n different *bip* is called *bip* vector. Different sampled host has different *bip* vector.

After *LAA* scanning all IP address pairs in the current time window, we use *LAA* to estimate the cardinality of the hosts in SIP^n. Each $sip_i^n \in SIP^n$ could be regarded as an *aip*. *LAA* selects *r* different *LE* from *LA* and merges them by bitwise "*AND*" operation to acquire the union *LE* (*ULE*). Then each *bip* in the *bip* vector of sip_i^n is used as the input data to update *ULE*. After scanning *bip* vector, the estimated cardinality of sip_i^n is calculated from *ULE* by Eq. (1).

Estimating the cardinality of each host in SIP^n according to the above method, we will obtain a set of n'. Since n is constant, $E(\Delta_n) = E(n') - n$. When estimating the value of $E(n')$, we can use the average or median of the estimated values of this sample set.

To let the sampling results better evaluate $E(\Delta_n)$, each host in SIP^n has n randomly selected bip, and the IP address sip_i of sip_i^n is also randomly generated. The number of hosts in SIP^n, i.e., the number of samples, will also affect the accuracy of the estimated value of $E(\Delta_n)$. Use β to represent the number of hosts in SIP^n. According to the

theorem of large numbers, the larger β is, the more accurate the calculation of E (Δ_n) is, but the greater the cost is. Therefore, the value of β needs to be determined flexibly according to specific application scenarios.

Estimating the cardinality of each host in SIP^n is processed after *LAA* scanning all IP address pairs in the current time window. To avoid introducing additional errors, it not supposed to modify *LE* in *LA* when estimating E (n'). Therefore, it is necessary to find out *RLE* (sip_i), merge each *LE* in *RLE* (sip_i) to get the *ULE*. Then update the *ULE* with those corresponding *bip* of sip_i, and use the updated *ULE* to estimate the cardinality of sip_i. In this way, the influence of *bip* of sip_i on *LA* could be avoided.

4.2 Estimation revision

The cardinalities of different *aip* vary from one to thousands. If every cardinality is sampled, lots of resources will be wasted. Therefore, this paper proposes a method, called step sampling, to estimate the deviation. The step sampling method first sets a lower cardinality limit *RC0* and an upper limit *RC1*. Then, starting from *RC0*, estimate the deviation of cardinality every other certain distance until the cardinality to be estimated is greater than or equal to *RC1*. The distance here is the estimated step recorded as α. *RC0* is an integer greater than or equal to 1, and *RC1* is an integer greater than *RC0*. The purpose of setting *RC0* and *RC1* is to improve the efficiency of sampling, avoid wasting resources to estimate those cardinality deviations that may exceed the maximum cardinality of aip. *RC0* can be determined based on the minimum cardinality to be calculated, while *RC1* can be determined based on the maximum cardinality of *aip* estimated by *LAA*.

When α and β are determined, the cardinality to be sampled and the number of samples of each cardinality are also determined. If n_i is the cardinality of the ith sample, then $n_i = RC0 + (i-1) * \alpha$ and $n_i \leq RC1$. According to the method in the previous subsection, E(n_i') is estimated by β samples of n_i.

The estimated value of *aip* can be adjusted according to n_i and E (n_i'). Assuming that the estimated value of an *aip* is n', rearrange E (n_i') in ascending order. If there are two integers j and k such that E $(n_j') \leq n_i' \leq$ E(n_k'), and E (n_j') and E (n_k') are in two adjacent positions after sorting, then n_i' can be adjusted according to the following formula. n_i'' is the revised value.

$$n_i'' = n_j + \frac{(n_k - n_j) * (\mathrm{n}' - \mathrm{E}(n_j'))}{\mathrm{E}(n_k') - \mathrm{E}(n_j')} \tag{3}$$

Eq. (3) uses linear relation to modify n_i' according to sample distribution. The estimated value of each *aip* can be adjusted according to Eq. (3) to reduce the random error of *LAA* and improve the accuracy of estimation.

α and β determine the sample capacity. Generally speaking, the larger the sample capacity is, the smaller the error is, but the more resources are consumed. For a cardinality n, when the confidence (δ), the allowable error (ε) and the total standard deviation (σ) of the estimated value are given, according to the statistical principle, the

sample capacity (μ) for n is $\mu \geq \left(\frac{Z_{\delta/2}*\sigma}{\varepsilon}\right)^2$. $Z_{\delta/2}$ is the bilateral critical value of the standard normal distribution at the level of δ. The sample capacity of α cardinalities is $\alpha * \mu$. Because *L2S* estimates the error every α cardinalities, $\beta = \alpha * \mu$. The value of α can be adjusted according to the historical operating results.

L2S can run in parallel to improve sampling speed. In the next section, we will discuss how to run *L2S* on GPU.

5 Translate *LAA* and *L2S* on GPU

In actual operation, *LAA* needs to be able to estimate the number of connection pairs of the host in real-time. Real-time running means that in a time window, the cumulative time of *LAA* scanning IP address pair and the time of estimating the host cardinality are less than or equal to the length of the time window. If *LAA* can not run in real-time, packet loss will occur, thus reducing the accuracy of detection results. For the high-speed network, millions of packets pass through the network boundary every second, and there are a large number of hosts in the high-speed network. Each packet corresponds to an IP address pair. *LAA* needs to scan these packets in real-time and estimate the cardinality of different hosts. Although the time complexity of accessing memory when *LAA* scans the IP address pair is O(1), it needs a lot of computation when mapping the host to different *LE* and the peer IP to a bit in *LE*. It can be seen that *LAA* is a computation-intensive algorithm. When running on the CPU, *LAA* scans each IP address pair serially. However, due to the limited computing power of CPU, *LAA* cannot process high-speed network data in real-time. Therefore, we need to use a platform with more computing resources to improve the running speed of *LAA*. The heterogeneous computing platform based on GPU and CPU is an ideal parallel computing platform for processing computing-intensive tasks.

Parallel computing is a common way to improve the speed of an algorithm. According to Bernstein's theorem, for the computation-intensive tasks without data access conflict, porting them to the parallel environment can obtain a high speedup ratio. The process of scanning IP address pairs and estimating host cardinality of *LAA* can be completed on GPU. *LAA* does not need to read data from *LA* when scanning IP address pairs, hence there will be no read-write conflict in Bernstein's theorem. When *LAA* updates *LA*, it will only set some bits in *LA* to 1. So the process of scanning IP address pairs of *LAA* can run in parallel. *LAA* does not need to write to *LA* when estimating the host cardinality, hence it conforms to Bernstein's theorem and can run in parallel.

GPU is a kind of parallel processor with a large number of computing units [Silber-Chaussumier, Muller and Habel (2013)], which is suitable for processing computing-intensive tasks without data conflicts. Therefore, transplanting *LAA* to GPU is an ideal way to improve the running speed of the algorithm. According to the characteristics of *LAA* and GPU, this paper proposes a way to transplant *LAA* to GPU.

5.1 Scanning IP pair on GPU

When *LAA* scans the IP address pairs on the CPU, it will scan each IP address pair in turn, i.e., every packet passing through *R* will extract the IP address pair and update *LA*.

However, GPU can only access graphic memory, can't directly access CPU memory. Therefore, to enable *LAA* to scan IP address pairs on GPU, it is necessary to store *LA* on GPU's graphic memory and copy IP address pairs to video memory. *LA* can be initialized on GPU memory at the beginning of the algorithm. However, the network packets will come continuously in the time window. If the IP address pair is copied to the graphic memory one by one, it will waste a lot of computing time. Therefore, this paper allocates a buffer of the same size on CPU and GPU to store IP address pairs. When the buffer on the CPU side is full or reaches the time window boundary, copy the IP address pair buffer on the CPU side to the IP address pair buffer on the GPU side, and start GPU threads to process IP address pairs. The number of IP address pairs in the buffer is the number of GPU threads to start. From the previous analysis, we can see that there is no overwriting problem between different GPU threads, hence each GPU thread can accurately scan the IP address pair.

5.2 Estimating host cardinality on GPU

When estimating each host's cardinality, *LAA* reads data from the *LA* without writing data to the *LA*. After GPU scanning all IP address pairs in the current time window, *LA* will contain the cardinality information of the host. At this time, different GPU threads can be used to estimate the cardinality of different hosts.

When estimating the cardinality of each host, a temporary *LE* is needed to store the union *LE*. Therefore, when using GPU to estimate the host cardinality, we need to allocate a temporary *LE* for each thread. To improve the running efficiency of the algorithm on GPU, a fixed number of *LE* is allocated before estimating the cardinalities of hosts. When estimating the host cardinality, each GPU thread uses a temporary *LE* separately. Due to the limited number of threads that GPU can start at the same time, and the excessive allocation of temporary *LE* will also occupy a large amount of memory space, this paper groups the hosts that need to estimate the cardinality. For each group of hosts, start the same number of GPU threads for cardinality estimation. This not only improves the efficiency of operation but also reduces the occupation of graphic memory.

5.3 Step sampling on GPU

Cardinality sampling and estimation revision are unique to *L2S*. Estimation revision only needs to modify the estimated cardinality of each *aip* according to Eq. (3). However, due to the influence of sampling parameters α and β, cardinality sampling can calculate a large number of cardinalities of sampling hosts, and the amount of calculation is far more than the amount of calculation needed for estimation revision. The speed of *L2S* can be improved by transplanting the cardinality sampling process to GPU. The cardinality sampling process does not modify the *LA*, so it can run on GPU. Fig. 2 illustrates how to obtain the estimated cardinalities of sampled hosts on GPU.

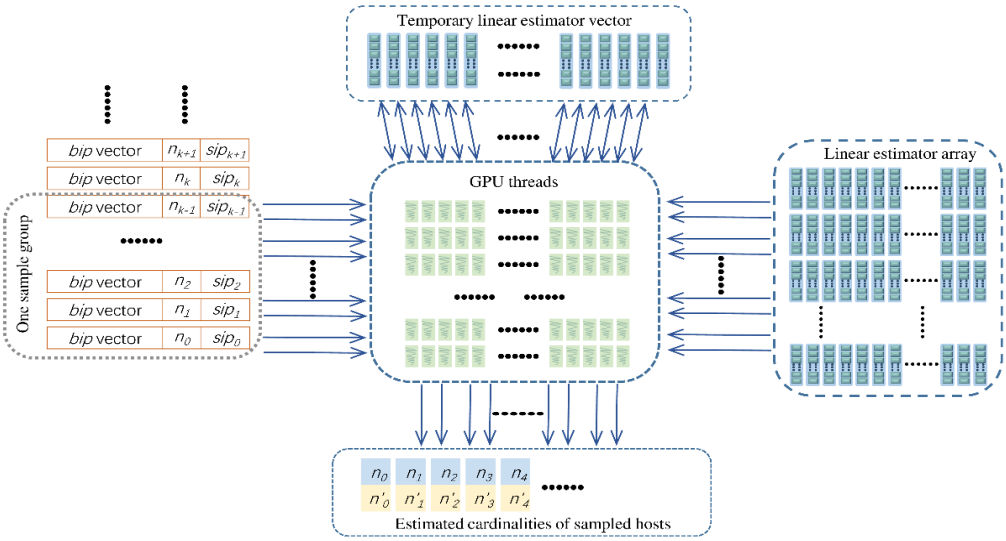

Figure 2: Parallel estimating the cardinalities of sampled hosts

When estimating the cardinality of a sampled host, we need to know the IP address *sip*, the cardinality n and the *bip* vector. In Fig. 2, the *sip*, n, and *bip* vector form a row of sampling data. Because the number of sampled hosts may be larger than the number of threads that GPU can start at one time, the sampling data is divided into different groups. Each group contains k sampled hosts. GPU processes the sampling data group by group. GPU starts k threads at a time, and one thread processes a row of sampling data. k can be determined according to the memory capacity of the GPU and the physical computing unit.

When estimating the cardinality of each sampling host, we need a temporary *LE* to store the merged *LE*. Therefore, we also allocate k *LE* on the GPU, that is, the temporary *LE* vector in Fig. 2. A GPU thread will read r different *LE* in *LA* according to *sip*, and then write the merged *LE* to a temporary *LE*. The *bip* vector of the *sip* is used to update the temporary *LE*. After scanning the *bip* vector, the cardinality of *sip* is estimated based on the temporary *LE*. As can be seen from Fig. 2, the sample data and *LA* will only be read, not modified. It ensures that k GPU threads can run at the same time to estimate the cardinality of the sampled host.

According to the above method, *LAA* and *L2S* can be transplanted to GPU to realize the real-time cardinalities estimation of high-speed networks.

The network traffic is divided into fixed-size time windows, and the network traffic in each time window is processed separately. Therefore, for high-speed network traffic, *L2S* can run in real-time when the time it takes in a time window is less than the size of a time window.

Although the static data is used in our experiments, *L2S* does not use the specific information of the network traffic such as the IP addresses distribution and only scans the network data once, so *L2S* can dynamically process the high-speed network data.

6 Experiment

This paper uses a set of real-world high-speed network data to evaluate the performance of *LAA* and *L2S*. The network data used in this article is the data set downloaded from the WIDE website [Fontugne, Abry, Fukuda et al. (2017)]. WIDE data set is a real-time network data stream collected from 1 Gb/s high-speed network. The network data used in this paper is the network traffic that lasts for 10 minutes from 13:00 on May 9, 2018, and April 9, 2019, respectively. Denote by WIDE 20180509 and WIDE 20190409 these two experiment data. Tab. 1 lists the information of our experiment data, including the IP number in ANet and BNet($\|AIP\|$ and $\|BIP\|$), packets number(*#Pkt*), unique IP pair number, average cardinality of each *aip* and so on.

Table 1: Details of experiment network traffic

Traffic	$\|\boldsymbol{AIP}\|$	$\|\boldsymbol{BIP}\|$	#Pkt	Unique IP pair	Average cardinality
WIDE 20180509	146978	1675748	157858284	1967100	13.38
WIDE 20190409	137477	1723667	164246725	2042693	14.86

According to the way of Section 5, this experiment is carried out on GPU. The GPU in the experiment is Nvidia GTX 950 m, with 2 GB graphic memory and 640 CUDA cores. In this experiment, g is set to 2^{10} and V is set to 2^{13}.

6.1 The influence of row number

To compare the impact of different rows on the accuracy of the estimation results, *LAA* is tested when r increases from 1 to 40.

6.1.1 Estimating accuracy

It can be seen from the analysis in Section 3 that with the increase of the number of *LA* rows, the estimation accuracy of *LAA* is not monotonically increasing. Figs. 3 and 4 show that *LAA*'s estimating result with rows 1, 3 and 40 on these two experiment data respectively. x-coordinate of each subgraph represents the actual cardinality, and y-coordinate represents the estimated cardinality of different *aip*. The closer the points in the graph are to the straight line x=y, the higher the accuracy of the estimation is.

As can be seen from Figs. 3 and 4, when r is 1, the estimated value of *LAA* deviates greatly from the real value. When the number of rows increases to 3, the estimated value of *LAA* is close to the real value. With the further increase in the number of rows, the estimated value of *LAA* is not more close to the real value. This shows that increasing the number of rows does not always reduce the error.

To compare the accuracy of the estimation more accurately, we use average bias to measure the experiment results. For a host *aip* whose cardinality is n and the estimated value is n', its estimation bias the $(n' - n)/n$. There are many *aip* in a time window, the

mean of these estimation bias of all *aip* is called average bias. To compare the fluctuation of estimation bias, the standard of estimation bias of all *aip* is calculated. Generally speaking, the lower the average bias is, the higher estimation accuracy is.

However, the average bias can't fully reflect the accuracy of the estimation, because the overestimation and underestimation will offset each other and reduce the average deviation. For this reason, we define absolute bias as $|n' - n|/n$ where n is the cardinality of *aip*, n' is its estimated value, $|n' - n|$ is the absolute value of the difference between n and n'. The mean value of the absolute bias of all *aip* is called the average absolute bias, and the standard deviation of the absolute bias of all *aip* is called the absolute bias standard deviation. Average absolute bias and standard deviation of absolute bias can reflect the influence of overestimation and underestimation on the accuracy of results.

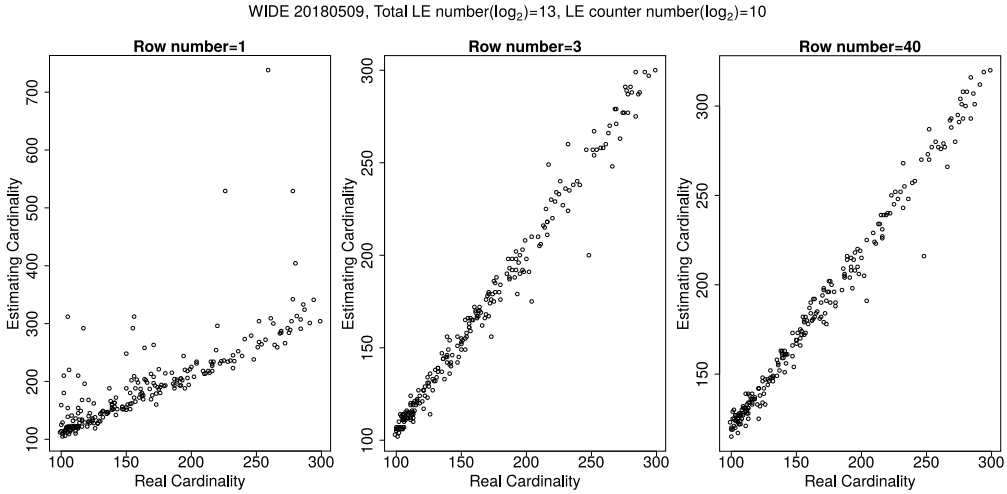

Figure 3: The estimating cardinalities distribution under different row numbers on experiment data WIDE 20180509

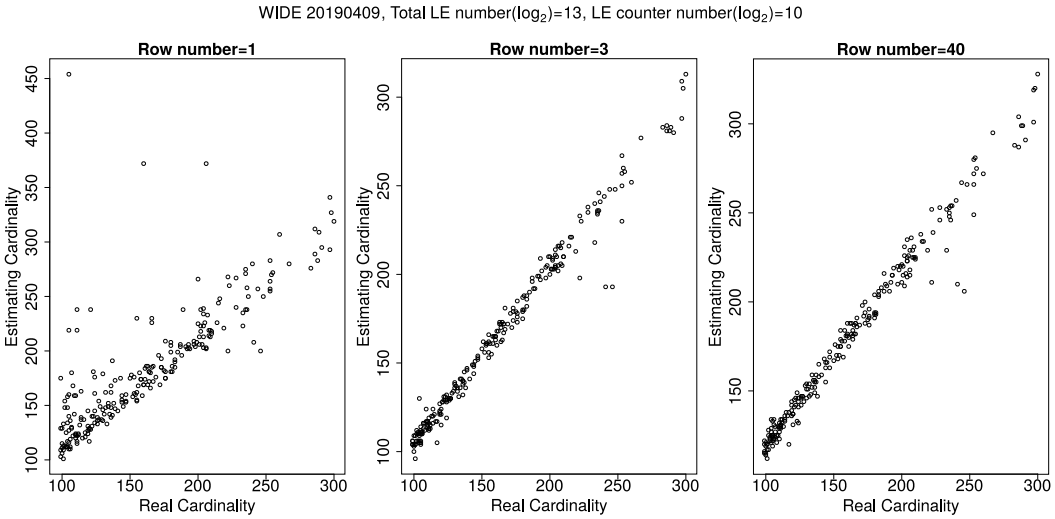

Figure 4: The estimating cardinalities distribution under different row numbers on experiment data WIDE 20190409

In the experiment, we use average bias, average absolute bias, the standard deviation of bias, the standard deviation of absolute bias to compare the estimation under different row numbers. Average bias, average absolute bias, the standard deviation of bias, the standard deviation of absolute bias of *LAA's* estimated result on two experiment data are shown in Figs. 5 and 6.

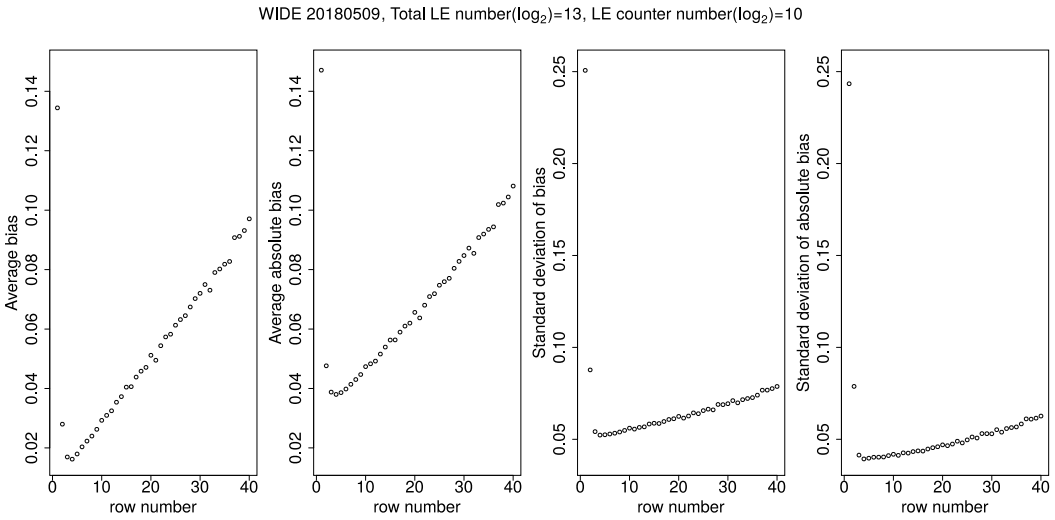

Figure 5: Estimating accuracy under different row number on experiment data WIDE 20180509

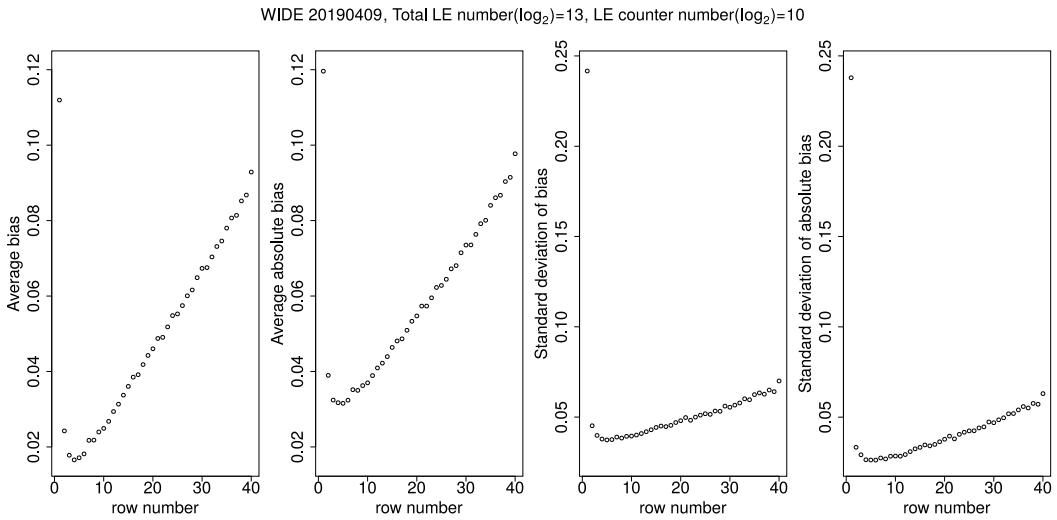

Figure 6: Estimating accuracy under different row number on experiment data WIDE 20190409

In Figs. 5 and 6, x-coordinate represents the number of rows of *LA*, and y-coordinate represents the average bias, average absolute bias, the standard deviation of bias, the standard deviation of absolute bias respectively. In each figure, the first and second subplots (i.e., the average bias and the average absolute bias subplots) have the same y-coordinate limitations; the third and the fourth subplots (i.e., the standard deviation of bias and standard deviation of absolute bias subplots) have the same y-coordinate limitations. So, it's easy to compare the differences between the average bias and the average absolute bias or the standard deviation of bias and the standard deviation of absolute bias.

It can be seen from Figs. 5 and 6 that the average bias, average absolute bias, the standard deviation of bias, the standard deviation of absolute bias all have the same variation trend that decreasing first and then increasing. The smaller the average bias, average absolute bias, the standard deviation of bias and the standard deviation of absolute bias are, the better the estimated result is. Consequently, it is not that the larger the number of rows, the higher the accuracy of estimation. Choosing an appropriate *r* is helpful to improve the accuracy.

By comparing the first and second subplots in both Figs. 5 and 6, we can see that the average absolute bias is higher than the average bias. This is because, in the average absolute bias, the overestimated value and the underestimated value will not offset each other like that in the average bias which makes the calculation result lower. However, the absolute bias moves the underestimation to the same side of the overestimation, which makes the standard deviation of the absolute bias smaller than the standard deviation of the bias, as shown in the third and fourth subplots of Figs. 5 and 6.

6.1.2 Running time

An increase in the number of rows also increases the time spent on the algorithm because more hash functions will be called to locate these *LE*. Figs. 7 and 8 show the running time of *LAA* on two datasets under different row numbers.

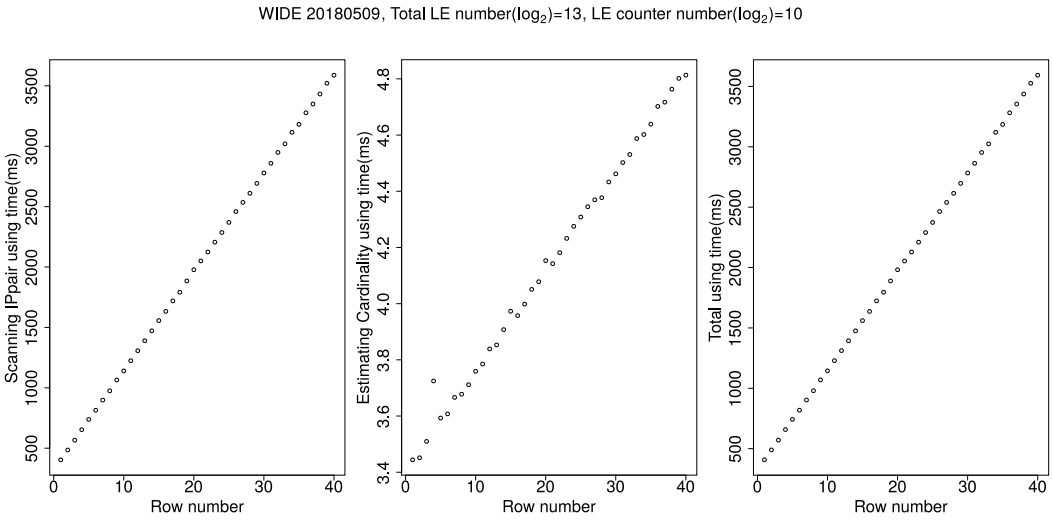

Figure 7: Experiment running time under different row number on experiment data WIDE 20180509

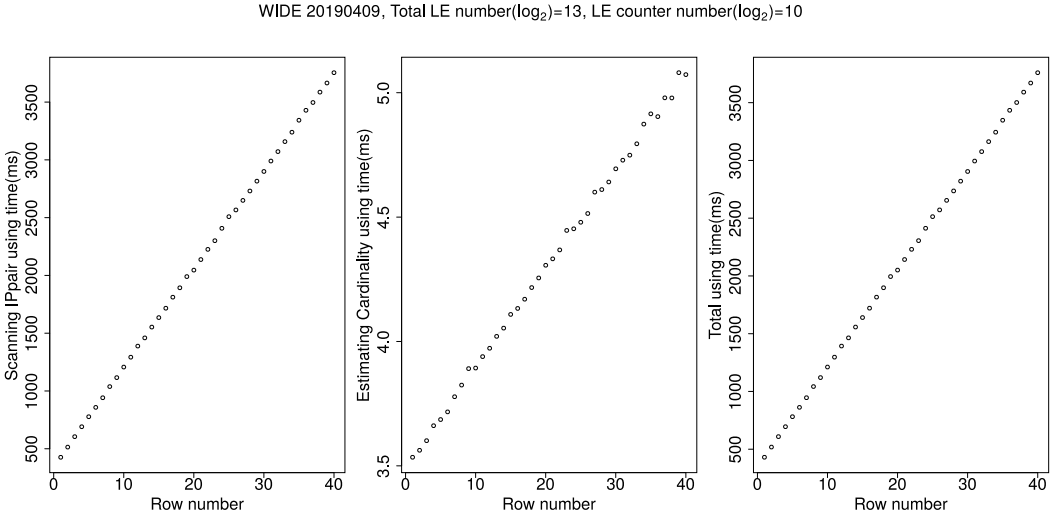

Figure 8: Experiment running time under different row number on experiment data WIDE 20190409

The running time is divided into two parts: IP address pairs scanning time and cardinality estimating time. The sum of them is called the total using time. The time of all the three subplots in both Figs. 7 and 8 are increasing with the row number. A bigger row number

requires more running time. It can be seen from Figs. 7 and 8 that the IP address pair scanning time is much longer than the cardinality estimating time. Actually, in our experiments, the IP address pair scanning time is 117 to 745 times the cardinality estimating time. Hence, the IP address pair scanning time has more influence on the total time.

Because this paper transplanted *LAA* to GPU, even when the number of rows is 40, the time used by *LAA* is not more than 3760 milliseconds, i.e. not more than 3.76 seconds. For example, in our experiment, when the number of rows is 3, the time used by *LAA* is only 570 and 609 milliseconds, and the speed of processing packets is 263.98 and 257.35 mpps (mpps means millions of packets per second); when the number of rows is 4, the time used by *LAA* is only 657 and 695 milliseconds, and the speed of processing packets is 229.00 and 225.50 mpps. Therefore, the hosts' cardinalities of a high-speed network can be estimated accurately in real-time by transplanting *LAA* to GPU using the method proposed in this paper.

6.2 Sampling revision experiment

In the previous subsection, we illustrate the accuracy and running time of the unadjusted *LAA* under different *r*. The accuracy of the estimated result can be improved by sampling revision. In this subsection, we will show and analyze the experiments of sampling adjustment.

6.2.1 Sampling revision accuracy

As can be seen from Figs. 5 and 6, *LAA* can achieve high accuracy when the number of rows is between 3 and 5. To analyze the improvement of sampling revision on the estimation results, in the experiment of *L2S*, we set the row number of *L2S* to 4. In *L2S*, sampling parameters (i.e., α and β) will affect the accuracy of the revision results. To compare the improvement of the results by different sampling parameters, we test the results when α are 5, 10, 20 and 30, and β are 5, 10, 20 and 30. The tuple composed of α and β, recorded as $<\alpha, \beta>$, is called sampling step number pair. Under each sampling step number pair, we also compare the influence of expected estimating value calculated by using mean and median of sampling cardinality, which are expressed by *L2S-mean* and *L2S-median* respectively.

To compare the accuracy of *L2S*, we compare the estimation results with *DCDS* [Wang, Guan, Zhao et al. (2014)], *VBFA* [Liu, Qu, Gong et al. (2016)] and *GSE* [Shin, Im and Yoon (2014)]. Figs. 9 and 10 describes the accuracy of different algorithms under two sets of experimental traffic. Subplots in Figs. 9 and 10 represent the average bias, average absolute bias, the standard deviation of bias and the standard deviation of absolute bias of different algorithms. The x-coordinate of each subgraph represents the sampling step number pair.

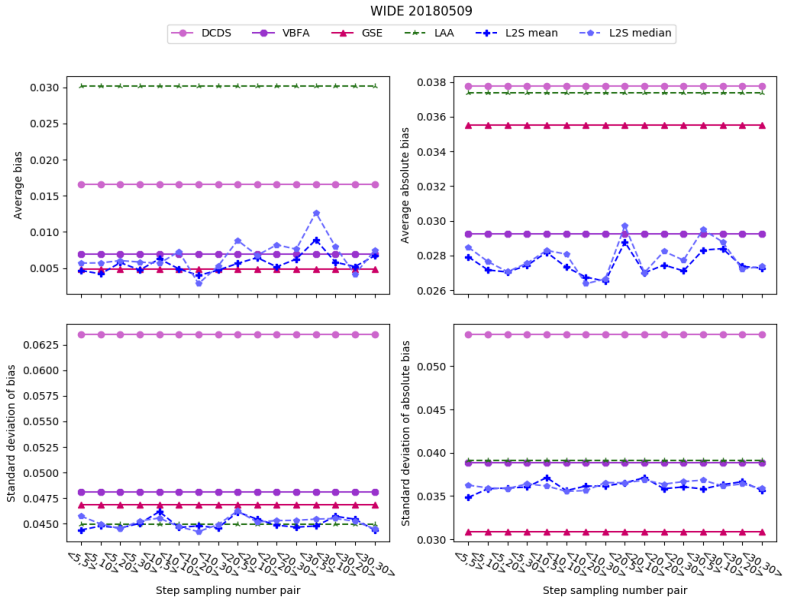

Figure 9: The accuracy of different algorithms on WIDE 20180509

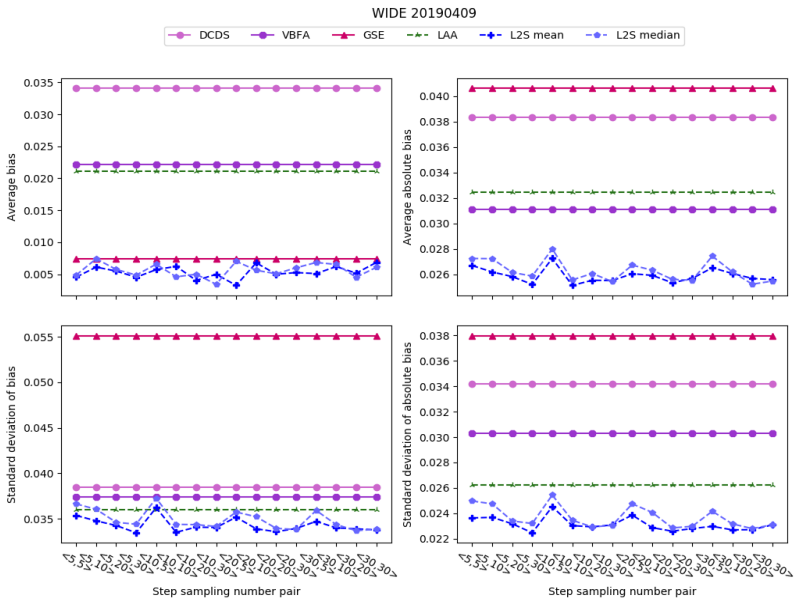

Figure 10: The accuracy of different algorithms on WIDE 20190409

The average bias is mainly used to measure the deviation of the estimated cardinality from the accurate value after the offset between the overestimation and the underestimation. The average bias may be higher or lower than zero. To compare the

average bias of different algorithms, the average bias in this experiment is the absolute value of the average bias. Different from the average absolute bias, the absolute value of the average bias can not reflect the overestimation and underestimation. The calculation method is to get the average bias first and then take the absolute value of it. Let $\|AIP\|$ indicate the number of *aip*, n_i' indicate the estimated cardinality of the ith *aip* whose actual cardinality is n_i. Then the absolute value of the average bias is calculated as $\left|\frac{1}{\|AIP\|}\sum_{i=1}^{i\leq\|AIP\|}\frac{n_i'-n_i}{n_i}\right|$. While the calculation method of average absolute bias is to take the absolute value of bias first and then calculate the average value, and the calculation formula is $\frac{1}{\|AIP\|}\sum_{i=1}^{i\leq\|AIP\|}\frac{|n_i'-n_i|}{n_i}$.

As can be seen from Fig. 9, for all algorithms, the average absolute bias is higher than the absolute value of the average bias. This further shows that the average absolute bias can more comprehensively reflect the accuracy of the estimated result. In Figs. 9 and 10, the average bias of *GSE* is lower than that of *DCD*, *VBFA* and *LAA*, but the average absolute bias of *GSE* is higher than that of *VBFA*. It shows that the algorithm with a good effect on the average bias does not necessarily have a good effect on the average absolute bias.

From the perspective of the difference between the estimated results and the accurate cardinalities, the average bias of *LAA* is higher than that of *DCD*, *VBFA* and *GSE*, and the average absolute bias is higher than that of *VBFA* and *GSE* in Figs. 9. In 10, the average deviation of *LAA* is lower than that of *DCDS* and *VBFA*, and the average absolute bias is lower than that of *DCDS* and *GSE*. It shows that the estimating deviation of *LAA* is similar to the existing algorithms.

From the perspective of the bias fluctuation, the bias standard deviation of *LAA* is lower than that of *DCDS*, *VBFA* and *GSE* in Figs. 9 and 10. The standard deviation of *LAA*'s absolute bias is only higher than that of *VBFA* and *GSE* in Fig. 10. This shows that the bias fluctuation degree of *LAA* is similar to the existing algorithms. Hence the unrevised *LAA* has similar accuracy to the existing algorithms.

The accuracy of *LAA* can be improved by step sampling revision, i.e., *L2S* algorithm, as can be seen in Figs. 9 and 10. In Figs. 9 and 10, the average bias and the average absolute bias of *L2S* are far lower than that of *LAA*, and also lower than that of the existing algorithms. To analyze the accuracy improvement of *L2S* more clearly, we define a reduction ratio to measure *L2S's* reduction of the average (absolute) bias and the standard deviation of (absolute) bias.

Let b denote *LAA*'s average (absolute) bias or standard deviation of (absolute) bias, and b' denote the *L2S*'s average (absolute) deviation or standard deviation of (absolute) bias. The reduction ratio of average (absolute) deviation or standard deviation of (absolute) bias is $\frac{b-b'}{b}*100\%$. Tabs. 2 and 3 list the reduction ratios of average (absolute) bias and (absolute) bias standard deviation when *L2S* uses the mean and median to calculate the sampling estimated value respectively, i.e., *L2S-mean* and *L2S-median*.

Table 2: The reduction ratio of *L2S-mean* under different step sampling number pairs

Traffic	α	β	Reduction ratio (%)			
			Average bias	Average absolute bias	Standard deviation of bias	Standard deviation of absolute bias
WIDE 20180509	5	5	84.5343	25.2805	1.1839	11.0196
		10	86.0690	27.2830	0.3388	8.4549
		20	80.9236	27.6081	0.7477	8.2548
		30	84.2686	26.5870	-0.2500	7.9424
	10	5	79.2214	24.5646	-2.8063	5.1661
		10	83.7857	26.8396	0.6485	8.9984
		20	86.7852	28.3776	0.2992	7.6786
		30	84.4045	29.0200	0.7416	7.6341
	20	5	81.2704	22.9752	-2.7075	6.7371
		10	78.6285	27.7243	-1.1564	5.1835
		20	82.8489	26.5201	0.1650	8.4357
		30	79.2547	27.4230	0.5684	7.9179
	30	5	70.4353	24.2113	0.3266	8.5014
		10	80.7459	23.9957	-1.7801	7.2516
		20	82.7373	26.7248	-1.1597	6.3820
		30	77.8209	26.9974	1.2011	8.9283
WIDE 20190409	5	5	78.4453	17.7376	1.7933	9.9254
		10	70.9934	19.2954	3.4324	9.7545
		20	73.9361	20.4180	4.8573	11.6815
		30	78.5078	22.3068	7.0708	14.3850
	10	5	72.8970	16.0212	-0.5614	6.5487
		10	70.4164	22.5287	6.9191	12.2325
		20	80.8363	21.3062	5.3494	12.6068
		30	76.3502	21.3695	5.4616	12.0394
	20	5	84.5242	19.7203	2.3243	9.0862
		10	67.8432	20.1071	5.8656	12.8564
		20	76.2632	21.9059	6.7538	13.9120
		30	75.0729	20.8549	5.7517	13.1114
	30	5	75.7847	18.2680	3.5947	12.4518
		10	70.6110	19.6590	5.5664	13.5118
		20	75.4908	20.8525	5.8816	13.4577
		30	67.6125	21.1392	6.1882	12.0212
Average			78.1037	23.3007	2.2690	9.8146

Table 3: The reduction ratio of *L2S-median* under different step sampling number pairs

Traffic	α	β	Reduction ratio (%)			
			Average bias	Average absolute bias	Standard deviation of bias	Standard deviation of absolute bias
WIDE 20180509	5	5	81.1368	23.7788	-1.8132	7.4136
		10	81.0298	26.0022	-0.1139	8.2019
		20	79.9764	27.5576	0.9168	8.4122
		30	80.6521	26.2436	-0.7859	7.0174
	10	5	81.2569	24.2709	-1.4003	7.6574
		10	75.8532	24.8377	0.4826	9.2013
		20	90.5733	29.4038	1.5437	9.0030
		30	82.5229	28.6878	0.0209	6.6486
	20	5	70.7298	20.4700	-2.9421	6.7008
		10	77.5631	27.7050	-0.5367	5.9217
		20	72.7454	24.4144	-0.8531	7.0828
		30	74.7670	25.7732	-0.9052	6.3526
	30	5	57.9694	21.0573	-1.1739	5.9026
		10	73.6337	22.9663	-1.3028	7.6757
		20	86.2858	27.1624	-0.7163	7.0517
		30	75.3508	26.7250	0.9407	8.3796
WIDE 20190409	5	5	76.9161	16.0260	-1.7356	4.8602
		10	64.8568	16.0759	-0.0881	5.7030
		20	72.7161	19.4176	3.8826	10.8474
		30	76.9121	20.2890	4.4684	11.6042
	10	5	68.8702	13.7384	-3.4430	3.0591
		10	78.1906	21.2185	4.5088	10.6257
		20	76.3189	19.6535	4.5921	12.6416
		30	83.7041	21.6179	5.1494	12.1466
	20	5	66.7204	17.6459	0.7521	5.6942
		10	73.3315	18.8713	2.1836	8.2809
		20	75.9669	21.0745	5.7323	12.9288
		30	71.4559	21.2953	6.0788	12.4489
	30	5	67.5964	15.4570	0.2683	7.9219
		10	69.0036	19.2537	4.5829	11.7114
		20	78.4889	22.2511	6.4213	13.1355
		30	71.2093	21.4953	6.0090	11.9817
Average			75.4470	22.2636	1.2726	8.5692

It can be seen from Tabs. 2 and 3 that the average (absolute) bias and (absolute) bias standard deviation of *L2S-mean* and *L2S-median* are all reduced. Among them, the reduction ratio of the average bias is the largest, which distributing from 57% to 90%, and the average bias reduction ratio of *L2S-mean* and *L2S-median* reach 78% and 75%, respectively. In Tab. 2, *L2S-mean* reduces the average absolute bias by more than 16%, the highest reduction ratio reaches 29%, and the average reduction ratio reaches 23%; in Tab. 3, *L2S-mean* reduces the average absolute bias by more than 13%, the highest

reduction ratio reaches 29%, and the average reduction ratio reaches 22%. Hence *L2S-mean* and *L2S-median* also have better performance on the average absolute bias.

From the perspective of reducing the fluctuation of estimation bias, *L2S*-mean and *L2S-median* also reduce the bias standard deviation. In Tabs. 2 and 3, although the bias standard deviation of *L2S-mean* and *L2S-median* is sometimes slightly higher than that of *LAA*, in general, the average of the bias standard deviation of *L2S-mean* and *L2S-median* is reduced by 2% and 1%, respectively. For *L2S-mean* and *L2S-median*, the reduction of absolute bias standard deviation is better than that of bias standard deviation. In Tab. 2, *L2S-mean* reduces the absolute bias standard deviation by more than 5%, the highest reduction ratio reaches 14%, and the average reduction ratio reaches 9%; in Tab. 3, *L2S-mean* reduces the absolute bias standard deviation by more than 3%, the highest reduction ratio reaches 13%, and the average reduction ratio reaches 8%. It can be seen from these experiments that *L2S-mean* and *L2S-median* also have an obvious good performance on absolute bias standard deviation.

It can be seen from the above analysis that the sampling revision algorithm, *L2S,* can not only improve the accuracy of the estimated results but also make the output results more stable.

6.2.2 Sampling using time

L2S can improve the accuracy of cardinality estimation with little additional sampling time. Tab. 4 shows the sampling time of *L2S* under different sampling parameters. *L2S-mean* and *L2S-median* use the same sampling data, so they have the same sampling time as shown in Tab. 4.

Table 4: Sampling time of *L2S*

α	β	Using Time (ms)	
		WIDE 20180509	**WIDE 20190409**
5	5	46.8586	45.6432
	10	80.7491	81.8959
	20	152.6805	157.3154
	30	219.5991	243.2497
10	5	19.0714	19.4652
	10	38.8418	36.9717
	20	68.5209	71.9618
	30	106.3597	110.5204
20	5	12.3004	10.2543
	10	21.4017	19.8806
	20	38.7598	38.4134
	30	52.8206	53.3638
30	5	7.9199	8.0159
	10	14.4797	15.7779
	20	23.2719	24.1849

30	35.7670	36.5461
Average	58.7126	60.8413

It can be seen from Tab. 4 that when α is the same, the sampling time increases with the increase of β. But when β are the same, the sampling time decreases with the increase of α. This is because when the sample step is determined, the bigger the β, the more the total quantity of samples; when β is determined, the larger the α, the less the total quantity of samples. The sampling time is directly proportional to the total quantity of samples. Therefore, the sampling time decreases with the increase of α and increases with the increase of β. In Tab. 4, the maximum sampling time of *L2S* is no more than 244 milliseconds (ms), the minimum is less than 9 ms, and the average sampling time is not more than 61 ms. Therefore, *L2S* can use a few additional times to improve the accuracy of *LAA*. It can be seen from the experimental results that the time used by *L2S*, including the time of the IP address pairs scanning, cardinalities estimating and the sampling process, is far less than the size of a time window, so *L2S* can run in real-time.

7 Conclusion

LAA algorithm is an efficient cardinality calculation method. *LAA* uses a fixed number of *LE* to calculate the cardinality of different hosts. Each *LE* will be used by multiple hosts to reduce memory consumption, and each host will also use multiple *LE* to improve estimation accuracy. But the accuracy of *LAA* is affected by some random factors. To reduce the influence of these random factors, this paper proposed a novel algorithm *L2S* which can improve the accuracy of *LAA* by cardinalities sampling. Both *LAA and L2S* are computation-intensive algorithms that can run in parallel. To improve the speed of *LAA* and L2S, this paper proposes a method to transplant them to GPU. When running on GPU, *L2S* can reduce the absolute bias of *LAA* by more than 22% with only 61 milliseconds extra time on average. High accuracy cardinality estimation is the foundation of many network security applications. In future work, we will study the applications of *L2S* in network security, such as intrusion detection, situation awareness, etc.

Acknowledgement: We thank Khushnood Abbas for contributing to the paper proofreading. We would like to thank the ICAIS 2020 reviewers and chairs for their valuable feedback and comments on the paper.

Funding Statement: The authors received no specific funding for this study.

Conflicts of Interest: The authors declare that they have no conflicts of interest to report regarding the present study.

References

Bhuyan, M. H.; Bhattacharyya, D. K.; Kalita, J. K. (2014): Network anomaly detection: methods, systems and tools. *IEEE Communications Surveys Tutorials*, vol. 16, no. 1, pp. 303-336.

Bianco, A.; Bonald, T.; Cuda, D.; Indre, R. (2013): Cost, power consumption and performance evaluation of metro networks. *IEEEOSA Journal of Optical Communications and Networking*, vol. 5, no. 1, pp. 81-91.

Ciriani, V.; Vimercati, S. D. C. D.; Foresti, S.; Jajodia, S.; Paraboschi, S. et al. (2010): Combining fragmentation and encryption to protect privacy in data storage. *ACM Transactions on Information and System Security*, vol. 13, no. 3, pp. 22.

CNNIC (2019): China Statistical Report on Internet Development.

Cohen, R.; Nezri, Y. (2019): Cardinality estimation in a virtualized network device using online machine learning. *IEEE ACM Transactions on Networking*, vol. 27, no. 5, pp. 2098-2110.

Fontugne, R.; Abry, P.; Fukuda, K.; Veitch, D.; Cho, K. et al. (2017): Scaling in internet traffic: a 14 year and 3 day longitudinal study, with multiscale analyses and random projections. *IEEE ACM Transactions on Networking*, vol. 25, no. 4, pp. 2152-2165. http://www.cac.gov.cn/2019-08/30/c_1124938750.htm.

Huang, L.; Yang, Q.; Zheng, W. (2018): Online hashing. *IEEE Transactions on Neural Networks and Learning Systems*, vol. 29, no. 6, pp. 2309-2322.

Liu, W.; Qu, W.; Gong, J.; Li, K. (2016): Detection of superpoints using a vector bloom Filter. *IEEE Transactions on Information Forensics and Security*, vol. 11, no. 3, pp. 514-527.

Mittal, S. (2017): A survey of techniques for architecting and managing GPU register file. *IEEE Transactions on Parallel and Distributed Systems*, vol. 28, no. 1, pp. 16-28.

Pacifici, V.; Lehrieder, F.; Dán, G. (2016): Cache bandwidth allocation for p2p file-sharing systems to minimize inter-ISP traffic. *IEEE/ACM Transactions on Networking*, vol. 24, no. 1, pp. 437-448.

Qian, C.; Ngan, H.; Liu, Y; Ni, L. M. (2011): Cardinality estimation for large-scale rfid systems. *IEEE Transactions on Parallel and Distributed Systems*, vol. 22, no. 9, pp. 1441-1454.

Shin, S. H.; Im, E. J.; Yoon, M. (2014): A grand spread estimator using a graphics processing unit. *Journal of Parallel and Distributed Computing*, vol. 74, no. 2, pp. 2039-2047.

Silber, F.; Muller, A.; Habel, R. (2013): Generating data transfers for distributed GPU parallel programs. *Journal of Parallel and Distributed Computing*, vol. 73, no. 12, pp. 1649-1660.

Tarkoma, S.; Rothenberg, C. E.; Lagerspetz, E. (2012): Theory and practice of bloom filters for distributed systems. *IEEE Communications Surveys and Tutorials*, vol. 14, no. 1, pp. 131-155.

Vormayr, G.; Zseby, T.; Fabini, J. (2017): Botnet communication patterns. *IEEE Communications Surveys Tutorials*, vol. 19, no. 4, pp. 2768-2796.

Wang, J.; Yang, Y. Q.; Wang, T.; Sherratt, R. S.; Zhang, J. Y. (2020): Big data service architecture: a survey. *Journal of Internet Technology*, vol. 21, no. 2, pp. 393-405.

Wang, P.; Guan, X.; Zhao, J.; Tao, J.; Qin, T. (2014): A new sketch method for measuring host connection degree distribution. *IEEE Transactions on Information Forensics and Security*, vol. 9, no. 6, pp. 948-960.

Xiao, Q.; Chen, S.; Zhou, Y.; Chen, M.; Luo, J. et al. (2017): Cardinality estimation for elephant flows: a compact solution based on virtual register sharing. *IEEE/ACM Transactions on Networking*, vol. 25, no. 6, pp. 3738-3752.

Xie, H.; Yan, Z.; Yao, Z.; Atiquzzaman, M. (2019): Data collection for security measurement in wireless sensor networks: a survey. *IEEE Internet of Things Journal*, vol. 6, no. 2, pp. 2205-2224.

Xu, Y.; Ren, J.; Wang, G.; Zhang, C.; Yang, J. et al. (2019): A blockchain-based nonrepudiation network computing service scheme for industrial IoT. *IEEE Transactions on Industrial Informatics*, vol. 15, no. 6, pp. 3632-3641.

Zargar, S. T.; Joshi, J.; Tipper, D. (2013): A survey of defense mechanisms against distributed denial of service (DDoS) flooding attacks. *IEEE Communications Surveys Tutorials*, vol. 15, no. 4, pp. 2046-2069.

Zhuang, D.; Chang, J. M. (2019): Enhanced peerhunter: detecting peer-to-peer botnets through network-flow level community behavior analysis. *IEEE Transactions on Information Forensics and Security*, vol. 14, no. 6, pp. 1485-1500.