Minimizing Noise in HyperLogLog-Based Spread Estimation of Multiple Flows

Dinh Nguyen Dao† *Inha Univ.* nguyen@inha.edu
Rhongho Jang† *Wayne State Univ.* r.jang@wayne.edu
Changhun Jung *Ewha Womans Univ.* mizno@ewha.ac.kr
David Mohaisen *Univ. of Central Florida* mohaisen@ucf.edu
DaeHun Nyang *Ewha Womans Univ.* nyang@ewha.ac.kr

***Abstract*—Cardinality estimation has become an essential building block of modern network monitoring systems due to the increasing concerns of cyberattacks (e.g., Denial-of-Service, worm, spammer, scanner, etc.). However, the ever-increasing attack scale and the diversity of patterns (i.e., flow size distribution) will produce the biased estimation of existing solutions if apply a monotonic hypothesis for network traffic. The most representative solution is virtual HyperLogLog (vHLL), which extended the proven HLL, a single element cardinality estimation solution, to a multi-tenant version using a memory random sharing and noise elimination approach. In this paper, we show that the assumption made by vHLL's does not work for large-scale network traffics with diverse flow distributions. To resolve the issue, we propose a novel noise elimination method, called Rank Recovery-based Spread Estimator (RRSE), which is tolerant to both attack and normal traffic scenarios while using limited computation and storage. We show that our recovery function is more reliable than state-of-the-art approaches. Moreover, we implemented RRSE in a programmable switch to show the feasibility.**

***Index Terms*—Network Anomaly Detection, Cardinality Estimation, Sketch, Programmable Switch**

I. Introduction

Cardinality estimation is a crucial primitive of network security function to address various cyberattacks, including Denial-of-Service (Dos), worm, spammer, scanner, among others. The main challenge is to estimate a large number of distinct elements under computation and storage constraints. As of the second quarter of 2020, Internet users are 4.8 billion, a 1,239% of growth from 2000 [1]. The massive user base not only generates a massive amount of data with a high diversity but also increases the complexity of the used network with their devices. Moreover, the diverse network patterns (*i.e.,* normal and attack traffics) existing in the modern network require cardinality estimation solutions to be adaptive and efficient in terms of memory management.

To address the aforementioned challenges, scalable and memory-efficient measurement data structures, such as sketches were proposed. Unlike a multiplicity estimation that counts the frequency of an identical element [2]–[9], the cardinality estimation sketches count the number of distinct elements [10]–[25] and are classified into two types, bitmap-based [12]–[24] and register-based approaches [10], [11]. The major drawback of most bitmap-based approaches is the linearly increasing counting capacity, which motivated later solutions like LogLog [10] and HyperLogLog to scale up the counting range with adaptive bitmap [26]. Especially, HyperLogLog (HLL) [11] is widely adopted in practice owing to its solid estimation theory and scalability.

Driven by the applications' increasing complexity, demands for multi-tenant cardinality estimation (*i.e.,* counting the number of distinct destination IP addresses for every source IP address, spreader detection hereafter) are increasing. A straightforward solution to the multi-tenant cardinality estimation problem is to use a dedicated HLL encoder for each spreader, although that requires too much memory to be practical. For memory efficiency, several sketch-based solutions that enable multi-tenant counting with a memory sharing strategy were proposed [9], [11], [16], [18], [19]. Namely, a HLL-based solution of a virtual HLL (vHLL) [27] has been shown to be more scalable and accurate than other approaches, such as PCSA [19], PMC [9], MultiresolusionBitmap, [18], and Compact Spread Estimator (CSE) [15]. vHLL follows HLL's theory to perform each spreader's cardinality counting and enables multi-tenant estimations by allowing spreaders to share a universal memory space randomly. This design will naturally introduce a noise issue that has to be handled carefully. Our key observation is that vHLL applies a universal noise (an identical value) for all different-sized spreaders, which we found to be inaccurate in an attack traffic scenario (see section IV-D), where a massive number of medium and high spreaders exist in the network traffic (*i.e.,* abnormal flow distribution). Moreover, the vHLL cardinality estimation in a normal traffic scenario can be improved due to the coarse noise estimation strategy.

In this paper, we design a novel noise elimination algorithm for HLL-based multi-tenant cardinality counting schemes, called the Rank Recovery-based Spread Estimator (RRSE). RRSE uses a global register array for estimating millions of spreaders and applies a random register sharing technique that is commonly adopted in multi-tenant counting algorithms [2], [3], [15], [16], [27]–[29]. However, unlike the previous approaches that remove a universal noise (*i.e.,* global average) for different-sized flows, RRSE focuses on the precise noise estimation and elimination for every single spreader with negligible overhead. More importantly, our new concept of *rank distribution recovery* manipulates the recorded intermediate values (*i.e.,* rank values), which is fundamentally different from state-of-the-art approaches that eliminate noise *after* the intermediate value-based estimation [15], [16], [27],

†These authors contributed equally.

[28]. To do so, we collect and recover a rank distribution of the local registers—a register value distribution of a spreader—by leveraging the distribution of global registers; a universal random-sharing register array. Then, the recorded/tainted rank distribution will be recovered at rank-level (*i.e.,* fine-grained recovering). Eventually, our estimator performs cardinality estimation with the recovered rank distribution without considering the noise. Our experimental results show that RRSE can achieve more precise estimation than vHLL and MCSE with attack and normal traffic scenarios while requiring a negligible overhead by performing the recovery function using a dynamic programming technique. We further show that our RRSE can be embedded in a programmable switch to support future network systems.

Contributions. In this paper, we make the following contributions:

- We present a rank distribution recovery technique, a new direction for noise elimination in multi-tenant spread counting algorithms. Our technique carefully recovers tainted intermediate data (*i.e.,* recorded rank distribution) independently for each spreader, instead of the post-hoc noise reduction approach adopting a coarse universal noise for all spreaders.
- We provide an error bound analysis of our algorithm and prove our rank distribution recovery is unbiased.
- We show comparatively trace-based simulation results with two real-world datasets of different distributions to demonstrate the unbiased estimation of our algorithm. Through extensive experiments, we show that our approach provides a precise noise reduction for all ranges of spreaders than state-of-the-art algorithms, regardless of traffic distributions.
- We designed and implemented a spreader detection framework in a programmable switch (Tofino) to show its feasibility. Moreover, a comprehensive analysis of resource consumption and latency is conducted.

II. Background and Motivation

In this section, we start with a background description, namely HyperLogLog [11], which is a cardinality estimation scheme that has been widely used for spread estimation. Then, we explore advanced solutions that allow the multi-tenant spread estimation demanded by modern network monitoring systems. Next, we discuss the limitations of state-of-the-art solutions. Lastly, we describe the inspirations behind our approach. We note that our work focuses on not only achieving better accuracy but also a tolerant estimation under different network scenarios (*i.e.,* normal and attack traffics).

Cardinality estimation. HyperLogLog (HLL), as a logarithm-based cardinality counting algorithm, has been proven to be accurate and scalable in practice [30]. It uses multiple registers to encode distinct elements. HLL is based on two techniques. The first technique is *rank*, which is the position of the leftmost 1's bit in the hash value of a distinct element. For instance, the rank of **00001**001 is 5. The other technique is called *stochastic averaging*, which improves the robustness of the counting. When encoding, all destination IPs are split uniformly into m registers (*i.e.,* rank array $M[]$) using a hash function, and each register maintains the maximum rank value in $M[i]$. That is:

$$M[i] = max\{\rho(x), M[i]\} \tag{1}$$

where $\rho(x)$ is the rank of hash value x. To estimate, HLL calculates a normalized harmonic mean of all registers as

$$\hat{n_m} = \alpha_m \cdot m^2 \cdot (\sum_{i=1}^{m} 2^{-M[i]})^{-1}, \tag{2}$$

where α_m is a constant determined by m.

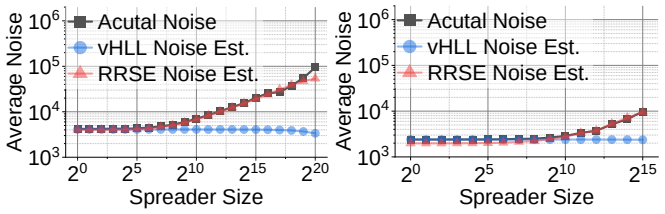
Multi-tenant cardinality estimation. HLL was designed to count a single spreader on scale. However, because of the increasing complexity of the deployment context, the multi-tenant spreader estimation became crucial. A straightforward solution then is to use dedicated registers for each spreader, but that requires a massive memory. To resolve this issue, vHLL [27] suggested maintaining a global register array for multiple spreaders, and each spreader uses only a random portion of the registers in the array. A register can be assigned to spreaders repeatedly. In vHLL, the registers' selection of each spreader follows a random behavior (*i.e.,* random memory sharing). We note the random memory sharing technique is widely used in designing memory-efficient data structures (*i.e.,* sketch) [2], [3], [15], [16], [28], [29]. However, the major challenge is *how to eliminate the noise caused by the memory sharing strategy?*

State-of-the-art noise handling solutions. In terms of noise elimination, vHLL and MCSE [16] are state-of-the-art works in multi-tenant spread estimation. The major difference between the two schemes is that vHLL is an exponential counter that shares memory at a register level, whereas MCSE is a linear counter sharing memory at a bit level. However, they both maintain a global array and consider the average of global estimation as a local noise, which will be eliminated from a local estimation value. To explain, let $M[1 \dots m]$ be the global register array of vHLL, n be the summation of all spreaders' cardinalities, and n_f is the cardinality of a flow f. In vHLL, all other flows $n - n_f$ is considered as noise that is distributed in $M[]$ following binomial distribution $Bino(n - n_f, \frac{1}{m})$. When each spreader uses s shared (virtual) register array that is significantly smaller than m ($s \ll m$), the expectation of the noise (random variable X) in $M[i]$ is given as $E(X) = \frac{n-n_f}{m}$. Then, the noise of s registers is given as $E(n_s - n_f) = s \cdot E(X) = s\frac{n-n_f}{m}$. Since $Var(\frac{n_s - n_f}{E(n_s - n_f)})$ approaches zero when s is sufficiently large [11], $E(n_s - n_f) \approx n_s - n_f$. Hence $n_s - n_f = s\frac{n-n_f}{m}$.

$$n_f = \frac{m \cdot s}{m - s} \cdot (\frac{n_s}{s} - \frac{n}{m}) \tag{3}$$

vHLL replaces n_s and n with HLL's ($\hat{n_s}$, $\hat{n}$) to estimate $\hat{n_f}$. Therefore,

$$\hat{n_f} = \hat{n_s} \cdot \frac{m}{m - s} - \hat{n} \cdot \frac{s}{m - s}, \tag{4}$$



(a) Attack Traffic (b) Normal Traffic

Fig. 1: Comparison of actual noise, vHLL's noise estimation, and our RRSE's noise estimation using real-world normal and attack network traces using 2Mb memory.

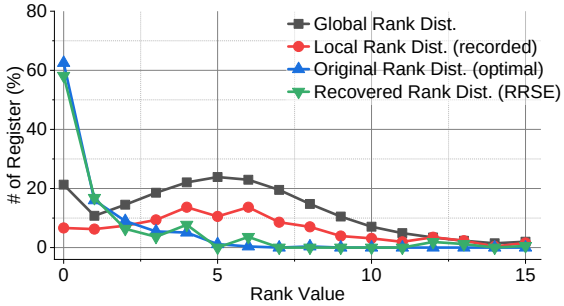


Fig. 2: A novel representation of cardinality estimation accuracy, namely rank (register) value distribution. The harmonic mean of all rank values in the distribution is the final estimation of HLL (see Eq. (2)). Original rank distribution is a noise-free rank value distribution. Closer distribution to the original distribution results in more accurate estimation.

the former is the estimated cardinality held by s registers before eliminating a noise and the latter is the noise estimated by vHLL. We note that $\hat{n}$ is the estimated cardinality of the global register array ($M[]$), s is the number of registers for a spreader, and m is the global array size. Simply put, vHLL assumes that the per-register noise is the average cardinality of global registers ($\frac{\hat{n}}{m-s}$, where $s \ll m$), and the virtual register array (s) noise for a spreader can be scaled up from the per-register noise, as $\frac{\hat{n}}{m-s} \cdot s$ (*i.e.,* the latter term in Eq.(4)).

Our observations. As discussed above, vHLL's noise estimation is based on the cardinality estimation of the global register array (*i.e.,* $\hat{n}$) regardless of individual spreaders' noise level. Since the noise increment of individual spreaders has minor effects to the overall noise estimation result, vHLL's noise estimation will lead to an underestimated noise, in general. Moreover, the estimated noise in vHLL, as a universal noise (a single value), will be subtracted from every spreader's estimation even though the noise varies in different spreaders. As Fig. 1 shows, the actual noise varies for different-sized flows, whereas the noise estimated by vHLL is negligibly small for medium and high spreaders. **(1)** We observed that medium and high spreaders increase the noise level of all spreaders due to the register sharing strategy and memory constraint. Accordingly, the noise level increases significantly when a massive amount of medium and high spreaders arrive simultaneously (*i.e.,* attack traffic) compared to the normal traffic, as shown in Fig. 1. **(2)** We also observed that most of the memory was occupied by small spreaders in both normal and attack scenarios. These two observations combined show why vHLL gives the biased overestimation. That is, the estimated noise of vHLL, which is a global average, is too small for medium and high spreaders. We note that the concept of assuming a global average as a local noise is used not only in vHLL, but a series of multi-tenant estimation algorithms [5], [15], [16]. Compared to these algorithms, our algorithm provides more precise noise reduction across all ranges and more tolerant spreader estimation regarding traffic distribution, as shown in Fig. 1.

TABLE I: Notation

$M[]$	global register array	$C_m[]$	global rank distribution
m	size of $M[]$	$C_s[]$	local rank distribution
$M_s[]$	local register array	$\hat{C_f}[]$	recovered $C_s[]$
s	size of $M_s[]$	$\hat{n_f}$	estimated cardinality
b	size of a register	`Hash()`	hash function
r	max rank value (=$2^b - 1$)	`R[]`	distinct integer array
w	r bits of `Hash`(f)	$\rho(w)$	rank calculation function

Our Approach. In this work, we demonstrate a different approach to deal with the noise caused by memory sharing. Our method, called the *Rank Distribution Recovery Function*, performs a fine-grained noise estimation and elimination for every single spreader, which is fundamentally different from vHLL that applies a universal noise (*i.e.,* identical value) to all spreaders. And unlike vHLL that performs noise elimination after estimation, we analyze the distribution of the recorded intermediate values (*i.e.,* tainted rank values due to memory sharing) of a spreader, then recover them to the original ones (*i.e.,* clean rank values; no memory sharing). Then, we use the cleaned intermediate data to estimate the spreader's cardinality without worrying about noise.

To recover the original rank distribution, we obtain a rank distribution from the tainted rank values of a spreader (hereafter, local rank distribution). Then, we leverage the global rank distribution obtained from the entire memory to recover the local rank distribution at a fine-grained rank level (see section III.D for details). Subsequently, we can leverage the recovered rank distribution to perform HLL estimation without considering the noise. Fig. 2 shows an example of rank distribution recovery. We note that the closer the recovered rank distribution to the original rank distribution, the more successful the rank recovery or noise elimination is. As shown in Fig. 2, the RRSE rank recovery results match well the original rank distribution.

III. Rank Recovery-based Spread Estimator(RRSE)

In this section, we introduce RRSE, a multi-tenant cardinality estimation algorithm based on the rank distribution recovery technique. We describe our data structure first, followed by the encoding and decoding algorithms. Next, we present the *Rank Distribution Recovery*, which is our main contribution. Finally, a theoretical analysis of our scheme, in terms of estimation bias and variance, is given. Table I shows the notations used in this paper.

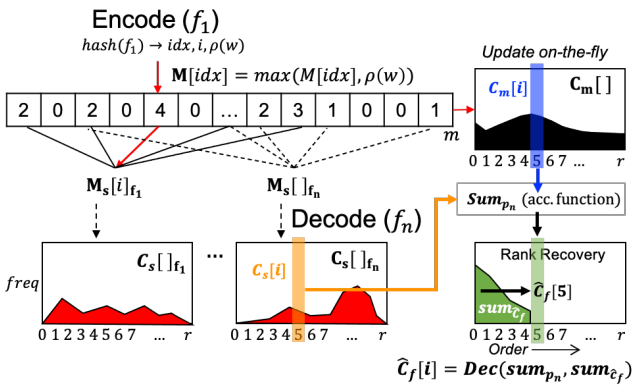


Fig. 3: RRSE: Data Structure with encoding and decoding processes. $M[]$ is the global register array, $M_s[]$ is a local (virtual) register array, $C_m[]$ is the global rank distribution, and $C_s[]$ is a local rank distribution. $C_m[]$ is updated on-the-fly and $C_s[]$ is retrieved when decoding. The decoding function $\hat{C}_f$ requires sum_{p_n} and $sum_{\hat{C}_f}$, which are an accumulative function realized by dynamic programming.

A. Data Structure: Random Register Sharing

As shown in Fig. 3, RRSE maintains a global register array, $M[]$, of which the size is m and each register is b bits. For memory efficiency, the global array allows register random sharing by using a hash function `Hash()`, which means that each register can be randomly shared by multiple spreaders. Meanwhile, each spreader is assigned s registers, namely the local (virtual) register array $M_s[]$, where s is much smaller than m.

B. Encoding

Algorithm 1 demonstrates the source IP (`srcIP`)-based encoding process. For each `dstIP` associated with a `srcIP`, we first hash the `dstIP`, and then use the hashed value (x) to derive two intermittent variables: i and w (line 3), where i is the first $c = log_2(s)$ bits of x (*i.e.,* $i \in [0, s-1]$) that are for randomly selecting the a register among s registers assigned to `srcIP`. To locate the selected register in the global array ($M[idx]$), RRSE hashes `srcIP` after `XOR` with $R[i]$, where $R[]$ is a constant and distinct integer array (line 4). Then, the next $r = 2^b - 1$ bits of x (*i.e.,* w) are used to calculate an encoding value, namely rank ($\rho(w)$), which is the number of consecutive 0's at the end of w plus one. Finally, $M[idx]$ is updated only if $\rho(w)$ is larger than $M[idx]$. We note that encoding of both RRSE and vHLL follows the theory of HLL, although RRSE applies a novel noise removing technique that recovers a local register array (*i.e.,* encoded values of a `srcIP`) by referencing a global rank distribution ($C_m[]$), which is the distribution of the encoded rank values in the global array $M[]$. To simplify and speed up the decoding process, RRSE records $C_m[]$ on the fly (line 7) with a negligible memory overhead ($\mathcal{O}(r), r \in [1, 16]$).

C. Decoding

Algorithm 2 shows RRSE's decoding proceess. To decode a spreader `srcIP`, RRSE retrieves the encoded rank values

Algorithm 1: Encoding

```
1  forall srcIP, dstIP ← pkt_f do
2      x ← Hash(dstIP);
3      i ←< x_1 x_2 ··· x_c >_2; w ←< x_{c+1} x_{c+2} ··· x_{c+r} >_2;
4      idx ← Hash(srcIP ⊕ R[i]);
5      if M[idx] < ρ(w) then
6          /*Record Global Rank Distribution on the fly*/
7          C_m[M[idx]]--; C_m[ρ(w)]++;
8          M[idx] := ρ(w);
9      end
10 end
```

Algorithm 2: Decoding

```
1  Function DECODING(srcIP):
2      for i = 1 to s do
3          idx := Hash(srcIP ⊕ R[i]);
4          C_s[M_s[idx]] += 1; /*Retrieve local rank dist.*/
5      end
6      Ĉ_f[] = RANK_DIST_RECOVERY(C_s[], C_m[]);
7      n̂_f := α_s s^2 (Σ_{i=0}^{r} (Ĉ_f[i] 2^{-i}))^{-1};
8      /*Low or high spreader correction*/;
9      if n̂_f ≤ (5/2) s then
10         n̂_f :=LINEAR_COUNTING(M_s); [17] for details
11     end
12     if n̂_f > (1/30) × 2^32 then
13         n̂_f := −2^32 log(1 − n̂_s/2^32); [27] for details
14     end
15     return n̂_f
16 Function RANK_DIST_RECOVERY(C_s[], C_m[]):
17     /* Dynamic Programming */
18     Set Ĉ_f[0 ... r] = 0, sum_{P_n} = 0, sum_{Ĉ_f} = 0;
19     for i = 0 to r do
20         sum_{P_n} = sum_{P_n} + (C_m[i] − C_s[i])/(m − s);
21         Ĉ_f[i] = (C_s[i] − ((C_m[i] − C_s[i])/(m − s)) · sum_{Ĉ_f}) / sum_{P_n};
22         sum_{Ĉ_f} = sum_{Ĉ_f} + Ĉ_f[i];
23     end
24     return Ĉ_f[]
```

from its local register array ($M_s[]$) and obtains the local rank distribution ($C_s[]$) (lines 2-5). We note that this is a low-cost process because $M_s[]$ and $C_s[]$ are small. As demonstrated in Fig. 2, we observe that the rank distribution of $M_s[]$ ($C_s[]$) is inappropriately shifted from the original distribution ($C_f[]$) that is recorded without noise or register sharing. The shift occurs because the registers are not dedicated to a single spreader but shared by multiple spreaders. Therefore, some rank values will be higher than the ground truth since the registers are always updated by a larger rank value. As a result, the rank distribution $C_s[]$ biases to higher rank values, which results in an overestimation of the cardinality. Based on this observation, RRSE aims to recover $C_s[]$ to an ideal status $\hat{C_f}[]$, where $\hat{C_f}[] \approx C_f[]$ (lines 16-24; see section III-D), and then uses the recovered $\hat{C_f}[]$ to estimate the cardinality (lines 6-7),

$$\hat{n_f} = \alpha_s s^2 (\sum_{i=0}^{r} \hat{C}_f[i] 2^{-i})^{-1} = \alpha_s s^2 (\sum_{j=1}^{s} 2^{-M_s[j]})^{-1}, \quad (5)$$

where $\alpha_{16} = 0.673, \alpha_{32} = 0.697, \alpha_{64} = 0.709$, and $\alpha_s = 0.7213/(1 + 1.079/s)$ for $s \geq 128$ [11]. Finally, as in HLL [11], RRSE performs estimation corrections for low spreaders (lines 9-11) and high spreaders (lines 12-14).

D. Local Rank Distribution Recovery

Here, we describe the rank distribution recovery function, which is a probabilistic approach for calculating $\hat{C_f}[]$ using the recorded rank distribution $C_s[]$ and the global rank distribution $C_m[]$ (Algorithm 2, lines 16-24). Fig. 3 also gives an overview of the decoding process. Given a flow f's rank distribution ($C_s[]$), we infer a rank's frequency of the original rank distribution from the lowest to the highest rank, respectively (line 19). For each rank i, we calculate the original frequency of rank i ($\hat{C_f}[i]$) (line 21), which is the realization of Eq. (6). In our algorithm, sum_{P_n} denotes the total number of rank 0 to i over the global register array excluding spreader's registers. Then, $sum_{\hat{C}_f}$ denotes the total count of previous lower ranks of a local register array. sum_{P_n} and $sum_{\hat{C}_f}$ are accumulated with dynamic programming. We note that both time and space complexities of `RANK_DIST_RECOVERY` are $\mathcal{O}(r)$, where r is a maximum value of a register which is small (*i.e.,* $7, 15$). Moreover, the overall rank distribution recovery process is lightweight since $C_s[]$ can be retrieved with negligible overhead and $C_m[]$ is recorded on the fly. Theorem 1 shows the processes of deriving $\hat{C}_f[]$.

Theorem 1. *Let i be a random variable that is recorded in our universal register array ($M[]$), where $i \in [0, r]$ and r is the range of rank values. Also, let $C_m[i]$ be the frequency of rank i in $M[]$, $C_s[i]$ be the frequency of rank i in a spreader's registers ($M_s[]$), where $M_s[] \subset M[]$. Let $P_n[i]$ be the probability of rank i over the global register array excluding spreader's registers. Assume M and M_s follow the same distribution. Then,*

$$C_f[i] = \frac{C_s[i] - P_n[i] \sum_{lo=0}^{i-1} C_f[lo]}{\sum_{lo=0}^{i} P_n[lo]}. \tag{6}$$

Proof. First, let X be the event that an original rank value of a register in $M_s[]$ (low) is overwritten by a higher rank value i of $M[]$, then the probability of X is

$$P(X) = \frac{C_f[lo]}{s} \cdot P_n[i], \tag{7}$$

where $P_n[i] = \frac{C_m[i] - C_s[i]}{m - s}$, and m, s are the size of $M[]$ and $M_s[]$, respectively. $P(X)$ defines the probability that a lower rank value low recorded in a flow's local registers be overwritten by a higher rank value i (noise) from global registers due to memory sharing. The former is the fraction of rank value low supposed to be recorded in local registers without the overwriting issue. Then, it is multiplied by $P_n[i]$ (i's probability in the global register) assuming that M and M_s have the same distribution, which represents the probability of the overwriting event. Here, $C_s[i]$ is subtracted from $C_m[i]$ for improving accuracy.

Similarly, let Y be the event that an original value of a register in $M_s[]$ (i) is overwritten by a higher rank value $high$ of $M[]$, then Y's probability is

$$P(Y) = \frac{C_f[i]}{s} \cdot P_n[hi]. \tag{8}$$

Thus, the times where all smaller ranks become rank i in $M_s[]$ is given by

$$H_i = \sum_{lo=0}^{i-1} P(X) \cdot s = P_n[i] \cdot \sum_{lo=0}^{i-1} C_f[lo], \tag{9}$$

and the times where the current rank i became higher rank in $M_s[]$ is calculated by

$$L_i = \sum_{hi=i+1}^{r} P(Y) \cdot s = C_f[i] \sum_{hi=i+1}^{r} P_n[hi]. \tag{10}$$

The observed frequency for rank i ($C_s[i]$) should be equal to $C_f[i]$, but this is not the case due to register sharing. Instead, $C_s[i]$ can be seen as $C_f[i]$ (true local rank distribution) **plus** H_i in Eq. (9), the number of registers with lower ranks overwritten by rank i, **minus** L_i in Eq. (10), the total amount of rank i's register that has been overwritten as a higher value. Hence,

$$C_s[i] = C_f[i] + P_n[i] \sum_{lo=0}^{i-1} C_f[lo] - C_f[i] \sum_{hi=i+1}^{r} P_n[hi] \tag{11}$$

Finally, we can derive $C_f[i]$ as

$$C_s[i] - P_n[i] \sum_{lo=0}^{i-1} C_f[lo] = C_f[i] \cdot (1 - \sum_{hi=i+1}^{r} P_n[hi]),$$

$$C_f[i] = \frac{C_s[i] - P_n[i] \sum_{lo=0}^{i-1} C_f[lo]}{1 - \sum_{hi=i+1}^{r} P_n[i]}.$$

Let $\hat{C_f}[i]$ be a variable to estimate $C_f[i]$, given as,

$$\approx \frac{C_s[i] - P_n[i] \sum_{lo=0}^{i-1} C_f[lo]}{\sum_{lo=0}^{i} P_n[lo]} = C_f[i]. \quad \square \tag{12}$$

We note that when estimating $\hat{C_f}[i]$, RRSE use estimated $\hat{C}_f[lo]$ instead of the unavailable ground truth $C_f[lo]$. Since $\hat{C_f}[i]$s' estimations follow the order $i = 0, 1, \ldots, r$, $\hat{C}_f[lo]$s ($lo \in [0, i-1]$) are available for $\hat{C_f}[i]$ with dynamic programming.

E. Bias and Standard Error

Finally, RRSE's estimation is shown to be unbiased, and the upper bound of the standard error is derived.

Theorem 2. *Let $\hat{n_s}$ be the estimation of RRSE using s shared registers, and n_f be the estimation without the register sharing. If $E(\hat{C}_f[i]) \approx C_f[i]$, then $E(\hat{n_f}) = E(n_f)$, which means our estimator is unbiased.*

Proof. Given Eq. (2), Eq. (5), and Eq. (6),

$$E(\hat{n_f}) = E(\frac{\alpha_s \cdot s^2}{\sum_{i=0}^{r} \hat{C_f}[i]2^{-i}}) \approx E(\frac{\alpha_s \cdot s^2}{\sum_{i=0}^{r} C_f[i]2^{-i}}) = E(\frac{\alpha_s \cdot s^2}{\sum_{i=1}^{m} 2^{-M[i]}}) = E(n_f). \quad \square \quad (13)$$

Theorem 3. *Given an arbitrary flow f with a cardinality estimation of $\hat{n_f}$, the upper bound of RRSE's relative standard error is*

$$StdErr(\hat{n_f}) = \frac{\sqrt{Var(\hat{n_f})}}{n_f} < \frac{1.04}{\sqrt{s}} + \frac{\sqrt{\varepsilon}}{n_f}, \quad (14)$$

where the error bound is defined as

$$\varepsilon = \frac{\alpha_s \cdot s^2}{\sum_{i=0}^{r} C_\varepsilon[i]2^{-i}} \quad (15)$$

Proof. Given s shared registers for a flow f, the HLL-based cardinality estimation n_s may include a noise n_n due to the register sharing. Let $\hat{n_f}$ be the estimation of RRSE with rank distribution recovery (*i.e.,* noise reduction function). Then, the variance of the $\hat{n_f}$ is

$$\begin{aligned} Var(\hat{n_f}) &= Var(n_s - n_n) \\ &= Var(n_s) + Var(n_n) - 2 \cdot Cov(n_s, n_n) \\ &< Var(n_s) + Var(n_n). \end{aligned} \quad (16)$$

Let ε be the noise worst case. Then,

$$Var(\hat{n_f}) < Var(n_s) + \varepsilon^2. \quad (17)$$

Accordingly,

$$StdErr(\hat{n_f}) = \frac{\sqrt{Var(n_f)}}{n_f} < \frac{\sqrt{Var(n_s) + \varepsilon^2}}{n_f} \quad (18)$$

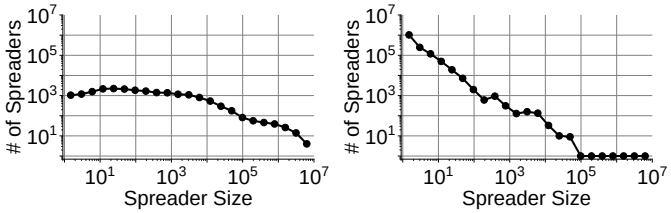
Since the relative standard error of HLL ($\frac{\sqrt{Var(n_s)}}{n_f}$) is $\frac{1.04}{\sqrt{s}}$, the bound of the standard error of RRSE is

$$StdErr(\hat{n_f}) < \frac{\sqrt{Vari(n_s)}}{n_f} + \frac{\varepsilon}{n_f} = \frac{1.04}{\sqrt{s}} + \frac{\varepsilon}{n_f}. \quad \square \quad (19)$$

As described in Theorem 1, the probability of a flow with a rank value i to overwrite another flows' registers is given as $P_n[i] = \frac{C_m[i] - C_s[i]}{m-s}$, thereby the number of registers that were overwritten by rank value i (*i.e.,* noise) follows a binomial distribution $Bino(s, P_n[i])$. Accordingly, the number of overwritten registers is $C_\varepsilon[i] = binoinv(\sigma, s, P_n[i])$, where $binoinv()$ is binomial inverse cumulative distribution function. To this end, we can calculate ε using the HLL's estimation (*i.e.,* Eq. (2)) and $C_\varepsilon[]$ as $\varepsilon = \frac{\alpha_s \cdot s^2}{\sum_{i=0}^{r} C_\varepsilon[i]2^{-i}}$.

IV. Evaluation

In this section, we conduct extensive evaluations of RRSE, including (1) comparing the experimental results of RRSE's performance with our theoretical error bound analysis, (2) evaluating the accuracy of RRSE by varying the memory and traffic distribution (*i.e.,* attack and normal), (3) comparing the performance of RRSE with two state-of-the-art schemes, namely vHLL [27] and MCSE [16]. We finally discuss the cost of RRSE.



(a) Witty Worm Traffic (b) Normal Traffic

Fig. 4: Spreader size distribution of attack and normal traffic

TABLE II: The statistics of datasets

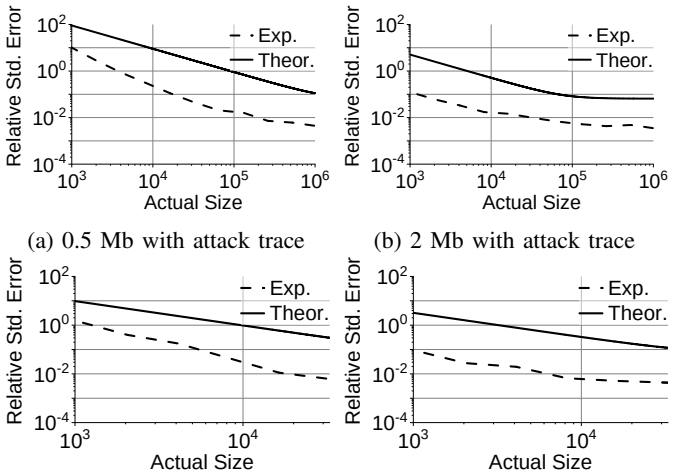
Dataset	Witty Worm Traffic	Normal Traffic
Number of spreaders	20,906	1,470,442
Total connections	192,265,216	16,322,155
Avg spreader size	9,196	11
Largest spreader size	7,266,976	6,859,211

A. Dataset Description

We use both attack and normal traffic datasets in our evaluation. The Witty Worm trace [31] is used as attack traffic, and it has a relatively small number of devices but a large spreader size on average. The normal traffic is an ISP trace [32], where many devices are attached to the network with a relatively small spreader size. Fig. 4 shows the spreader size distribution of two datasets. As shown, the normal network trace follows a heavy-tail distribution, a general case of most networks, while the attack trace has a uniform distribution for different sized groups; alludes to a large number of compromised devices where each of them recursively generates many connections. Table II summarizes the statistical details of the two datasets.

B. Standard Error and Variance

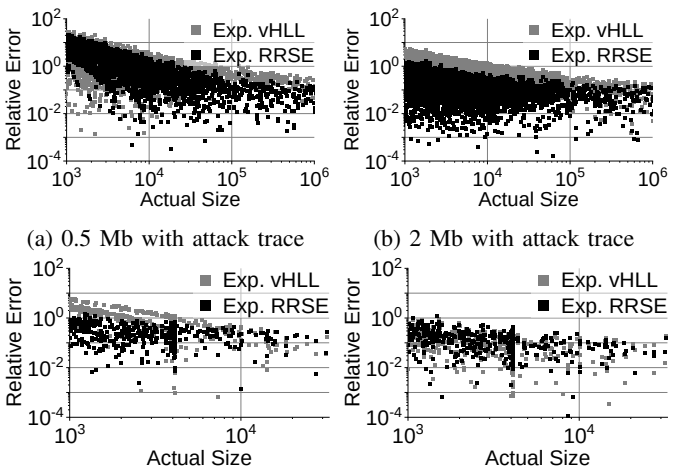
To verify our analysis, we compare the theoretical and experimental relative standard errors (Eq. (14)) of RRSE using 0.5 Mb and 2 Mb memory sizes. As shown in Fig. 5, RRSE's experimental standard error is better than the theoretical error for spreaders greater than 1,000 regardless of traffic distribution and memory usage (*i.e.,* correctness), because the theoretical error bound is not tight. Here, we note that RRSE's estimation is only for medium or high spreaders (*i.e.,* $\geq \frac{2}{5}s$). Similar to HLL, RRSE takes advantage of a linear counting algorithm for low spreader estimation [17]. As shown, we can observe a clear trend where the relative standard errors of RRSE decrease as the memory increases. As shown in Fig. 6, RRSE's accuracy in terms of the absolute relative error ($\frac{|f_i - \hat{f_i}|}{f_i}$) shows a similar trend and is better than vHLL's estimations in different scenarios (*i.e.,* lower is better). Moreover, the variance of RRSE is smaller than vHLL's with less memory. Under the attack traffic that contains many high spreaders, vHLL clearly shows higher estimation errors than RRSE, especially in the high spread range. Moreover, the vHLL's estimations, in the low spread range, show a larger variance than our RRSE. The results suggest that RRSE's noise elimination strategy can precisely remove the noise in the different sized spreader, and performs better than vHLL's strategy that applies a universal noise (value) across regardless of



(a) 0.5 Mb with attack trace (b) 2 Mb with attack trace

(c) 0.5 Mb with normal trace (d) 2 Mb with normal trace

Fig. 5: Comparison: Theoretical and experimental relative standard error of RRSE varying memory usage, and under normal and attack scenarios. See Eq. (14) for theoretical error.



(a) 0.5 Mb with attack trace (b) 2 Mb with attack trace

(c) 0.5 Mb with normal trace (d) 2 Mb with normal trace

Fig. 6: Experimental results: absolute relative error ($\frac{|f_i - \hat{f}_i|}{f_i}$) of RRSE and vHLL varying memory usage, and under normal and attack scenarios.

spreaders' sizes. vHLL's performance becomes stable when a larger memory is given. However, RRSE's estimations achieve a similar variance but better accuracy compared to vHLL, as shown in Fig. 6(b). In the normal traffic scenario, vHLL shows higher estimation errors compared to our RRSE and achieves a similar performance when a larger memory is given Fig. 6(b). We note that vHLL shows the biased estimation because the normal trace used in this work is larger than the vHLL work [27]. Detailed comparisons and discussions will be given in section IV-D.

C. Attack Traffic Scenario: Witty Worm Trace

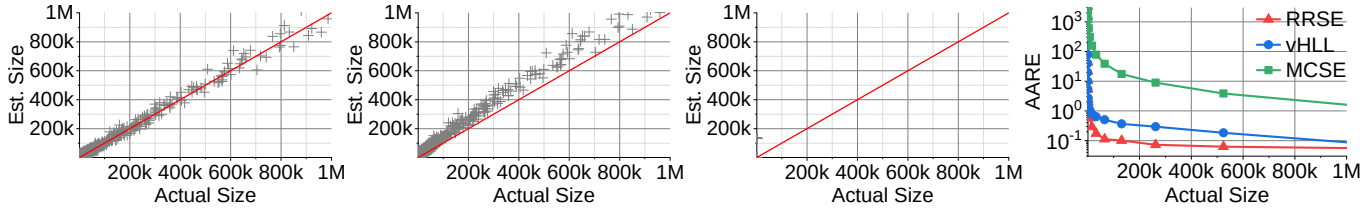
We now simulate an attack scenario using the Witty Worm trace [31] and compare the accuracy of RRSE with two state-of-the-art schemes, namely vHLL [27] and MCSE [16].

Setup. We varied the memory from 0.5 Mb to 4 Mb for the three schemes. Since the largest spreader size of the Witty Worm trace is about 6.8 million (see Table II), RRSE sets $s = 256$ and $b = 4$ to count up to 8.39 million destination IPs for each source IP (*i.e.,* up to $s \cdot 2^r, r = 2^b - 1$). This parameter setting means that each source IP (*i.e.,* potential spreader) can use up to 256 4-bit registers of the global register array, some of which are shared among different spreaders. For fairness, vHLL and RRSE use the same configuration. For MCSE, we set $g = 16$ and $s = 256$ for a sufficient counting range, where g is the number of memory segments and s is the number of bits used in each segment for each spreader.

Results. Fig. 7∼10 shows the accuracy of RRSE, vHLL, and MCSE varying the memory from 0.5 Mb to 4 Mb. In Fig. 7∼10(a)-(c), the x-axis is the actual spreader size (n_f) and the y-axis is the estimated spreader size ($\hat{n_f}$). A guideline $y = x$ is shown to demonstrate bias and variance. An estimation is underestimated when it's below the guideline and overestimated if above the guideline. As shown, RRSE outperforms vHLL and MCSE under an attack scenario when many high spreaders arrive simultaneously. Moreover, while RRSE's estimations are shown to be unbiased with varying memory usage, vHLL tends to overestimate the spreaders (biased), and the amount of the overestimation becomes worse when a smaller memory is given. Our results also show that MCSE has a scalability issue when the average size of the spreaders is large. We note that MCSE estimates spreaders around 3 million with 0.5 Mb of memory and 5 million with 1 Mb of memory, thus the data points are invisible in Fig. 7(c)∼8(c).

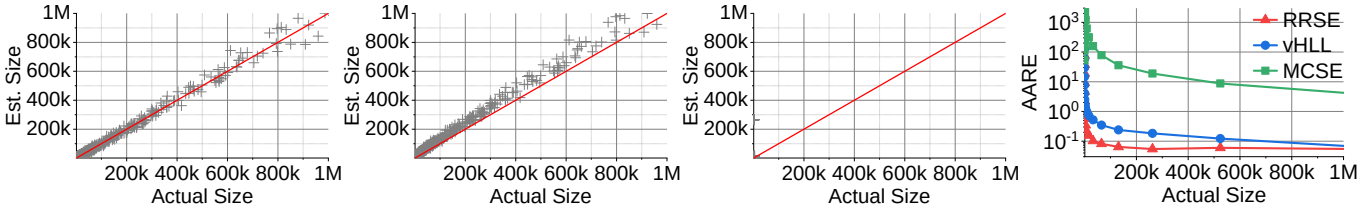
To compare the three schemes, we evaluated in our experiments, we use the average absolute relative error ($AARE = \frac{1}{n}\sum_1^{i=n} \frac{|f_i - \hat{f}_i|}{f_i}$). As shown in Fig. 7(d)∼10(d), RRSE's AAREs are lower than vHLL's AARE and MCSE's AARE for spreaders with different sizes. We note that AARE (y-axis) is shown in a log-scale, which means that a small gap in the AARE value is actually a large estimation gap. For instance, although AARE of vHLL is slightly higher than RRSE, the bias of vHLL's estimations is much larger than RRSE's, as shown in Fig. 7(a)-(b).

Analysis. By design, MCSE divides memory into several small segments and encodes spreaders into each segment independently. As such, MCSE's memory is saturated quickly (*i.e.,* memory efficiency), especially when the traffic includes many high spreaders (e.g., attack traffic). As a result of this saturation, MCSE fails to provide valid decoding results, as shown in Fig. 7(d)∼10(d). As we highlighted in our motivation, vHLL calculates a universal noise from the global average ($\frac{s}{m} \cdot \hat{n}$) and applies it to all different-sized flows. However, as shown in Fig. 1(a), the universal noise of vHLL is insufficient to eliminate the actual noise of high spreaders. That is because **(1)** the massive high spreaders in the attack traffic increase the chance of filling registers with a higher rank value, especially when the number of registers allocated to each flow is big ($s = 256$). Due to the register sharing strategy



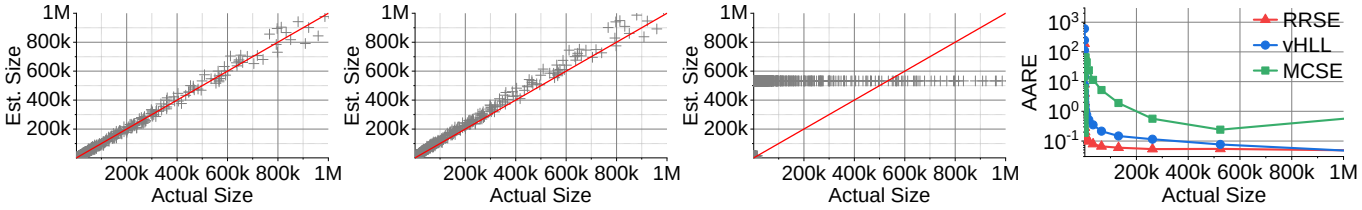
(a) RRSE 0.5 Mb (b) vHLL 0.5 Mb (c) MCSE 0.5 Mb (d) 0.5 Mb AARE

Fig. 7: Attack scenario (Witty Worm trace): Accuracy of RRSE, vHLL, and MCSE with 0.5 Mb memory. In (a)-(c), each data point stands for an individual spreader. The closer the point is to $y = x$, the more accurate the estimation is. When given 0.5 Mb, most of the MCSE's estimations are greater than 3 million; thus, data points are out of the visible range. In (d), the absolute average relative error (AARE) varying the spreader size is shown.



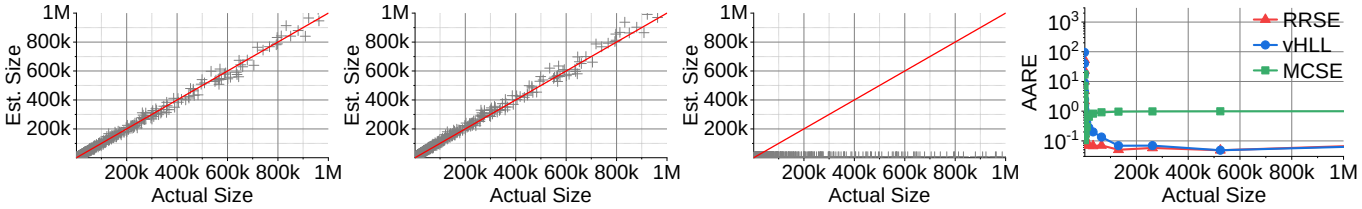
(a) RRSE 1 Mb (b) vHLL 1 Mb (c) MCSE 1 Mb (d) 1 Mb AARE

Fig. 8: Attack scenario (Witty Worm trace) with 1 Mb memory.



(a) RRSE 2 Mb (b) vHLL 2 Mb (c) MCSE 2 Mb (d) 2 Mb AARE

Fig. 9: Attack scenario (Witty Worm trace) with 2 Mb memory.



(a) RRSE 4 Mb (b) vHLL 4 Mb (c) MCSE 4 Mb (d) 4 Mb AARE

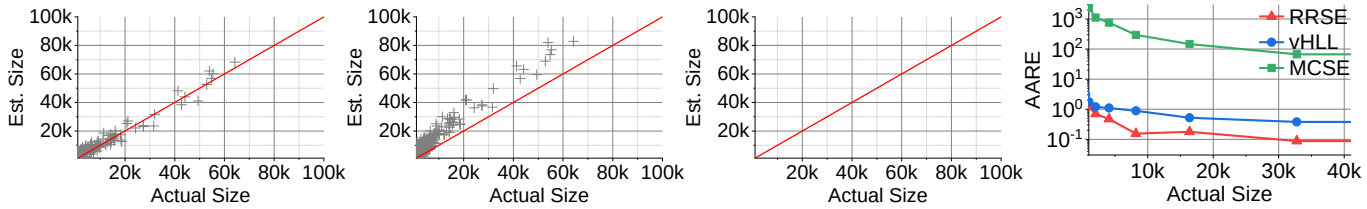
Fig. 10: Attack scenario (Witty Worm trace) with 4 Mb memory.

and the hash collision, overestimation occurs naturally with the local point of view of each spreader, of which evidence can be found in Fig. 7(b)∼10(b). The bias of vHLL becomes smaller when a larger memory is given (*i.e.,* hash collision mitigation). **(2)** Despite the massive number of high spreaders, the majority of the Witty Worm traffic is still small spreaders (Fig. 4(a)), which fill most registers with lower rank values. As a result, the universal noise ($= \frac{s}{m} \cdot \hat{n}$) is smaller than the actual noise of the high spreaders, as shown in Fig. 7(b). Therefore, we conclude that vHLL's approach of estimating a universal noise (*i.e.,* an identical value) is not sufficient to eliminate the noise when the traffic involves a massive amount of medium and high spreaders (*i.e.,* attack scenario). Unlike vHLL that removes an identical noise for all spreaders, RRSE can remove a different amount of noise for an individual spreader by recovering a measurement rank register vector as close as possible to the original vector (*i.e.,Local Rank Distribution Recovery*); thus, the noise estimation of RRSE is more precise than that of vHLL.

D. Normal Traffic Scenario: ISP Trace

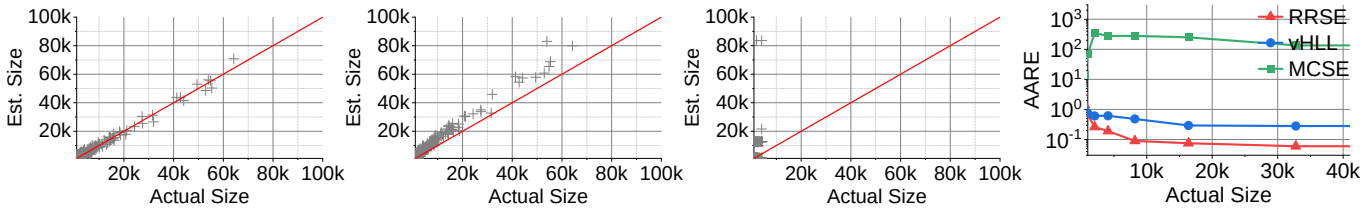
To evaluate the performance of the three schemes under a normal traffic scenario, we repeated the same experiments we had done earlier but using the ISP trace [32].

Setup. We used the same parameters for three schemes as in the attack traffic scenario. We note that the number of spreaders (*i.e.,* source IPs) in the normal traffic scenario is around 70 times more than the attack traffic. Although the largest spreader in the normal traffic remains similar to the attack traffic (*i.e.,* 7.3 million), the average size of spreaders is much smaller (*i.e.,* 11 versus 9K).



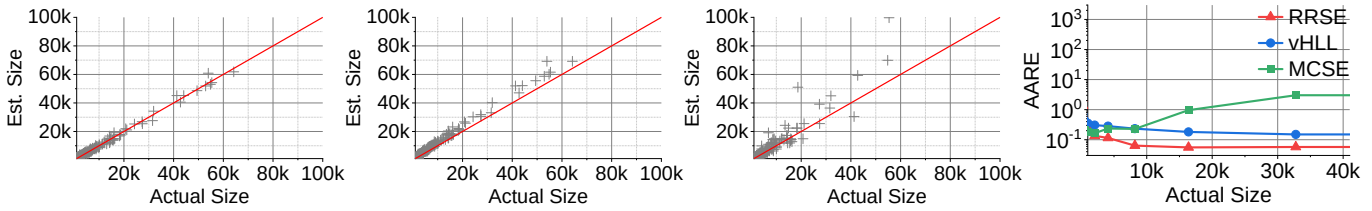
(a) RRSE 0.5 Mb (b) vHLL 0.5 Mb (c) MCSE 0.5 Mb (d) 0.5 Mb AARE

Fig. 11: Normal scenario (ISP trace): Accuracy of RRSE, vHLL, and MCSE with 0.5 Mb memory. In (a)-(c), each data point stands for an individual spreader. The closer the point is to $y = x$, the more accurate the estimation is. When given 0.5 Mb memory, most of the MCSE's estimations are greater than 3 million, thus, data points are out of the visible range. In (d), the absolute average relative error (AARE) varying the spreader size is shown.



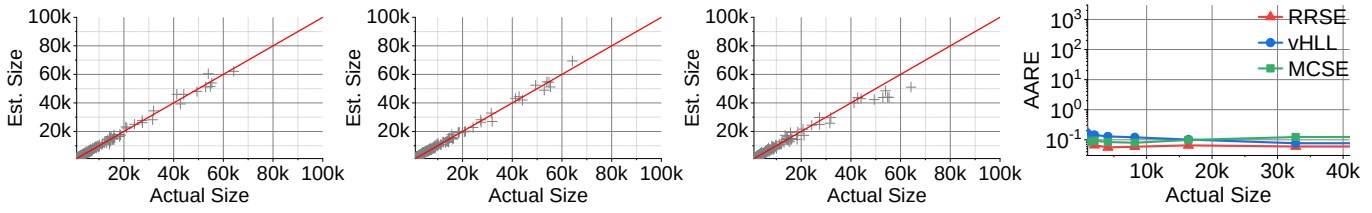
(a) RRSE 1 Mb (b) vHLL 1 Mb (c) MCSE 1 Mb (d) 1 Mb AARE

Fig. 12: Normal scenario (ISP trace) with 1 Mb memory.



(a) RRSE 2 Mb (b) vHLL 2 Mb (c) MCSE 2 Mb (d) 2 Mb AARE

Fig. 13: Normal scenario (ISP trace) with 2 Mb memory.



(a) RRSE 4 Mb (b) vHLL 4 Mb (c) MCSE 4 Mb (d) 4 Mb AARE

Fig. 14: Normal scenario (ISP trace) with 4 Mb memory.

Results. Fig. 11∼14 show the accuracy of RRSE, vHLL, and MCSE varying the memory from 0.5 Mb to 4 Mb. Similar to the attack scenario, RRSE's estimation, regardless of memory usage, is unbiased due to our *Local Rank Distribution Recovery* function that can precisely estimate and eliminate noise for each spreader. However, vHLL shows higher overestimation for high spreaders when smaller memory is given, as shown in Fig. 11(b)∼14(b). *We note that vHLL shows the biased estimation (*i.e., *overestimation) because the network trace used in this work is larger than the trace used in vHLL work [27] (see Table II), while a similar memory is given.* For example, our trace has 11 distinct destination IPs per source IP in average, whereas vHLL's trace has only 2 destination IPs per source IP. Moreover, our trace has 16 million connections in total, whereas the vHLL's trace has around 3 million connections. To these end, our experiments show that vHLL's noise elimination technology produces the biased estimation when measuring a dense network traffic. MCSE starts providing valid decoding results with 2 Mb, although the estimation bias and variance are larger than RRSE, as shown in Fig 13∼14(c)-(d).

Analysis. Through our analysis, we observed that the relatively fewer high spreaders in the normal trace mitigated the memory saturation issue of MCSE when compared with the attack scenario. However, the memory utilization rate remains at a high level, which results in MCSE's inaccurate estimation. For vHLL, the fewer high spreaders in the normal trace mitigated the overestimation (*i.e.,* noise level) compared to the attack scenario. However, the majority of registers are occupied by small spreaders, making the estimated universal noise insufficient to eliminate the actual noise of the medium and high spreaders, as shown in Fig. 1(b).

TABLE III: Spreader detection varying threshold. Settings: attack trace and 2 Mb memory. vHLL's high FPR and zero FNR explain its overestimation, which affects benign users.

Threshold	RRSE		vHLL		MCSE	
	FPR	FNR	FPR	FNR	FPR	FNR
1K	0.043	0.028	0.293	0	0.01	0.035
10K	0.003	0.035	0.046	0	0.058	0.003
100K	0	0.014	0.001	0	0.089	0

Summary. RRSE is shown to be more reliable than vHLL and MCSE with the attack scenario, where the size of spreaders follows a uniform distribution. Moreover, RRSE shows the best performance among the three schemes with normal traffic, where the size of spreaders follows a heavy-tailed distribution. Therefore, per our results, we conclude that our *Local Rank Distribution Recovery* algorithm can precisely eliminate the amount of noise caused by random memory sharing.

E. Use Case: Spreader Detection with a Threshold

We performed a use case of spreader detection using RRSE, vHLL, and MCSE with the Witty Worm trace and varied the detection threshold from 10^3 to 10^6. We used 2 Mb for three schemes and compared the false positive and false negative rates. If the actual size of a reported spreader is smaller than our threshold, we record a false positive (FP); otherwise, or when a spreader is not reported, we record a false negative (FN). We calculate the FP Rate (FPR) and the FN Rate (FNR) using the reported events. Table III shows the detection results, where MCSE provides valid estimations for small spreaders only (*i.e.,* 1K$\sim$10K) with 2 Mb memory, as shown in Fig. 9(c). Moreover, RRSE outperforms vHLL in terms of FPR. However, RRSE has 2.1-3.5% of FNR, whereas vHLL is 0%.

Remark. vHLL achieved 0% of FNR because of its biased estimation (*i.e.,* overestimation), which can be confirmed by its high FPR. With vHLL, many benign users will be misclassified as high spreaders, which is unacceptable for many applications.

F. Query on the Fly: Cost

Next, we discuss the cost incurred by RRSE for providing fast and more accurate estimation when compared with vHLL and MCSE. As shown in Table IV, RRSE's encoding has the highest cost by requiring two additional memory reads and writes per packet. Aside from that, all of the three schemes need two hash computations and a similar amount of logic/arithmetic operations ($\mathcal{O}(1)$). The additional encoding cost of RRSE is consumed for recording the global rank distribution $C_m[]$ on the fly. This, however, allows RRSE to instantly respond to queries while encoding the network traffic (*i.e.,* faster decoding). As shown in Table IV, RRSE reads only $s+r$ registers for decoding, where s is the number of registers assigned for a queried flow and r is the size of $C_m[]$. We note that s usually ranges from 16 to 256 and $r = 7, 15$ to cover sufficiently large estimation ranges (*i.e.,* $s \cdot 2^r$).

Our experiments use $s = 256$ and $r = 15$ to count up to about 8.39 million distinct destinations for a source IP.

TABLE IV: Overhead comparison: encoding and decoding

	Schemes	Reads	Writes	Hashes	Operations
Encode	RRSE	3	3	2	$\mathcal{O}(1)$
	vHLL	1	1	2	$\mathcal{O}(1)$
	MCSE	1	1	2	$\mathcal{O}(1)$
Decode	RRSE	$\mathcal{O}(s+r)$	$\mathcal{O}(1)$	$\mathcal{O}(s)$	$\mathcal{O}(s+r)$
	vHLL	$\mathcal{O}(s+m)$	$\mathcal{O}(1)$	$\mathcal{O}(s)$	$\mathcal{O}(s+m)$
	MCSE	$\mathcal{O}(s.g)$	$\mathcal{O}(g^2)$	$\mathcal{O}(s.g)$	$\mathcal{O}(s.g^2)$

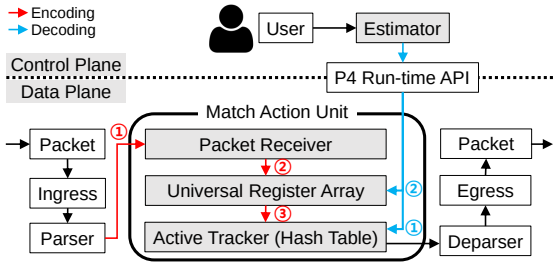


Fig. 15: RRSE-based spreader detection system on P4 switch.

While providing the same counting capacity, vHLL is almost infeasible to respond to queries on the fly, since it has to read m registers, which is the entire memory space (e.g., 0.5 Mb $\sim$ 4 Mb), to estimate noise. Unfortunately, the noise changes over time, therefore vHLL must repeat $\mathcal{O}(s+m)$ memory readings and operations for each query. Also, MCSE's decoding complexity requires applying the maximum likelihood estimation, which makes it infeasible for online decoding.

Remark. The additional memory read and write overheads of RRSE will suppress its performance in a CPU environment. However, in Field Programmable Gate Arrays (FPGAs) or Application-specific integrated circuit (ASIC) environments, these overheads can be significantly relaxed by the fast on-chip memory access.

V. Spreader Detection Framework

We demonstrate the feasibility of RRSE by designing and implementing a spreader detection framework in a programmable switch (Tofino) [33]. Resource consumption and packet processing latency in the data plane are given to show the performance.

Architecture. Fig. 15 depicts the architecture of our spreader detection framework. As shown, our framework consists of three data plane components: *Packet Receiver (PR)*, *Universal Register Array (UAR)* and *Active Tracker (AT)*. These components reside in the data plane of the switch for recording (encoding) packets and detecting high spreaders. When a packet arrives, *Packet Receiver* receives the flow ID from Parser, and then hash the flow ID with CRC32 function to derive a rank value and register indexes (①). The *Universal Register Array*, which resides in SRAM, is responsible for storing the rank value at the designated index and always recording a larger rank value with the register (②). At the same time, the *Active Tracker* will store flows that the rank value is larger than a pre-defined threshold for building a candidate pool of high spreaders. These flow IDs will be further pulled by our last components *Estimator*, which resides

TABLE V: Normalized resource usage of three data plane components: the packet receiver (PR), the universal register array (URA), and the active tracker (AT).

Component	PR	URA	AT	Total
SRAM	0.1%	0.93%	0.41%	1.46%
ALU	0%	2.08%	4.16%	6.25%
Hash	5.5%	4.16%	2.77%	12.44%

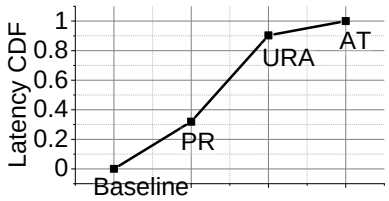


Fig. 16: RRSE's packet processing overhead: a function-wise breakdown of added latencies based on P4 compiler logs: PR, URA, and AT. The baseline is the switch implementation with default routing functions. Our RRSE only adds an insignificant overhead as the switch still has a large room for operating other functions (see Table V).

in the control plane for a precise estimation of the cardinality. To do so, the *Estimator* retrieves flows' rank values with their IDs via a P4 run-time API (①) [34]. Eventually, *Estimator* performs spread estimation using the proposed RRSE.

Resource Usage. Table V shows the additionally required resources (*i.e.,* overhead) for operating our RRSE in a switch's data plane. As can be seen in the table, 1.45% of SRAM, 6.25% of the Arithmetic Logic Unit (ALU, computation unit), and 12.44% of hash power are required for our data plane components. Overall, our framework adds only insignificant overheads to the standard data plane functions. Moreover, it is worth mentioning that our framework does not affect the switch's maximum packet processing speed since the switch still has a large room (e.g., resource and computational budgets) for other functions. Therefore, we conclude that our spreader detection framework can detect the high spread at a line rate [35]–[37].

Latency. We further break down the packet processing process in the data plane in terms of latency. The latency analysis is based on P4 compiling logs generated after data plane function deployment. Through the analysis, we show which data plane component contributes the most to the total packet processing latency. Fig. 16 illustrates the accumulated latency of our framework with a function-wise breakdown. As shown, URA contributes 58.33% of the latency out of the total latency contributed by RRSE components, *Packet Receiver* contributes 31.94%, and *Active Tracker* contributes 9.72%. The results indicate that the rank value derivation and register value update require some computations. However, since the switch still has plentiful resources remaining, the total latency added by our spreader detection framework is still in an acceptable range.

To sum up, we conclude that the proposed RRSE is lightweight and feasible for a switch's data plane, which has a strong potential to be used as an in-network security function.

VI. Related Works

The cardinality estimation problem is to count the number of distinct elements in a stream, where scaling up the estimation range without significant computational and memory overheads has been a challenge.

To count the number of distinct elements, a compact data structure is usually used. Linear Counting [17] uses a bitmap to store and remove duplicate elements. Each element is hashed, the corresponding bit is set to one, and the estimation is $\hat{n} = -b \cdot \ln V$, where b is the total number of bits and V is the number of 0's bits. However, the Linear Counting's counting capacity is linear in m. To scale up the counting capacity, a sampling-based technique is used by MultiresolusionBitmap [26] to exponentially decrease an encoding probability by the series $\frac{1}{2}, \frac{1}{4}, \frac{1}{8}$.... Meanwhile, MultiresolusionBitmap combines samples with multiple bitmaps to perform cardinality counting. PCSA [19] has a similar approach but combines the sampling with registers. However, the major drawback of the two schemes is an unstable accuracy. The fundamental issue of the two schemes is memory efficiency, which means they do not work well when counting massive elements or the memory is constrained.

To resolve the issue, LogLog [38] and HyperLogLog [11] compress the memory of each element from r bits to b bits, where $b = \log_2 r$, and r is the number of leading zeros at the end of a hashed value, called rank. Therefore, the memory cost for counting n distinct elements is reduced by $\log_2 \log_2 n$ while having the same estimation range of 2^r. HyperLogLog is simple yet powerful. Its relative error is $1.04/\sqrt{m}$ and needs $\mathcal{O}(\varepsilon^{-2} \log \log n + \log n)$, where m is the number of registers and n is the maximum estimation. In practice, HyperLogLog is shown to be superior to other practical approaches such as CSE [15], MultiresolusionBitmap [26], or Linear Counting [17].

Driven by the increasing complexity of networks, multi-tenant cardinality counting has emerged and gained interest. The state-of-the-art works in this space are 1) vHLL [27], which is an extension of HLL, and 2) Multiple CSE [16], which is an extension of CSE [15]. They both use a random memory sharing technique but at different levels: register and bit. The major challenge in this domain is to eliminate noise caused by memory sharing, which motivates our work.

VII. Conclusion

In this paper, we proposed a novel noise elimination technique for the random memory sharing-based multi-tenant HyperLogLog. Our solution, called RRSE, is shown to be superior to the prior works, supported by theoretical proof and extensive experiments. Moreover, RRSE provides a reliable estimation under normal and attack traffic scenarios. To show its feasibility, we implemented RRSE on a programmable switch and showed a use case of threshold-based spreader detection. We believe this work opens a new direction in addressing noise reduction for multi-tenant HyperLogLog and will inspire further developments in sketch-based designs.

ACKNOWLEDGMENT

This research was supported by the Global Research Laboratory (GRL) Program through the National Research Foundation of Korea (NRF) funded by the Ministry of Science and ICT (NRF-2016K1A1A2912757), by the National Research Foundation of Korea (NRF) grant funded by the Korea government (MSIT) (NRF-2020R1A2C2009372), and by the Ewha Womans University Research Grant of 2020 (1-2020-0311-001-1). DaeHun Nyang is the corresponding author.

REFERENCES

[1] World internet statistics. [Online]. Available: https://www.internetworldstats.com/stats.html

[2] R. Jang, D. Min, S. Moon, D. Mohaisen, and D. Nyang, "Sketchflow: Per-flow systematic sampling using sketch saturation event," in *Proc. IEEE INFOCOM 2020*.

[3] G. Cormode and S. Muthukrishnan, "An improved data stream summary: the count-min sketch and its applications," *J. Algorithms*, vol. 55, no. 1, pp. 58–75, 2005.

[4] R. Jang, S. Moon, Y. Noh, A. Mohaisen, and D. Nyang, "Instameasure: Instant per-flow detection using large in-dram working set of active flows," in *Proc.of IEEE ICDCS 2019*.

[5] D. Nyang and D. Shin, "Recyclable counter with confinement for real-time per-flow measurement," *IEEE/ACM Trans. Netw.*, vol. 24, no. 5, pp. 3191–3203, 2016.

[6] X. A. Dimitropoulos, P. Hurley, and A. Kind, "Probabilistic lossy counting: an efficient algorithm for finding heavy hitters," *Computer Communication Review*, vol. 38, no. 1, p. 5, 2008.

[7] Y. Lu, A. Montanari, B. Prabhakar, S. Dharmapurikar, and A. Kabbani, "Counter braids: a novel counter architecture for per-flow measurement," in *Proc. of ACM SIGMETRICS 2008*.

[8] Y. Lu and B. Prabhakar, "Robust counting via counter braids: An error-resilient network measurement architecture," in *Proc. of IEEE INFOCOM 2009*.

[9] P. Lieven and B. Scheuermann, "High-speed per-flow traffic measurement with probabilistic multiplicity counting," in *Proc. of IEEE INFOCOM 2010*.

[10] C. Estan and G. Varghese, "New directions in traffic measurement and accounting: Focusing on the elephants, ignoring the mice," *ACM Trans. Comput. Syst.*, vol. 21, no. 3, pp. 270–313, 2003.

[11] P. Flajolet, Éric Fusy, O. Gandouet, and F. Meunier, "Hyperloglog: The analysis of a near-optimal cardinality estimation algorithm," in *Proc. AOFA: The international conference of analysis of algorithms*, 2007.

[12] Q. Zhao, J. J. Xu, and A. Kumar, "Detection of super sources and destinations in high-speed networks: Algorithms, analysis and evaluation," *IEEE J. Sel. Areas Commun.*, vol. 24, no. 10, pp. 1840–1852, 2006.

[13] Q. Xiao, B. Xiao, and S. Chen, "Differential estimation in dynamic RFID systems," in *Proc. IEEE INFOCOM 2013*.

[14] Q. Xiao, Y. Qiao, Z. Mo, and S. Chen, "Estimating the persistent spreads in high-speed networks," in *Proc. IEEE ICNP 2014*.

[15] M. Yoon, T. Li, S. Chen, and J. Peir, "Fit a spread estimator in small memory," in *Proc. of IEEE INFOCOM 2009*.

[16] M. Yoon, T. Li, S. Chen, and J.-K. Peir, "Fit a compact spread estimator in small high-speed memory." *IEEE/ACM Trans. Netw.*, vol. 19, pp. 1253–1264, 2011.

[17] K.-Y. Whang, B. T. Vander-Zanden, and H. M. Taylor, "A linear-time probabilistic counting algorithm for database applications," *ACM Trans. Database Syst.*, vol. 15, no. 2, Jun. 1990.

[18] C. Estan, G. Varghese, and M. E. Fisk, "Bitmap algorithms for counting active flows on high-speed links," *IEEE/ACM Trans. Netw.*, vol. 14, no. 5, pp. 925–937, 2006.

[19] P. Flajolet and G. N. Martin, "Probabilistic counting algorithms for data base applications," *J. Comput. Syst. Sci.*, vol. 31, no. 2, p. 182–209, Sep. 1985.

[20] Z. Bar-Yossef, T. S. Jayram, R. Kumar, D. Sivakumar, and L. Trevisan, "Counting distinct elements in a data stream," in *RANDOM*, vol. 2483, 2002, pp. 1–10.

[21] J. Cao, Y. Jin, A. Chen, T. Bu, and Z. Zhang, "Identifying high cardinality internet hosts," in *Proc. IEEE INFOCOM 2009*, 2019.

[22] F. Giroire, "Order statistics and estimating cardinalities of massive data sets," *Discret. Appl. Math.*, vol. 157, no. 2, pp. 406–427, 2009.

[23] Q. Zhao, J. Xu, and a. Kumar, "Detection of Super Sources and Destinations in High-Speed Networks: Algorithms, Analysis and Evaluation," *IEEE Journal on Selected Areas in Communications*, vol. 24, pp. 1840–1852, 2006.

[24] D. M. Kane, J. Nelson, and D. P. Woodruff, "An optimal algorithm for the distinct elements problem," in *Proc. of ACM SIGMOD-SIGACT-SIGART Symposium on PODS 2010*, J. Paredaens and D. V. Gucht, Eds.

[25] S. Venkataraman, D. X. Song, P. B. Gibbons, and A. Blum, "New streaming algorithms for fast detection of superspreaders," in *Proc. NDSS 2005*.

[26] C. Estan, G. Varghese, and M. Fisk, "Bitmap algorithms for counting active flows on high speed links," in *Proc. ACM IMC 2003*.

[27] Q. Xiao, S. Chen, M. Chen, and Y. Ling, "Hyper-compact virtual estimators for big network data based on register sharing." in *SIGMETRICS*. ACM, 2015, pp. 417–428.

[28] T. Li, S. Chen, and Y. Ling, "Fast and compact per-flow traffic measurement through randomized counter sharing," in *Proc. IEEE INFOCOM 2011*.

[29] T. Li, S. Chen, W. Luo, M. Zhang, and Y. Qiao, "Spreader classification based on optimal dynamic bit sharing," *IEEE/ACM Trans. Netw.*, vol. 21, no. 3, pp. 817–830, 2013.

[30] M. Honarkhah and A. Talebzadeh. Hyperloglog in presto. [Online]. Available: https://engineering.fb.com/data-infrastructure/hyperloglog/

[31] C. Shannon and D. Moore. The caida dataset on the witty worm - march 19-24, 2004,. [Online]. Available: http://www.caida.org/passive/witty/

[32] G. Maciá-Fernández, J. Camacho, R. Magán-Carrión, P. García-Teodoro, and R. Therón, "Ugr'16: A new dataset for the evaluation of cyclostationarity-based network idss," *Comput. Secur.*, vol. 73, pp. 411–424, 2018.

[33] Tofino switch. [Online]. Available: https://www.barefootnetworks.com/products/brief-tofino/

[34] O. N. Foundation, "P4 runtime api." [Online]. Available: https://opennetworking.org/news-and-events/blog/p4-runtime-putting-the-control-plane-in-charge-of-the-forwarding-plane/

[35] R. Miao, H. Zeng, C. Kim, J. Lee, and M. Yu, "Silkroad: Making stateful layer-4 load balancing fast and cheap using switching asics," in *Proc. ACM SIGCOMM 2017*.

[36] T. Yang, J. Jiang, P. Liu, Q. Huang, J. Gong, Y. Zhou, R. Miao, X. Li, and S. Uhlig, "Elastic sketch: Adaptive and fast network-wide measurements," in *Proc. ACM SIGCOMM 2018*.

[37] X. Jin, X. Li, H. Zhang, N. Foster, J. Lee, R. Soulé, C. Kim, and I. Stoica, "Netchain: Scale-free sub-rtt coordination," in *Proc. USENIX NSDI 18*.

[38] M. Durand and P. Flajolet, "Loglog counting of large cardinalities (extended abstract)," in *Proc. of ESA 2003*, ser. Lecture Notes in Computer Science, G. D. Battista and U. Zwick, Eds.