A FIRST LOOK AT THE DECISIONAL WEIGHTED BALLS-INTO-BINS PROBLEM

Arshia Mahajan[1], Chelsea Ling Xinyi[1], Tan Gim Qi Danielle Jeanne[1], Ruth Ng Ii-Yung[2]

[1]Raffles Girls' School (Secondary), 2 Braddell Rise, Singapore 318871
[2]DSO National Laboratories, 20 Science Park Drive, Singapore 118230

Abstract. The classic mathematical Balls-Into-Bins problem measures probabilities associated to randomly throwing a set of balls into a fixed number of bins. Our variant of this problem assigns each ball a unique weight and a maximum weight that each bin can hold. We want to observe the relationship between the probability percentage of the balls in the bins not exceeding the bin's maximum weight and the maximum weight needed to achieve this probability. We conjecture that there is no mathematical closed-form for these values, so we instead simulate the balls being thrown and each one landing uniformly at random into any one of the bins to estimate solutions to the problems above and analyse the trends therein. We then demonstrate a real-world usage of this analysis in improving storage efficiency of volume-hiding cryptographic schemes.

1 Introduction

The Balls-into-Bins problem is a classic problem in probability theory. In this paper, we tackle a new variant of this problem, where the balls have weights and there exists a specific limit to the weight that each bin can hold. We refer to this problem as the Decisional Weighted Balls-Into-Bins Problem (see section 2). Our objective is to study the relationship between this maximum weight and the success probability of each bin not exceeding this maximum weight. In our literature review, we conjecture that a closed-form probability expression is unlikely (see section 2), so we approach the problem via simulations.

Each simulation was run on balls with weights that were generated according to a probability distribution: linear, Zipfian and normal (see section 3). From our results, we conclude that balls with weights according to the Zipfian distribution would require the bins to have the greatest maximum weight in order to succeed, while balls with weights according to the linear distribution require the lowest maximum weight. Furthermore, when we require success rates that approach 100%, the maximum weight that the bins need to accommodate increases at a faster rate.

This variant of Balls-into-Bins is important to study as it can be used to evaluate a cryptographic scheme. Structured encryption is used when storing sensitive data on a cloud server. In this project, we focus on structured encryption of multimaps. Multimaps map labels to tuples of values and allow users to query labels and retrieve the associated tuple of values. In standard structured encryption, when labels are queried, lengths of each tuple are leaked when they are returned. This leakage can be used by adversaries to perform Leakage Abuse Attacks to retrieve the data. Volume-Hiding structured encryption is more secure and prevents this leakage. Volume-Hiding is a subset of structured encryption schemes where a tuple of the same length is returned for every query made. Hence, we introduce 2 such Volume-Hiding schemes: a naive scheme and a colliding scheme that parallels our Balls-Into-Bins problem. We then extend our earlier simulations to compare both schemes in terms of storage savings. Our results concluded that the colliding scheme saves storage for lower success probabilities but works best for Zipfian distributions, which models real-world data such as the frequency distribution of words in a text. The success rates in all experiments were generally inversely proportional to storage space. Thus, in both the naive and colliding schemes, users have to consider the trade off between success rate and storage efficiency.

2 Decisional Weighted Balls-into-Bins Problem

In the (**b**,m,W) Decisional weighted Balls-into-Bins problem (which we will abbreviate to the (**b**,m,W)-problem), W represents the maximum weight that each bin can hold, **b** represents the respective weights of n balls, and m represents the number of bins. W is a positive integer, $\mathbf{b}=(b_1,\ldots,b_n)$ is a tuple where $1\leq b_i \leq W$ for all $i \in \{1,..,n\}$, and m is a positive integer. In an instance of this problem, the n balls are thrown uniformly at random into m bins. We define the "success" event as when the sum of ball weights in each bin is less than or equal to W after all balls are thrown. We denote $P_{\mathbf{b},m,W}$ = Pr[Success in the (**b**,m,W)-problem].

In applications, it is often more helpful to consider the minimum weight needed to achieve a certain probability of success, p. Therefore, we define $W_{p,\mathbf{b},m}$ to take on this minimum value for a particular p,**b**,m. More precisely, $W_{p,\boldsymbol{b},m} = \min_{p\leq P_{\boldsymbol{b},m,w}} W$.

There is existing literature that handles similar, but simpler, Balls-into-Bins problems. In [1], a similar problem is studied, but with balls of uniform weight, while in [2] weights are non-uniform but there is no bin weight limit (i.e. not "decisional"). In both these works, no simple closed-form probability expressions were found. Thus, we believe that our problem, which is strictly harder than both of these, likely has no closed-form mathematical expression either. Nonetheless, this problem is still important to study due to its applications in cryptography, thus we chose to run simulations instead.

3 Experimentation Methodology

In our experiment, the ball weights are generated in accordance with a probability distribution. To capture this, we define an n-probability distribution as a tuple dist = $(d_1,...,d_n)$, such that $\sum_{i \in\{1,\ldots,n\}} d_i$ = Y. Then we define **b**(Y, dist) as the tuple of integer ball weights. totalling weight Y. which most closely resembles the probability distribution (i.e. $\frac{b_i}{Y} \approx d_i$ for all i). We generated **b**(Y, dist) according to 3 distribution types -- linear, Zipfian, and normal. We wrote Python programs to generate the data and run our experiments.

The elements i ∊ {1,..,n} in **b** are generated according to the following equations for each dist.

dist	Formula
Linear	$\frac{i+1}{\frac{n(n+1)}{2}}$
Zipfian	$H = \sum_{k=1}^{n} \frac{1}{k}$ $i = \frac{1}{H} \cdot \frac{1}{k}$
Normal	$i = \frac{e^{-\frac{x^2}{2}}}{\sqrt{2\pi}}$ where $j = \frac{3(i-1)}{n-1}$

As the elements in **b** may not be integers, we need to round them and ensure that they still add up to Y. We scale **b** such that $\sum_{i \in b} d_i$ = Y to get **b**(Y, dist). **b**(Y, dist) needs to fulfil 2 conditions – the distribution of **b**(Y, dist) must be as close to the distribution of **b** as possible, and every element in **b**(Y, dist) must be a positive integer (i.e. weight of each ball > 0). Hence, we generate **b**(Y, dist) according Fig. 1, as shown below.

```
let b(Y, dist) = d
scale = $\frac{Y}{\sum_{i=1}^{n} a_i}$
For i=1…n do
    $b_i$ = scale * $a_i$
    $c_i$ = $b_i$ mod 1
    $d_i$ = $\lceil b_i \rceil$
r = $\sum_{i=1}^{n} d_i$ - Y
Let $c_{i_1}, \dots, c_{i_r}$ be the smallest values in c, where $d_{i_1}, \dots, d_{i_r}$ is not 1
For j=1…r do
    $d_{i_j}$ = 1
Return d
```

Fig. 1. Pseudocode of generation of **b**(Y, dist)

We generate datasets and implement the above simulations with Python programs, which throw the balls into bins uniformly at random. After the balls have been distributed, we find the required $W_{p,\mathbf{b}(Y,\ dist),m}$. We repeat the experiment N times for different m and Y. This is done according to Fig. 2, as shown below.

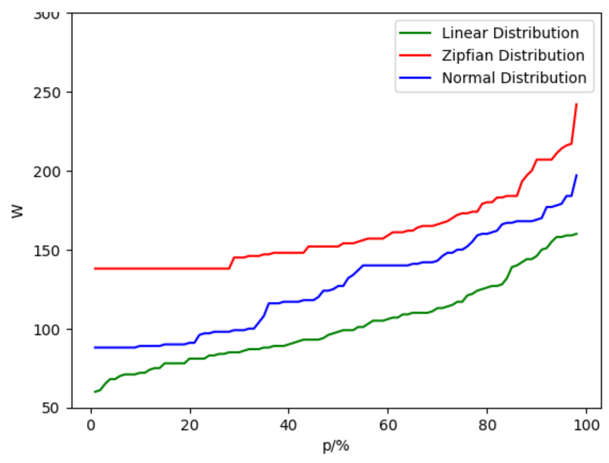
```
For i in b(Y, dist) ∈ {1…N}, do:
        Throw balls into bins
        Store the maximal weight across all bins as $w_i$
Sort $(w_1,...,w_N)$ in increasing order
For p in P_range do
        Set $W_{p,\mathbf{b}(Y,\ dist),m}$ to $w_i$ where $i = \frac{N \times p}{100}$ (rounded up)
Plot the graph of $W_{p,\ \mathbf{b}(Y,\ dist),\ m}$ against p
```

Fig. 2. Pseudocode of experimental steps

We simulated the throwing of balls into the bins by assigning each bin to a unique hash and each ball to a random hash, then we threw the balls into the bin that corresponds to their hash. We chose to use hashes as hashes have randomness and compute faster than random number generators.

3.1 Results of Experiment

In the graphs below, we present the results of our simulations.



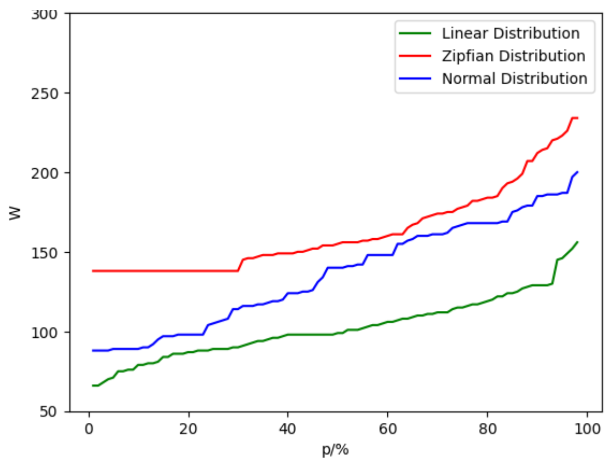


Fig. 3. Graphs of minimum weight of $W_{p,\mathbf{b}(Y,dist),m}$ needed to achieve a success p with 8 (left) and 16 (right) bins and 500 balls weighted with different distributions.

3.2 Discussion of Trends

Observation 1. As seen in Fig. 3, all the graphs of Wp, b(Y, dist), m against p have the same shape: they start increasing linearly before becoming steeper and increasing exponentially as p approaches 100%. The value of Wp,b(Y, dist),m starts to increase at a faster rate as p approaches 100%.

Explanation 1. Considering two extreme scenarios will aid in the intuition behind this trend in maximum bin weight. In the 1st scenario, we want the smallest possible weight of the heaviest bin. For this, the heaviest ball lands in a bin, all other balls land randomly in other bins, and all other bin's individual weights do not exceed the weight of the bin with the heaviest ball (even when there are multiple balls in the same bin). In the 2nd scenario, we want the greatest possible weight of the heaviest bin, this scenario happens when all the balls land in the same bin. In between these 2 scenarios, there are many more combinations of the possible balls that make up the bin with the maximum weight. Since the weight of the heaviest bin is inconsistent, the trend will not be linear. As the success probability increases, the simulation has to account for more of these scenarios and accommodate them with a greater $W_{p,\mathbf{b}(Y,dist),m}$. In addition, the extreme cases where, for example, the heaviest balls are thrown into the same bin, are rarer. When p approaches 100%, it has to accommodate these rarer cases, causing $W_{p,\mathbf{b}(Y,dist),m}$ to have a sharp increase.

Observation 2. The graphs follow a staircase shape. Majority of data points can be grouped into different clusters where the largest bin has the same $W_{p,\mathbf{b}(Y,dist),m}$.

Explanation 2. The same weight is needed to achieve a range of thresholds p, represented by the cluster. A cluster occurs when the combination of balls in the heaviest bin is the same. As the hash domain increases, the width of the staircase increases. This is because the balls collide less frequently, so there are fewer combinations present that make up the longest tuple. For example, for Zipfian distribution, when the number of bins is 16 and Y = 500, the length of the longest tuple is 140 before collisions. The first step of the staircase occurs when W = 140 for $0<p<40$. This shows that for 40% of the cases, the length of the longest tuple was 140 even after collisions.

Observation 3. As seen in Fig. 3, the graph of Zipfian distribution always lies above that of linear and normal. For example, to achieve success probability p = 99%, the bin limits for Zipfian, linear and normal need to be set at 217, 184 and 159 respectively. Hence, Zipfian generally requires higher $W_{p,\mathbf{b}(Y,dist),m}$ to succeed compared to the other distributions.

Explanation 3. For the bin to succeed, $W_{p,\mathbf{b}(Y,dist),m}$ has to be greater than the heaviest ball in the bin. Even without collisions, this heaviest ball is significantly greater in Zipfian than in linear and normal hence requiring a larger value of $W_{p,\mathbf{b}(Y,dist),m}$ to succeed.

Observation 4. As the number of bins increases, $W_{p,\mathbf{b}(Y,dist),m}$ needed for the same p is lower. As seen by comparing the normal distribution lines in Fig. 1, at p = 99%, the normal line on the left is at $W_{p,\mathbf{b}(Y,dist),m} = 279$ while the normal line on the right is at $W_{p,\mathbf{b}(Y,dist),m} = 213$.

Explanation 4. As the total weight of all the balls is constant, having more buckets results in a lower expected weight of the balls in each bucket at the end of the experiment so we would also expect that the weight needed to achieve a certain probability of success is lower.

4 Applications of **b**,m,W-**Problem in Cryptography**

In structured encryption, data is encrypted and uploaded onto a server for storage and subsequent querying. When queries are made, information about the data can be "leaked" to an eavesdropping adversary. One such leakage is the length of the query response. The adversary can then use this leakage to perform Leakage Abuse Attacks [3] to infer these query responses. In this report, focus on volume-hiding [4, 5] multimap encryption schemes, a type of structured encryption which avoids this leakage.

4.1 Definitions

A **multimap** M maps a label of fixed length to a tuple of any length containing values $v_1,...,v_n$ where all labels l are mapped to $(\perp)$ initially. We therefore write $M[l] = (v_1, v_2,...,v_n)$ where n is arbitrary and the v_i are of fixed length.

Cryptographic Hash Functions (CHF), Symmetric Encryption (SE) and Volume-Hiding multimap encryption (VH) are cryptographic primitives, which describe a family of cryptographic schemes with similar properties.
CHF are used to convert arbitrary length input to fixed length output via an evaluation function. Throughout the rest of the paper, we will indicate this evaluation function being called on plaintext PT with a key K using the following notation: CHF.Ev(K, PT) → H

An example of a CHF is HMAC-SHA256 [6], which we use in our Python programs. Notice also that by the pigeonhole principle, collisions exist in CHF outputs and will become more frequent when CHF output length is shorter.

SE consists of 2 algorithms – Encryption and Decryption. Encryption is a randomized algorithm that takes K and an arbitrary length PT and generates a ciphertext as an output. Decryption algorithm takes K and CT as input and generates PT as an output.

Our VH schemes all enable the encryption and querying of multimaps. In particular, we study two VH schemes that "encode" the plaintext multimap to an intermediate multimap in different ways, then uses a common standard technique to encrypt and query them.

The latter technique was given by Cash et. al [7] and involves hashing each multimap label with CHF and encrypting its tuple of values with SE. At query time, the hashed label is used to retrieve values which can be decrypted by the user. This does not innately hide query length leakage, as that is handled in the encoding step. Our two scheme descriptions below will thus focus on this encoding.

4.6 Naive Volume Hiding Scheme

A naive and obvious way to achieve VH would be to use padding. More formally, this "encoding" step just pads all tuples to the length of the longest tuple in the multimap. We will refer to this method of VH as the Naive Volume Hiding Scheme (NVHS).

Original Multimap	
Label #1	(abcdefghijk)
Label #2	(lmnop)
Label #3	(q)

Padded Multimap	
Label #1	(abcdefghijk)
Label #2	(lmnop000000)
Label #3	(q0000000000)

(Padded till all tuples are of the same length – 11 characters)

4.7 Drawbacks of NVHS:

While this method achieves VH, it can be very storage-inefficient. If the longest tuple is significantly longer than the other tuples in the multimap, then all the tuples have to add a significant amount of padding. For example, if the longest tuple is 256-bits long, and all the other tuples are 1-bit long, all the other tuples would have to pad 255-bits, hence significantly increasing the storage space used. To address this issue we introduce a new VH scheme: the Colliding Volume Hiding Scheme that parallels the **b**,m,W-Problem.

4.8 Colliding Volume Hiding Scheme (CVHS)

Our second VH scheme requires a CHF with a small range as it exploits hash collisions between labels. In this “encoding”, if label hashes collide, we concatenate the values of the collided labels together and store them together in the intermediate multimap. We then pad this multimap to a fixed maximum length, which is selected by the user. If the length of any tuple is more than this maximum length, the multimap fails. Ideally, less padding would be required in CVHS than in NVHS as the collided tuple now closer to the length of the longest tuple.

Original Multimap	
Label #1	(abcdefghijk)
Label #2	(lmnop)
Label #3	(q)

Collided and Padded Multimap (Assuming Label #2 and Label #3 collided)	
CHF.Enc(K1, Label #1)	(abcdefghijk)
CHF.Enc(K1 , Label #2)	(lmnopq00000)

5 Comparing VH Schemes using **b**,m,W-**Problem**

In theory, if the collisions occur in an ideal way, i.e. shorter tuples collide while longer tuples remain unaffected, CVHS is able to reduce the amount of padding needed and improve storage efficiency.

We notice that CVHS has parallels to the **b**,m,W-Problem. Instead of throwing balls into bins, our CVHS randomly assigns tuples into labels using hashes and collisions. Similar to how each ball has a weight and each bin has a maximum weight it can hold, tuples of values in the original have different lengths and the intermediate multimap has a parameter for failure. The "success" event can then be redefined as when the length of each tuple in a multimap is less than or equal to this maximum length parameter. Using this parallel, we are able to use our earlier simulations to analyse maximum tuple length against success probability in CVHS. To do this, we follow the same methodology but we let the balls be the tuples of values, let the bins be labels, let the respective weights of the balls be the respective tuple lengths and let W be the maximum length that a tuple can be. In contrast, after colliding and padding the multimap, we encrypt the tuples with structured encryption.

We can then extend the above simulation to compare the strengths of CVHS and NVHS in terms of storage savings by add additional steps to the original methodology: Firstly, we add padding to all the tuples after collisions to make them the same maximum length, W, to achieve VH; Then we measure the percentage of storage savings for CVHS.

$\text{Percentage of Storing Savings} = \frac{\text{NVHS storage} - \text{CVHS storage}}{\text{NVHS storage}} \times 100\%$. From there, graphs that illustrate the percentage of storage savings can be generated.

5.1 Simulation results

The below graphs show the memory savings that a user would get if they used CVHS instead of NVHS with varying number of labels (n) across the 3 types of distributions.

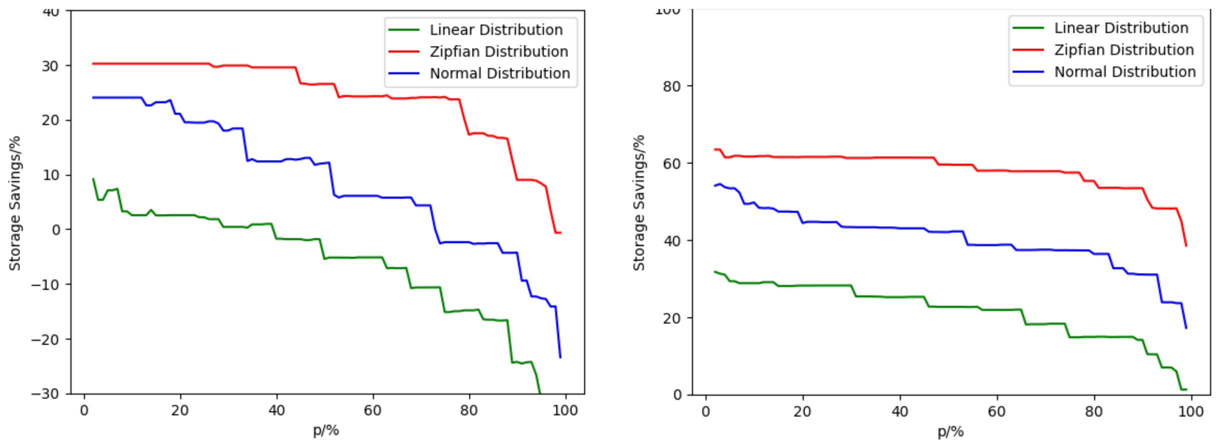


Fig. 4. Graphs of storage savings when using CVHS instead of NVHS with 20 labels (left), 50 labels (right), 16 hashed labels, 500 balls weighted with different distributions.

5.2 Trends and Discussions

Observation 1. CVHS is generally better than NVHS in terms of storage savings. With 20 labels, the % savings from using CVHS instead of NVHS is greater than 0 when p = 98%, 73% and 40% Zipfian, normal and linear respectively.

Explanation 1. The following diagram illustrates the amount of storage space taken up by NVHS (left) and CVHS (right) assuming CHF.Ev(L1) and CHF.Ev(L2) collide.

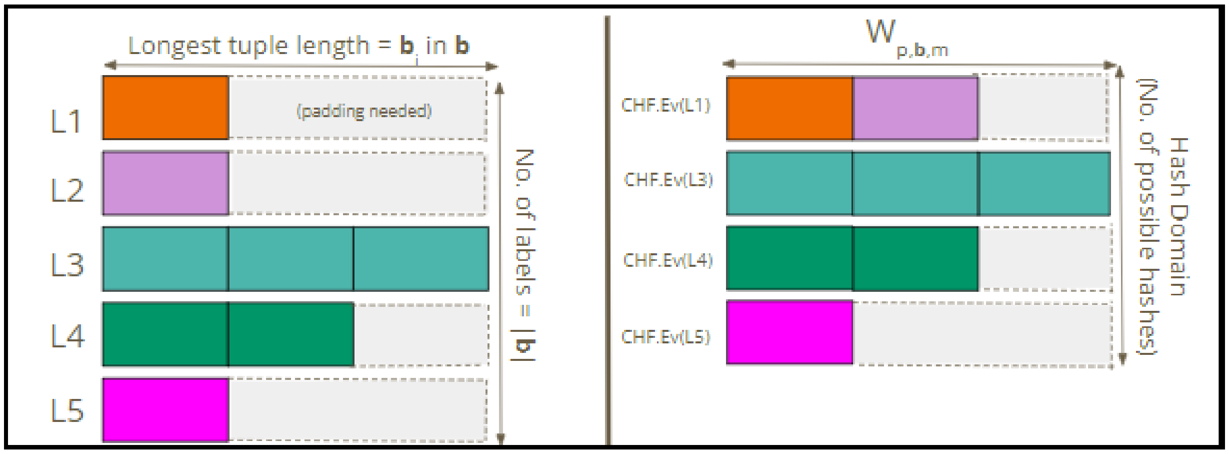


Fig. 5. Diagram of padding required for multimaps (left) and hash domain of labels (right)

Equations for deriving storage space

Storage of NVHS = $|\mathbf{b}| \times$ max. $\mathbf{b}_i$ in $\mathbf{b}$

Storage of CVHS = $W_{p,\mathbf{b},m} \times$ hash domain

As seen by comparing Fig. 4 and Fig. 5, CVHS performs better when the original multimap has a greater number of labels. CVHS saves storage when hash domain in CVHS is less than $|\mathbf{b}|$. The number of labels in the NVHS depends on the initial number of labels set whereas the number of labels in the CVHS depends on the hash domain. When the number of initial labels increases past the size of the hash domain, the CVHS performs better compared to the NVHS. This is especially so when the difference between the hash domain size and the initial number of labels is greater. This is because there are less labels in the end for CVHS hence less storage space is taken up. When the shorter tuples concatenate together, less padding is required to achieve volume hiding, resulting in less storage being used. If the difference is not large enough, there is a sharp decline in the storage savings as p increases from 90% (which might be unfavourable to a user). In general, CVHS are always more storage efficient when p is below 90%.

Observation 2. Using CVHS for Zipfian distribution is most storage efficient followed by normal followed by linear.

Explanation 2. Zipfian distribution has a few large tuples but many small tuples. In the NVHS, all the small tuples would have to be padded up to the longest tuple which would take up a lot of storage. Using the CVHS instead, the small tuples can be concatenated together and less padding is needed to achieve volume hiding. However, for the other distributions, the difference between the longest tuple and the other tuples is not that significant, so the CVHS does not save as much space compared to the Zipfian distribution. The difference between the longest tuple and the other tuples is greatest for Zipfian, followed by normal followed by Linear. This correlates to how CVHS works best for Zipfian followed by normal followed by linear distribution as shown in the graphs.

6 Real-world applications

Our research project is relevant to users who would like to categorise and store data in multimaps and subsequently perform queries on this multimap dataset. For example, encrypting and storing hospital records, bank records or PDF files.

For practical uses, we recommend the CVHS to users so long as the user chooses a suitable hash domain. Hash domain should be determined such that the number of values in the hash domain is smaller than the number of values in the original multimap. For larger datasets. users would have to make a tradeoff between storage efficiency and success probability based on our graphs. Since CVHS with Zipfian distributions showed greater savings at higher success probabilities than other distributions, CVHS would be more beneficial to a user that needs to store data of Zipfian distribution.

7 Future Work and Extensions of our Project

In the future, we could delve into more robust formulae for finding probability success given specific parameters to increase accuracy in results. This would allow us to extend our cryptographic applications to specific real-world cases and parameters for that dataset which would be beneficial for more practical use-cases.

8 Acknowledgements

We would like to thank our mentor, Ruth Ng Ii-Yung for her guidance and unwavering support throughout the project. We would also like to thank Choo Jia Guang for his programming lessons and assistance in designing the simulations.

9 References

[1] “probability of number of balls in a bin” (2020) *Mathematics Stack Exchange.* Retrieved on 21 December 2023 from https://math.stackexchange.com/questions/2839811/probability-of-number-of-balls-in-a-bin

[2] Berenbrink, P, Friedetzky T., Hu Z., Martin R., “On weighted balls-into-bins games” (2008). *Theoretical Computer Science Volume 409, Issue 3, 28 December 2008.* Retrieved on 26 December 2023 from https://www.sciencedirect.com/science/article/pii/S0304397508006713

[3] Naveed, M., Kamara, S., and Wright, C., "Inference Attacks on Property-Preserving Encrypted Databases" (2015). *Computer Science Faculty Publications and Presentations. 153.* Retrieved on 19 December 2023 from http://archives.pdx.edu/ds/psu/20867

[4] Kamara, S. and Moataz, T., “Computationally Volume-Hiding Structured Encryption” (2019). Retrieved on 21 December 2023 from https://www.iacr.org/archive/eurocrypt2019/114760319/114760319.pdf

[5] Patel, S.; Persiano, G.; Yeo, K.; and Yung, M., “Mitigating Leakage in Secure Cloud-Hosted Data Structures: Volume-Hiding for Multi-Maps via Hashing” (2019). Retrieved on 21 December 2023 from https://eprint.iacr.org/2019/1292.pdf

[6] Azeez, N.; Chinazo, O., “Achieving Data Authentication with HMAC-SHA256 Algorithm” (2018) *GESJ: Computer Science and Telecommunications 2018|No.2(54).* Retrieved on 22 December 2023 from https://www.researchgate.net/profile/Nureni-Azeez/publication/332182220_ACHIEVING_DATA_AUTHENTICATION_WITH_HMAC-SHA256_ALGORITHM/links/5ca4f16c299bf1b86d632692/ACHIEVING-DATA-AUTHENTICATION-WITH-HMAC-SHA256-ALGORITHM.pdf

[7] Cash, D., Jaeger, J., Jarecki, S., Jutla, C., Krawczyk, H., Roşu, M-C., & Steiner, M. (2014). “Dynamic Searchable Encryption in Very-Large Databases: Data Structures and Implementation.” *Cryptology ePrint Archive, Paper 2014/853*. Retrieved on 21 December 2023 from https://eprint.iacr.org/2014/853