VOLUME-HIDING DICTIONARY ENCRYPTION: NEW SCHEMES AND BENCHMARKING RESULTS

Jemma Lee Miin Yee[1], Cadence Wern Sea Loh[1], Sheng Yu Fei Carol[1], Ruth Ng Ii-Yung[2]

[1]Raffles Institution (Junior College), 1 Raffles Institution Lane, Singapore 575954
[2]DSO National Laboratories, 20 Science Park Drive, Singapore 118230

ABSTRACT

Structured encryption (STE) schemes allow search queries to be made on an encrypted dataset. The focus of our project is on volume-hiding encryption schemes, which are a subset of STE schemes where the same volume of data is returned for each search query, thus making the scheme more secure. By implementing novel algorithmic and cryptographic techniques, we improved upon 4 encryption schemes[1] from the literature in terms of their query bandwidth and storage. Our experimentation consists of 2 phases: Intra-scheme and inter-scheme comparisons. In the former, we enumerate the improvements on each scheme, and selected between them by comparing their tradeoffs in a few case studies, constructing an improved variant of each scheme. In the latter, we benchmark these new variants to evaluate tradeoffs between storage and query bandwidth. Thus, our work improves upon the current state-of-the-art with novel techniques and uniform benchmarking. Our work is impactful in practical use cases since our new schemes significantly improve storage and bandwidth, and we can firmly recommend our Cuckoo Volume-Hiding (CVH) and Parametrized Volume-Hiding (PVH) variants as the most suitable schemes for Zipfian and linear datasets respectively.

[1] introduced by Kamara, S., & Moataz, T. (2019), Patel, S., Persiano, G., Yeo, K., & Yung, M. (2019), Naveed, M., Prabhakaran, M., & Gunter, C. A. (2014), Chase, M., & Kamara, S. (2010)

1. INTRODUCTION

Encryption has become increasingly important in a digital world that values privacy and convenience. To ensure the security of their confidential databases, clients encrypt their databases and store the encrypted data on an external cloud server, like Amazon Web Services and IBM Cloud.

To make sure that the database can be queried after encryption, structured encryption (STE) schemes are used. When clients send queries to the server, the server will return the relevant data to clients (think of this as searching for a word on a document and being returned all instances of that word). However, data returned for each query may vary in length, leaking query volume in the process. To prevent this leakage, **volume-hiding (VH) structured encryption schemes** are used instead, where the same amount of information is returned for every query. This increases security as this method does not reveal the actual amount of information for each query, and prevents adversaries from running leakage abuse attacks (LAAs) on the encrypted data. Existing literature has introduced several volume-hiding encryption schemes: **Naive VH (NVH)** [KM19]**, Graph VH (GVH)** [NPG14]**, Bucket VH (BVH)** [KM19] **and Cuckoo VH (CVH)** [PPYY19].

1.1 Our contributions

In this project, we optimized existing schemes in literature and compared the improved encryption schemes in terms of their security and storage. Our novel contributions are -

1. Creating new variants of each existing scheme via novel cryptographic techniques to improve schemes' memory and query bandwidth
2. Comparison between improved schemes and recommendations for practical use cases

Our project has successfully improved the existing VH encryption schemes - the improvements introduced have significantly reduced the query bandwidth and storage size of the encrypted data structure (EDS). All the improved schemes are equally secure, but each has a different trade-off in query bandwidth and storage size. Our comparison of the new schemes is also the first of its kind: we found that our improved **Parametrized VH (PVH)** and **CVH** schemes work best for linear and Zipfian datasets respectively.

2. BACKGROUND

We aim to encrypt dictionaries since it is an important data structure used in many large-scale systems. Dictionaries are a collection of label-value pairs: they create associations between a label **L** and value **v** (see Appendix 1 for more details). Structured encryption (STE) schemes encrypt data such that queries can be made. When a dataset is encrypted, total volume of the dataset is leaked. When queries are made, query volume and query equality is leaked (see Appendix 1 for more details). In particular, query volume leakage is the leakage of frequency information when queries are made; this occurs as the length of the value returned for each query is indicative of the length of the value stored. By using volume-hiding, we return a same-length value regardless of what label is queried. This eliminates query volume leakage, resulting in a more secure scheme.

3. INTRA-SCHEME IMPROVEMENTS: DESIGN & EXPERIMENTATION

3.1 Methodology

In this section, we employ a consistent approach to design improved STE schemes: (1) introduce the existing scheme from literature; (2) enumerate novel improvements we considered and; (3) explain choice of improvements via simulation data. For (2), improvements may either be technical improvements or encoding techniques. An example dataset (Fig 2) will be used.

L_1	$v_1 \| v_2 \| v_3$
L_2	$v_4 \| v_5$
L_3	v_6

Fig 2: Unencrypted Dictionary

The schemes were implemented in Python using the following primitives: HMAC-SHA256 to hash labels and AES-CTR-128 to encrypt the values. For clarity, this encryption is left implicit in diagrams and descriptions. For more details on the implementation of the schemes, please refer to Appendix 2.

In-keeping with the literature, our analysis of VH STE involves optimising two metrics:

(1) **Storage** of the encrypted data structure (EDS) refers to the total size of the EDS. We aim to minimize the storage size of the EDS. In order to hide query volumes, volume-hiding encryption schemes employ significant padding and are thus less efficient in memory and bandwidth. often result in larger memory sizes as values have to be padded to a fixed maximum length to conceal the true length of the values from the eavesdropper adversary. In the real world, many databases are extremely large, and the cost of storing data on external cloud servers would increase with increasing memory of the EDS.
(2) **Query bandwidth** refers to the size of the data returned to the client by the server when the client queries a particular label. We aim to minimize the query bandwidth - the greater the query bandwidth, the greater the amount of and the more time-consuming decryption work that the client has to do after receiving the data.
An additional peculiarity of STE schemes is that encryption may sometimes fail. In our work, we make this a controlled variable as real-world use-cases cannot tolerate wide ranges of failure rates.

Intra-scheme experimentation was done on a single dataset whose query length follow a Zipfian distribution.

Current schemes in literature	Improvements	Final scheme	Results: New vs Old Scheme	
			Storage	QB
Naive Volume-Hiding (NVH)	Parametrization; Record length of values	PVH	-78.6%	+15.1%
Greedy Graph Volume-Hiding (GVH)	Graph-matching; Storing used counters	New GVH	-9.2%	-64.0%
Bucket Volume-Hiding (BVH)	Frog-hopping	New BVH	-6.3%	-25.4%
Cuckoo Volume-Hiding (CVH)	3-bit map	New CVH	-9.6%	-19.0%

Fig 3: Summary of the results of our improvements
A negative percentage is preferred as it demonstrates a reduction in storage and query bandwidth.

3.2 Naive VH to Parametrized VH (NVH to PVH)

NVH is parametrized by integer m, the length to pad each query to. This is illustrated in Fig 4.

L_1	$v_1 \| \| v_2 \| \| v_3 \| \| pad$	**Fig 4: NVH EDS stored on server** NVH stores each label with all values associated to it, and is volume-hiding since **m** bits is returned for all queries.
L_2	$v_4 \| \| v_5 \| \| pad$	
L_3	$v_6 \| \| pad$	

(1) Parametrization (Technical improvement)

Our first improvement is to add a new parameter, **h**, to which all hashes (encrypted labels) are truncated to. When hashes collide, we store the values of both labels alongside each other, and employ padding as in NVH. To eliminate false positives when a query is made, we "encode" the values by storing its label alongside. An example is shown in Fig 5.

L_1.truncate(**h**)	$L_1 \| \| v_1 \| \| L_1 \| \| v_2 \| \| L_1 \| \| v_3 \| \| pad$	**Fig 5: PVH EDS stored on server** x.truncate(**h**) means that x has been truncated to **h** bits.
L_2.truncate(**h**) = L_3.truncate(**h**)	$L_2 \| \| v_4 \| \| L_2 \| \| v_5 \| \| L_3 \| \| v_6 \| \| pad$	

(2) Recording number of bits in values (Encoding technique)

Instead of appending a label to the start of every value, we append only one of each label, followed by the number of bits in its corresponding values. For example, Fig 5, the value corresponding to truncated L_2/L_3 would be "$L_2 \| \|$[length of v_4+v_5]$\| \| v_4 \| \| v_5 \| \| L_3 \| \|$[length of v_6]$\| \| v_6$".

(3) Storing the start and end position (Encoding technique)

For each label, the first index of its first value and the last index of its last value is stored in an alternate data structure on the server (e.g. [0,1] for L_2 in Fig 5).

3.2.1 Choice of improvements

We define the parametrized volume-hiding (PVH) scheme to use our "parametrization" and "recording number of bits in values" improvements. "Parametrization" results in a reduction in storage by 72.2% and an increase in query bandwidth by 49.5% from NVH. Since the reduction outweighs the increase, "parametrization" results in overall improvement. "Recording number of bits in values" causes storage and query bandwidth to decrease by the largest percentage (23%), as it does not require an alternate data structure and labels are not repeated.

3.3 Graph VH to new Graph VH (GVH to new GVH)

The GVH scheme[2] works by associating labels with array indices. The array has **g** indices and each label is assigned to a subset of **t** indices. Labels are pseudorandomly assigned to indices, and values are stored at those indices via greedy allocation (i.e. values for a given are assigned to the first empty adjacent indices).

[2] introduced by Kamara, S., & Moataz, T. (2019)

(1) Maximum bipartite matching (Technical improvement)
Instead of greedily assigning indices, maximum bipartite matching will ensure an optimal assignment to minimize failure rate. We express the pseudorandom assignment with bipartite graph **B** and employ a graph matching algorithm to derive a matching **M** which is used to allocate values to indices. An example case, where **g**=7 and **t**=3, is shown in Fig 6 and Fig 7.

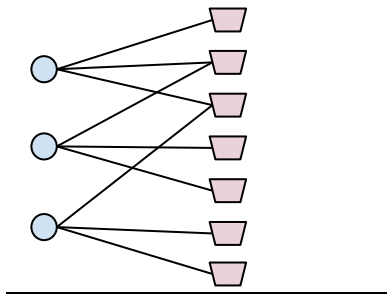

Fig 6: Bipartite graph **B**

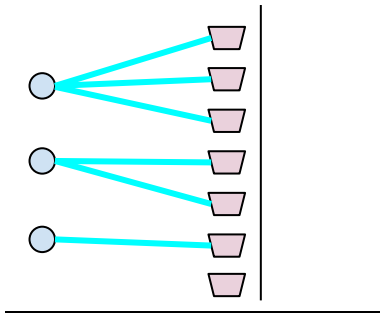

Fig 7: Optimal Graph matching **M**

When L_1 is queried, indices 1, 2 and 3 are returned. **t** edges are returned for every label queried, thus making the scheme volume-hiding.

(2) Storing counters (Encoding technique)
Counters for the used edges are stored in an alternate data structure that is padded to be volume-hiding. In the case of Fig 7, we store L_1::[1||2||3], L_2::[2||3||pad] and L_3::[2||pad].

(3) Storing used edges (Encoding technique)
Used edges are stored in an alternate data structure that is padded to be volume-hiding. In the case of Fig 7, we store L_1::[1||2||3], L_2::[4||5||pad] and L_3::[6||pad].

(4) Bitmap for used edge (Encoding technique)
A bitmap is stored for each label in an alternate data structure. Each bitmap has a bit for each assigned index which is set to "1" if and only if the edge is used in the matching In the case of Fig 7, we store L_1::[1,1,1,0,0,0,0], L_2::[0,0,0,1,1,0,0] and L_3::[0,0,0,0,0,1,0].

(5) Frog-hopping (Encoding technique)
Like a linked-list, each value is accompanied by a pointer informing how many places ahead the next value is. A prepended bit is used to distinguish each list's "head" from all other values.

3.3.1 Choice of improvements

Our new GVH scheme uses the "maximum bipartite matching" and "storing counters" improvements. Maximum bipartite matching is strictly better than greedy matching since it is optimal - using maximum bipartite matching decreases storage size by 4.7% and query bandwidth by 78.6%. "Storing counters" is chosen as it results in the largest decrease in storage from the "naive" method (4.7%), though query bandwidth increases (68.1%).

3.4 Bucket VH to new Bucket VH (BVH to new BVH)

In BVH, there exists **n** buckets, each of size **m (**padding is needed if size<**m).** Each label is allocated slots in **t** buckets where values are stored. Each query returns all values in **t** buckets.

L	d[L]
1	v_1, v_2, v_3
2	v_4, v_5
3	v_6

Fig 8: Unencrypted dictionary

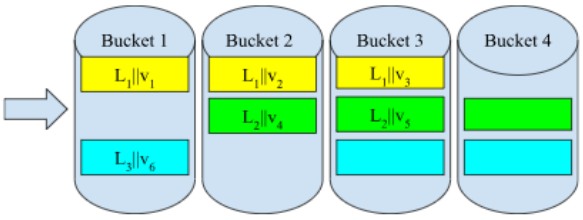

Fig 9: Allocation of slots in buckets (n=4, m=3 and t=3)

Bucket	Values
1	$v_1||v_6||pad$
2	$v_2||v_4||pad$
3	$v_3||v_5||pad$
4	pad

Fig 10: BVH EDS stored on server

(1) Modified bitmap (Encoding technique)

The modified bitmap stores the position of each value in a bucket and "-1" if the value is not in the bucket. In the above example, the bitmap contains L_1::[1,1,1], L_2::[2,2,0] and L_3::[2,0,0].

(2) Frog-hopping (Encoding technique)

L	d[L]
1	V_1, V_2, V_3
2	V_4, V_5
3	V_6

Unencrypted dictionary

Fig 11: Allocation of slots in buckets with frog-hopping

Similar to 3.3.5, but each stored value is instead accompanied by two pointers informing how many buckets and positions ahead the next value is.

3.4.1 Choice of improvements

Our new BVH scheme employs the "frog-hopping" improvement. It is the best encoding since it does not require the use of an alternate data structure, reducing storage size by 6.3%.

3.5 Cuckoo VH to new Cuckoo VH (CVH to new CVH)

In PPYY19, CVH uses a cuckoo data structure (refer to Appendix 2 for more information) to store data. In this cuckoo data structure, there are n rows, and the maximum size of the stash is S.

(1) 3-bit map (Encoding technique)

"0" is stored if the value is found in the left column, "1" if the value is found in the right column and the index if the value is found in the stash.

3.5.1 Choice of improvements

Our new CVH scheme employs the "3-bit map" improvement. It is the best encoding since it results in 9.6% decrease in storage and 19.0% decrease in query bandwidth.

3.6 Summary of results

All the new schemes result in substantial savings in storage and/or query bandwidth due to technical improvements and encoding techniques (Fig 3). Moving forward, when we conduct the inter-scheme comparison, we will be using the improved schemes.

4. INTER-SCHEME BENCHMARKING

4.1 Metrics for inter-scheme comparison

Important metrics in comparing volume-hiding schemes are failure rate, security, storage and query bandwidth. By the definition of volume-hiding schemes, it is noted that each scheme has the same security since the information leaked is the same. As discussed in section 2, all VH schemes carry an inherent failure probability (e.g. in PVH, all labels could hash to the same value). To keep this (relatively) constant, we select parameters which keep failure below 5% in all experiments.

This leaves storage size and query bandwidth as metrics to be evaluated. The schemes were compared on different artificially-generated datasets to compare the tradeoffs between these two metrics, with varying (1) number of labels, (2) total length of labels, and (3) distribution of data. The distribution of data follows either a Zipfian (value length increases exponentially - refer to Appendix 4) or linear distribution (value length increases linearly). NVH is used as a benchmark.

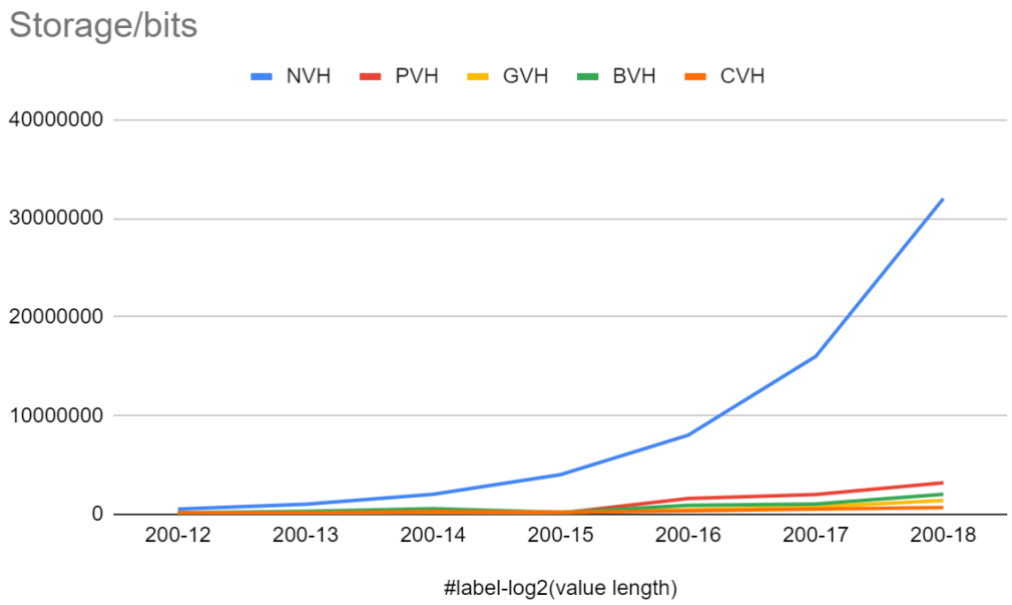

Fig 12: Storage for Zipfian datasets

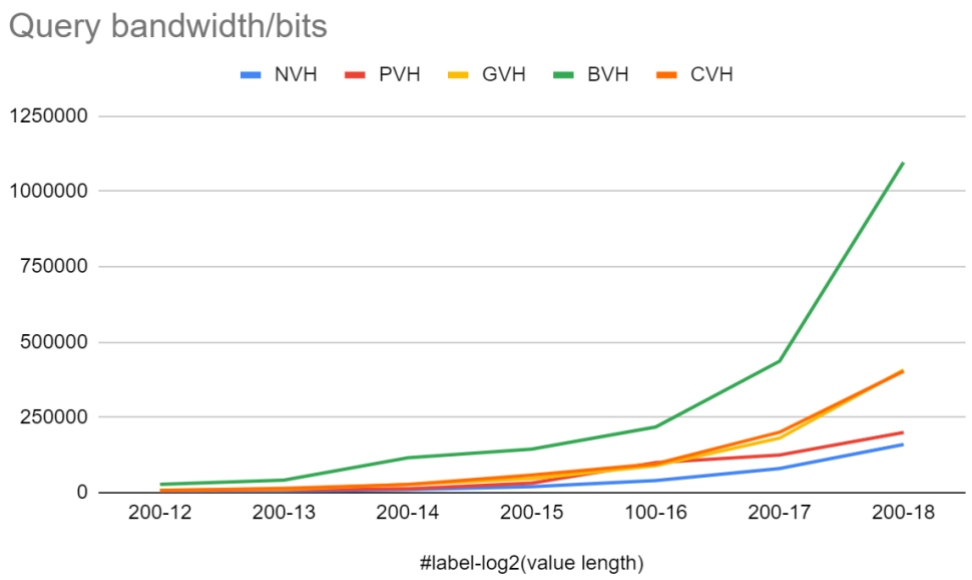

Fig 13: Query bandwidth for Zipfian datasets

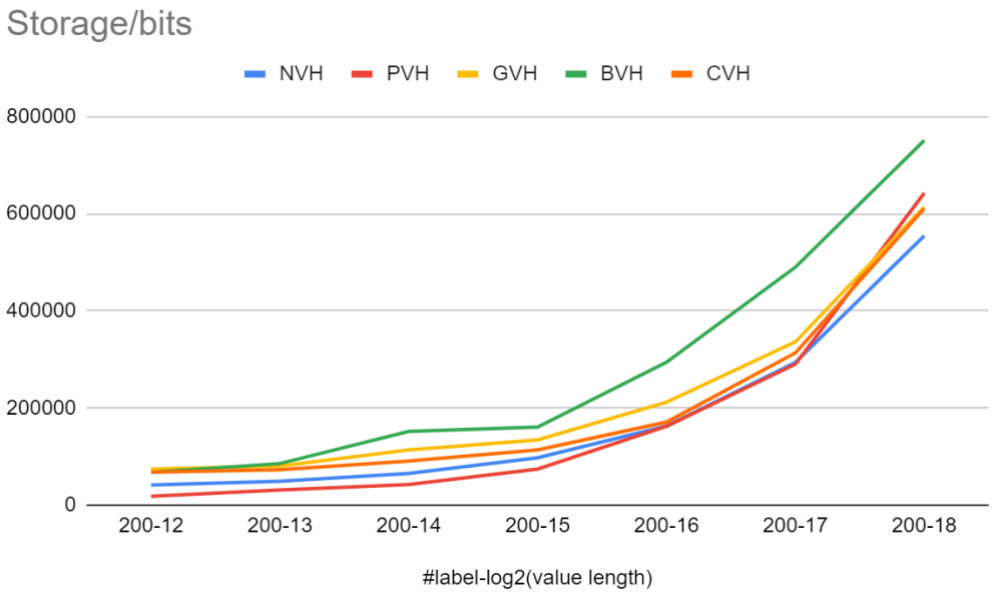

Fig 14: Storage for Linear datasets

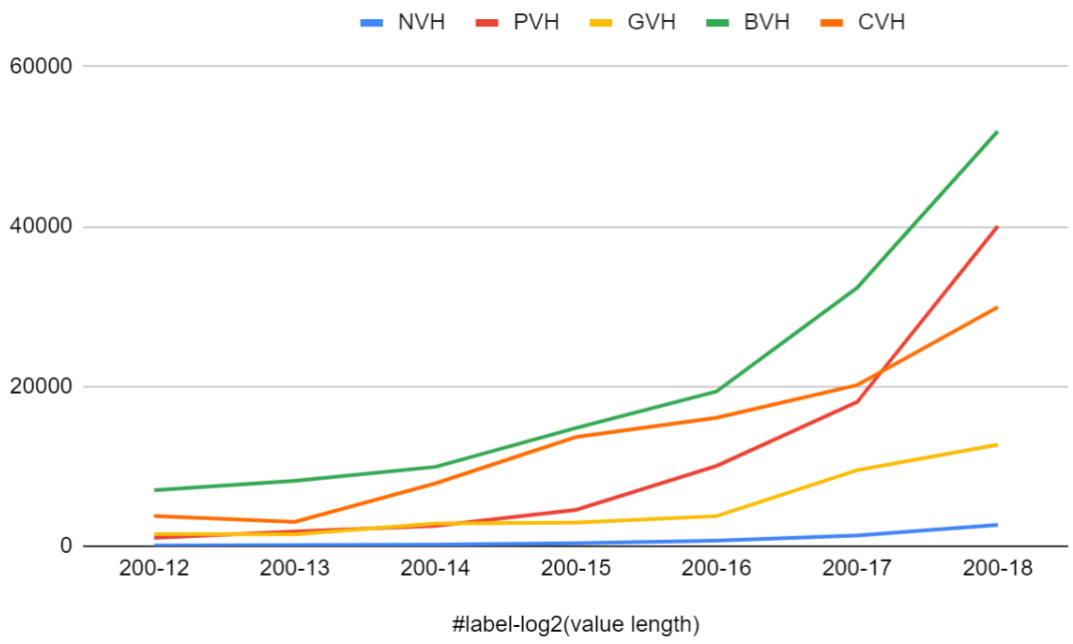

Fig 15: Query bandwidth for Linear datasets

4.2 Storage size

4.2.1 Zipfian dataset

Storage of EDS decreases from **NVH to PVH to BVH to GVH to CVH** (Fig 12).

(a) **The storage size of NVH is the largest** as a large amount of padding is required to pad the shorter values to the same maximum length in a Zipfian dataset. **The storage size of PVH is significantly smaller** as the labels could collide such that shorter values are concatenated and less padding is needed, but there is still a small chance that labels collide such that the longest values are concatenated and more padding is required.
(b) **The storage size of BVH is smaller than that of PVH** since the number of buckets can be increased, reducing the number of values in each bucket and thus padding for each bucket, but **the storage size of GVH is smaller than that of BVH** since parameters are more easily controlled.
(c) **The storage size of CVH is the smallest**, as the EDS in CVH is a cuckoo data structure which takes up less storage compared to other encryption schemes that use a dictionary.

4.2.2. Linear dataset

Storage of EDS decreases from **BVH to GVH to CVH to NVH to PVH** (Fig 14). We note that **NVH and PVH perform much better for the linear dataset** than the Zipfian dataset. This is because much less padding is required for NVH. For PVH, the collision of the largest values will not result in a significantly larger value, hence lowering storage.

4.3 Query Bandwidth

For Zipfian datasets, the query bandwidth of the EDS decreases from **BVH to CVH to GVH to PVH to NVH** (Fig 13). For linear datasets, the query bandwidth of the EDS decreases from **BVH to CVH to PVH to GVH to NVH** (Fig 15).

(a) **The query bandwidth of BVH is the highest** as many values in different buckets are returned.
(b) **The query bandwidth of CVH is slightly lower**. Although the entire stash is returned for each query, there are less values in the rows being returned than the values in buckets returned for BVH.
(c) **The query bandwidth of GVH is lower than that of CVH.** The assignment of values to indices is optimised in GVH so fewer indices have to be returned, but this is not the case for CVH as it returns at least double the size of the maximum value due to CVH returning 2 columns and a stash.
(d) **The query bandwidth for GVH is lower than that of PVH for linear datasets but higher for Zipfian datasets** since the length of the maximum value is not large relative to the size of the linear dataset but the opposite is true for Zipfian datasets. Thus, more array indices need to be returned for GVH for Zipfian datasets and it has a larger query bandwidth.
(e) However, **the query bandwidth of PVH is larger than NVH** as the longest concatenated value length in PVH is mostly always longer than the longest value length in NVH.

4.4 Takeaways

There is a trade-off between storage size of the EDS and query bandwidth - from our experiments, there is no way to minimise both metrics at the same time. This is intuitive since NVH would be used if lower query bandwidth is desired, but it has the highest storage, and storage could be minimised by encrypting all values as a single block, but the entire database has to be returned each time a query is made. From our results, PVH should be used for linear datasets while CVH should be used for Zipfian datasets as storage savings are substantial while increase in query bandwidth is insignificant.

5. CONCLUSION

5.1 Summary

We considered 4 existing schemes in literature: NVH, GVH, CVH and BVH. We then introduced a variety of technical improvements and encoding techniques to improve the query bandwidth and storage size of these schemes and chose the best improvements after some experimentation. Using a few datasets with different distributions and value lengths, we compared these improved encryption schemes. CVH and PVH are the most suitable schemes for Zipfian and linear datasets respectively.

5.2 Future work

We could expand the project by exploring different definitions of security and efficiency (e.g. time efficiency) and fine-tuning our parameters to further optimize the schemes. We could also look at dynamic datasets where information can be added or updated.

6. ACKNOWLEDGEMENTS

We would like to thank our mentor, Ruth Ng Ii-Yung, for her guidance and support throughout our research process and for providing us with inspiration to come up with the various improvements. We would also like to thank Choo Jia Guang for helping us with the programming aspects of the project. Finally, we would like to thank the SP20 interns for their emotional support and friendship during our internship.

7. REFERENCES

[1] Kamara, S., & Moataz, T. 2019. Computationally volume-hiding structured encryption. *Advances in Cryptology – EUROCRYPT 2019*, 183–213. https://doi.org/10.1007/978-3-030-17656-3_7

[2] Patel, S., Persiano, G., Yeo, K., & Yung, M. 2019. Mitigating leakage in secure cloud-hosted data structures. *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*. https://doi.org/10.1145/3319535.3354213

[3] Naveed, M., Prabhakaran, M., & Gunter, C. A. 2014. Dynamic searchable encryption via blind storage. *2014 IEEE Symposium on Security and Privacy*. https://doi.org/10.1109/sp.2014.47

[4] Chase, M., & Kamara, S. 2010. Structured encryption and controlled disclosure. *Advances in Cryptology - ASIACRYPT 2010*, 577–594. https://doi.org/10.1007/978-3-642-17373-8_33

Appendix 1: Definitions

a) Leakage

For all structured encryption schemes, there are three types of leakages:

1. Total volume leakage, where the total size of the dictionary **d** is made known to the adversary. The total volume is leaked from the Encrypted Data Structure (EDS) when the dictionary **d'** is encrypted, as the adversary will be able to see the total number of values encrypted in the dictionary, $\sum_{L \in \{0,1\}^{Llen}} |d'[L]|$
2. Query volume leakage, where the frequency information of values is leaked. When the client performs a search query for a particular label, all values corresponding to that label are returned. The number of encrypted values (from the EDS) returned by the server to the client is leaked to the adversary, and adversary learns |**d'**[L_1]|,...,|**d'**[L_n]|
3. Query equality leakage, where the adversary is able to tell whether pairs of queries made by the client are the same or different. Given queries L_1,...,L_n, the leakage can be captured as a binary matrix X where $X_{i,j}$=1 if and only if L_i=L_j.

b) Dictionary encryption

A dictionary encryption defines two protocols: Encrypt and Lookup

Encrypt (D.Encrypt)	Lookup (D.Lookup)
Client input: Dictionary **d** with labels of a fixed length *Llen* and values of arbitrary length	Client input: *L* (label), K
Server input: ε (epsilon)	Client output (search result): (v_1, … v_n) `or` ⊥
Client output: Key K	Server input: Encrypted Data Structure (**EDS**), bit string of arbitrary length
Server output: Encrypted Data Structure (**EDS**)	Server output: ε (epsilon)

Correctness condition:
Given (K, **EDS**) = D.Encrypt(**d**; ε), for some K, **d**
`Then` ∀ `L D.Lookup(K, L;` **EDS**) = (x; ε) is such that x = **d**[L]

Appendix 2: Pseudocodes

Pad and split functions

Define pad(v, fixed_length):
 # |v| = length of the value
 # fixed_length = multiple length that v should be padded to
```
  v.append(1)
  While |v| mod fixed_length > 0:
    v.append(0)
  return v
```

Define split(v, block_length):
```
  pad(v, block_length)
  split_v ← []
  counter = 0
  while counter < |v|:
    split_v.append(v[counter:counter+block_length])
    counter += block_length
  return split_v
```

Define unpad(v):
```
  While last_digit(v) =/= 1:
    v.remove(last_digit(v))
  v.remove(last_digit(v))
  return v
```

1. Naive Volume Hiding (NVH)

NVH.Encrypt (input **d,** K_L, K_V)

CLIENT

Initialize empty dictionary **d'** and **EDS**
Let $[b](\mathbf{d}) = \{l \in \{0,1\}^* | d[l] \neq \perp\}$
For L∈ $[b](\mathbf{d})$:
```
 x ← len(d[L])
 i ← 0
 K_G = H.Ev(K_L, L)
 While i ≤ x:
   v_L ← d[L][i]
   L_i ← H.Ev(K_G, i) #hash index with label key
   d'[L_i] ← v_L
   i ← i + 1
```

SERVER

For L ∈ {0,1}$^{\|L\|}$ where **d'**[L]≠⊥:
  v_L ← **d'**[L]
  pad(v_L, m)
  EDS[L] ← SE.Enc(K_V, **d'**[L])
 	
client.send(**EDS**)
 	
NVH.Lookup (input L, K_L, K_V)
CLIENT
i ← 0
ret ← []
L' ← H.Ev(K_L, L)
client.send(L')
 	
client.receive(**EDS**[L'])
v ← unpad(SE.Dec(K_v, **EDS**[L']))
 	
return v
	
	server.receive(**EDS**)
	
	server.receive(L')
	server.send(**EDS**[L'])

2. Parameterized Volume Hiding (PVH) **PVH(m, h)** m is size that all values are padded to h is length of the hash output	

PVH.Encrypt (input **d,** K_L, K_V, m, h**)**
CLIENT
 	
Initialize empty dictionary **d'** and **EDS**
Let $[b](\mathbf{d}) = \{l \in \{0,1\}^* \mid d[l] \neq \perp\}$
For L∈ $[b](\mathbf{d})$:
  v_L ← **d**[L]
  L ← H.Ev(K_L, L) *#hash label with label key*
  L ← L.truncate(h) *# truncate label*
  If **d'**[L] ≠ ⊥:
  d'[L].append(L||v_L)
  Else:
  d'[L] ← L||v_L
 	
For L ∈ {0,1}$^{\|h\|}$ where **d'**[L]=/=⊥:
  v_L ← **d'**[L]
  If $v_L \leq m$:
  pad(v_L, m)
  Else:
	SERVER

Error **EDS**[L] ← SE.Enc(K_V, **d'**[L]) client.send(**EDS**) **PVH.Lookup** (input L, K_L, K_V, h) ret ← [] i ← 0 L' ← H.Ev(K_L, L).truncate(h) client.send(L') client.receive(**EDS**[L']) D ← unpad(SE.Dec(K_V, **EDS**[L'])) For x in len(D): L_x||V_x ← D[x] if L_x == L: ret.append(V_X) end while return ret	server.receive(**EDS**) server.receive(L') server.send(**EDS**[L'])
3. Graph Volume Hiding (GVH) There are two different ways to assign values to indices. The first method is using the greedy algorithm (highlighted in green) and the second is using the maximum bipartite matching algorithm (highlighed in yellow).	

GVH.Encrypt (input **d,** K_V, K_L, t, g**)**
CLIENT
Initialize empty dictionary **d'** and **EDS**
 	
E ← [] #edges
V ← [] #values
Let $[b](\mathbf{d}) = \{l \in \{0,1\}^* \mid d[l] \neq \perp\}$
For L∈ $[b](\mathbf{d})$:
For i in len(**d**[L]): **d**[L][i] ← L||**d**[L][i]
K_G ← H.Ev(K_L, L)
B ← $[0]^g$ # to store visited edges
i ← 0
counter ← 0
while counter < t:
K_N ← H.Ev(K_G, i)
K_N ← K_N.truncate(log(g))
i ← i + 1
if B[K_N] == 1: continue
else:
B[K_N] = 1
	SERVER

counter ← counter + 1 For i in len(**d**[L]): E.append(B) V.append(**d**[L]) res ← max_bipartite_matching(E) If res == ⊥: Error For L ∈ [*b*](**d**): If 1 in E and E[1] = ⊥: res[i] ← E.index(1) For i in len(res): **EDS**[res[i]] = SE.Enc(K_v, V[i]) For i in range(g): If **EDS**[i] == ⊥: **EDS**[i] = SE.Enc(K_V, 0) client.send(**EDS**)	
GVH.Lookup (input L, K_V, K_L, t, g) K_G ← H.Ev(K_L, L) client.send(K_G, t, g)	server.receive(**EDS**)

	server.receive(K_G, t, g)
	S ← []
	B ← [0]^g # to store visited edges
	i ← 0
	counter ← 0
	While counter < t:
	K_N ← H.Ev(K_G, i)
	K_N ← K_N.truncate(log(g))
	i ← i + 1
	if B[K_N] == 1: continue
	else:
	B[K_N] = 1
	counter ← counter + 1
	end while
	For i in len(B):
	If B[i] == 1: S.append(**EDS**[i])
	server.send(S)

client.receive(S) ret ← [] For i in len(S): S[i] ← SE.Dec(K_V, S[i]) L_i‖V_i ← S[i]	

If L_i == L: ret.append(S[i]) Return ret	
4. Bucket Volume Hiding (BVH)	

BVH.Encrypt (input: **d**, K_V, K_L, n, m, t)
CLIENT
Initialize empty dictionary **d'** and **EDS**
 	
B ← [] # buckets
V ← [] # values
Let $[b](\mathbf{d}) = \{l \in \{0,1\}^* \mid d[l] \neq \perp\}$
For L∈ $[b](\mathbf{d})$:
For i in len(**d**[L]): **d**[L][i] ← L||**d**[L][i]
K_G ← H.Ev(K_L, L)
A ← $[0]^n$ # to store visited buckets
i ← 0
counter ← 0
while counter < t:
K_N ← H.Ev(K_G, i)
K_N ← K_N.truncate(log(n))
i ← i + 1
if A[K_N] == 1 and len(B)>m: continue
else:
A[K_N] = 1
counter ← counter + 1
For i in len(**d**[L]): B.append(A)
V.append(**d**[L])
 	
If res == ⊥: Error
For i in len(res):
EDS[res[i]] = SE.Enc(K_v, V[i])
For i in range(n):
If **EDS**[i] == ⊥:
EDS[i] = SE.Enc(K_V, 0)
client.send(**EDS**)
 	
BVH.Lookup (input L, K_V, K_L, n, m, t)
K_G ← H.Ev(K_L, L)
client.send(K_G, n, m, t)
	SERVER
	
	server.receive(**EDS**)

	server.receive(K_G, n, m, t)
	S ← []
	A ← $[0]^n$ # to store visited buckets
	i ← 0
	counter ← 0
	While counter < t:
	  K_N ← H.Ev(K_G, i)
	  K_N ← K_N.truncate(log(n))
	  i ← i + 1
	  if A[K_N] == 1 and len(B)>m: continue
	  else:
	  A[K_N] = 1
	  counter ← counter + 1
	  For i in len(**d**[L]): B.append(A)
	  V.append(**d**[L])
	end while
	
	For i in len(A):
	  If B[i] == 1: S.append(**EDS**[i])
	server.send(S)

client.receive(S)
ret ← []
For i in len(S):
  S[i] ← SE.Dec(K_V, S[i])
  $L_i \| V_i$ ← S[i]
  If L_i == L: ret.append(S[i])
Return ret	

5. Cuckoo Data-structure Volume Hiding (CVH)

A data structure is a storage that is used to store and organize data. Examples of data structures are dictionaries (which we have mentioned throughout the above parts and used for the other volume hiding encryption schemes). Dictionaries can be converted into another data structure known as the cuckoo data structure. Due to its rather complex nature, we will not be describing its features in full - please refer to this paper for more information.

To provide a brief introduction to the cuckoo data structure: In a dictionary, each key corresponds to one label. On the other hand, the cuckoo data structure has multiple rows, with each row corresponding to two values. Each label in the dictionary can be hashed twice and the hash output determines the possible position of the corresponding value: each value in the dictionary has two possible positions in the cuckoo data structure. If the first hash output of a newer label collides with a previous one, the cuckoo hashing imitates the real-life cuckoo bird by "pushing out" the previous hash output. The previous label is hashed a second time to obtain a hash output in its second possible position of the cuckoo dictionary. This cycle continues, and it succeeds when each value can be stored in either one of the two possible positions in the cuckoo data-structure, and fails when it results in an infinite loop.

In the case of a failure, the values that are unable to be stored are placed in a stash. The stash box is padded to a size S. During a lookup, values in the stash are also returned. To prevent leakage, the stash is padded.

In this cuckoo data structure, there are **n** number of labels, and the block-size of the values is **m**. The size of the stash is **S**.

Dictionary

Label	Value	Hash 1	Hash 2
x_1	y_1	1	4
x_2	y_2	2	3
x_3	y_3	4	3
x_4	y_4	2	5
x_5	y_5	4	2

Cuckoo data-structure

Label	**Value 1**	**Value 2**	**Stash box**
1	**y_1**	**pad**	**y_2‖pad**

2	~~**y_2**~~ **y_4**	**pad**	
3	**pad**	~~**y_2**~~ **y_3**	
4	~~**y_3**~~ **y_5**	**pad**	
5	~~**pad**~~	**pad**	

Appendix 3: Intra-scheme results

As a benchmark, the query bandwidth of NVH is 10063.92 bits and the storage size is 1018288 bits.

PVH

	Appending labels to each value	**Recording number of bits in values**	**Storing start and end position**
Storage size/bits	241699.84	186119.1667	214095.5789
Query bandwidth/bits	15048	11544	12048

Both "recording the number of bits in values" and "storing start and end positions of values" are improvements upon just appending labels to each value in terms of storage size and query bandwidth. The storage size when "storing the start and end indexes of values" is 15.0% larger than that when "recording the number of bits in values" as it requires the introduction of an alternative data structure, which is padded to make it volume-hiding. The indices returned during querying also means that it has a slightly higher query bandwidth (4.3%). Thus, we choose "recording number of bits in values" as the most optimal encoding technique as (1) it has a smaller storage size and (2) it reduces query bandwidth.

GVH

	Appending labels to values	**Store counter**	**Store edges**	**Edges bit-map**	**Frog-hopping**
Storage size/bits	168555.76	160611.92	186918.08	294310.08	165902.24
Query bandwidth /bits	21692.24	36470.64	39774.32	39775.04	38446.4

"Storing counters" results in the largest savings in storage size. This is because storing counters has a smaller storage size than storing edges, as counters are likely to be smaller than edges. The maximum number of used edges (which is the size of each tuple in "storing counters") is also smaller than the total number of array indices (which is the size of each tuple in the bitmap). Thus, we think that storing the counters is the most suitable encoding technique for GVH.

BVH

	Appending labels to values	**Modified bitmap**	**Frog-hopping**
Storage size/bits	384675.2	5620602.88	360626.96
Query bandwidth/bits	88639.68	111610.8	112193.52

The storage size of "modified bitmap" is over 15 times larger than that of the "frog-hopping" encoding technique due to the presence of the alternative dataset, while the query bandwidth of "frog-hopping" is only 0.5% larger than that of modified bitmap – this is because "frog-hopping" returns both the label and two Δ values for each value when a label is queried, while the modified bitmap only returns values. The size of the alternative dataset (i.e. the modified bitmap) is $l \times m \times n$, while the increase in storage size due to appending of labels is $l \times block\ size$. We choose "frog-hopping" as the most optimal encoding technique as the tradeoff in terms of query bandwidth is smaller than the tradeoff in storage size of the modified bitmap is used.

CVH

	Appending labels to values	**3-bit map**
Storage size/bits	154249.2	139553.2
Query bandwidth/bits	23992	19448.08

Appendix 4: Zipfian data distribution

Zipf-distributed multi-maps. To get a concrete bound on the number of truncations, we have to make an assumption on how the response lengths of the multi-map are distributed. Here, we will assume that they are distributed according to the Zipf distribution which is a standard assumption in information retrieval [14,43]. We note that our analysis can be extended to any power-law distribution. More precisely, we say that a multi-map MM is $\mathcal{Z}_{a,b}$-distributed if its r^{th} response has length

$$\frac{r^{-b}}{H_{a,b}} \cdot N$$

where $N \stackrel{def}{=} \sum_{\ell \in \mathbb{L}} \#\mathsf{MM}[\ell]$ is the volume of MM and $H_{a,b}$ is the harmonic number $\sum_{i=1}^{a} i^{-b}$. Throughout, we will consider multi-maps that are $\mathcal{Z}_{m,1}$-distributed where $m = \#\mathbb{L}_{\mathsf{MM}}$. From this assumption, it follows that the set of all response lengths is

$$L = (L_1, \ldots, L_m) = \left(\frac{N}{1 \cdot H_{m,1}}, \ldots, \frac{N}{m \cdot H_{m,1}} \right),$$

https://www.iacr.org/archive/eurocrypt2019/114760319/114760319.pdf