NEW FRAMEWORKS FOR ANALYSING SECURITY-EFFICIENCY TRADEOFFS IN RANGE SEARCHABLE ENCRYPTION

Soh Jun Heng[1], Ang Wei Sheng Wilson[1], Elton Ng Yew Tien[1], Ruth Ng Ii-Yung[2], Phang Yan Feng Benito[2]

[1]Catholic High School, 9 Bishan Street 22, Singapore 579767
[2]DSO National Laboratories, 12 Science Park Drive, Singapore 118225

Abstract. Range Searchable Encryption (RSE) is a way to secure sensitive data on a third-party server (cloud) while allowing searches over said data. Many RSE schemes exist for potential use, each with inherent trade-offs in security and efficiency. However, with no existing benchmarks for direct comparison, schemes are likely applied ineffectively, potentially hurting system performance.

This work seeks to solve this issue, by recommending optimal schemes for use through standardising benchmarks to evaluate the security-efficiency trade-offs of various schemes. We applied this to different, state-of-the-art schemes for comparison. The evaluated schemes are consolidated from existing literature and are: Constant.RSE, Quadratic.RSE, Logarithmic.RSE and Augmented.RSE.

1 Introduction

Outsourcing data storage to a third-party server, commonly known as cloud storage, is a popular option today due to the ease of data access and cost advantage. For users storing sensitive information, like health records or employee salaries, it is important to hide the contents of their data from the cloud service provider and attackers eavesdropping on the network. Hence, RSE is important to meet this need. RSE allows users to securely store confidential data on the cloud while being able to retrieve data by performing range searches.

In existing literature, many RSE schemes were devised [1] [2], each with tradeoffs in efficiency and security based on scheme design. However, the absence of direct metrics to evaluate these schemes makes it challenging for developers to understand these tradeoffs and pick the best scheme to implement and meet user needs. In this study, we standardise metrics to evaluate the efficiency and security of RSE schemes. We apply these metrics to the existing schemes: Constant.RSE, Quadratic.RSE, Logarithmic.RSE and Augmented.RSE. Then, we recommend optimal schemes for use, enabling developers to make better, informed decisions.

1.1 Our Contributions

Our contributions are as follows:

- We **standardise metrics** to compare the space efficiency, time efficiency, and security of RSE schemes and apply these metrics for an **overall comparison** between the schemes studied. These metrics can **apply to other schemes in future work** for more points of comparison.

- We are the **first to devise a quantifiable metric** for the information leakage of RSE schemes, known as "volume leakage". This allows for **an easy comparison of scheme security**. Previous work achieved security comparisons through unwieldy leakage profiles which are qualitative and often incomparable. We further evaluate scheme security by running simulations and bounding each scheme's "volume leakage".

- We used **complexity theory** to model the security, space efficiency, and time efficiency of different schemes to identify trade-offs made between efficiency and security in schemes.

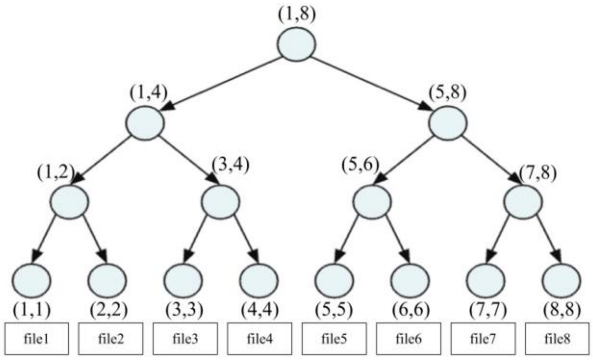
- We make a **final recommendation** for schemes that are optimal for implementation, enabling developers to make informed decisions.

2 Definitions

RSE background An RSE scheme RSE is built using SE and CHF. It has 4 algorithms: RSE.Setup, RSE.Token, RSE.Search and RSE.Dec. RSE.Setup takes in files and a key, and uses the key to encrypt the files into ciphertexts and to evaluate labels corresponding to each file into attributes to form an Encrypted Data Structure (EDS). RSE.Token takes a query range (*a*,*b*) and splits the range into labels within EDS, then evaluates these labels into attributes, which get returned as a single token. RSE.Search runs server-side, and takes the token from RSE.Token and returns ciphertexts from EDS that correspond to the attributes in the token. RSE.Dec decrypts ciphertexts from RSE.Search into files within query range (f_a , f_{a+1} , ⋯ , f_b). RSE.max is the maximum number which can be queried for in an RSE scheme.

Constant (CONST) The data structure of CONST.RSE is structured where each unique characteristic with a numerical value (e.g. height, salary, weight, etc.) within the dataset is a label. Files are assigned to the labels matching their characteristics. E.g, characteristic *a* becomes a label, and file f_a is its corresponding value. Characteristic *b* becomes a label and f_b is the corresponding value. Each label and its assigned file become 1 entry in the data structure.

Quadratic (QUAD) QUAD.RSE's data structure has every possible range query, for a given size of scheme, assigned to a label. Files with characteristics within a query range are assigned to labels within that query range. E.g., query range (*a*,*b*) becomes label (*a*,*b*), and files (f_a , f_{a+1} , ⋯ , f_b) correspond to this label. Each label and its assigned files become 1 entry in the data structure.



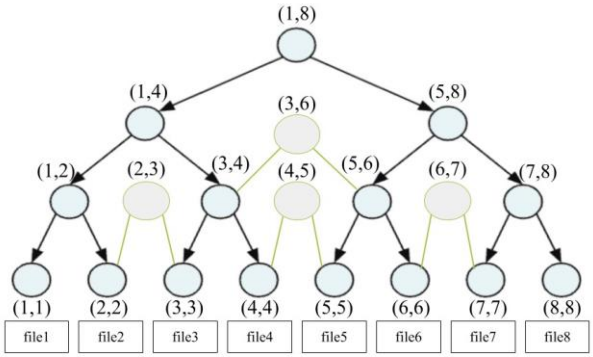
Labels	Files
(1,8)	file1, file2, … file8
(1,4)	file1, file2, file3, file4
(5,8)	file5, file6, file7, file8
(1,2)	file1, file2
(3,4)	file3, file4
(5,6)	file5, file6
(7,8)	file7, file8
(1,1)	file1
(2,2)	file2
⋮	⋮
(8,8)	file8

(a) Derivation of Labels in LOG.RSE (b) Data Structure of LOG.RSE

Fig.1: Diagrammatic Representation of Data Structure of LOG.RSE

Logarithmic (LOG) The labels of LOG.RSE are derived from the expansion of a binary tree, where the binary tree nodes represent query ranges that are labels in the data structure (expressed as 2-tuples). The root node is (1,RSE.max). Every node splits into 2 children nodes. The left child node covers the first half of the parent node's range, while the right child node covers the second half of that range. For example, node (1, RSE.max) has children nodes (1, $\frac{\text{RSE.max}}{2}$) and ($\frac{\text{RSE.max}}{2}$ + 1,

RSE.max). This division continues until the start and end integers of each child are the same, these nodes (e.g. (1,1), (2,2), etc.) are leaf nodes. Files with characteristics within each label's range correspond to that label.



(a) Derivation of Labels in AUG.RSE

Labels	Files
(1,8)	file1, file2, … file8
(1,4)	file1, file2, file3, file4
(3,6)	file3, file4, file5, file6
(5,8)	file5, file6, file7, file8
(1,2)	file1, file2
(2,3)	file2, file3
(3,4)	file3, file4
⋮	⋮
(1,1)	file1
(2,2)	file2
⋮	⋮
(8,8)	file8

(b) Data Structure of AUG.RSE

Fig.2: Diagrammatic Representation of Data Structure of AUG.RSE

Augmented (AUG) The labels of AUG.RSE are derived from the expansion of a binary tree. It has all the nodes of LOG.RSE, with the addition of "augmented" nodes. The augmented nodes connect from 2 adjacent children nodes not sharing a direct parent. For example, in LOG.RSE nodes (3,4) and (5,6) do not share a direct parent. In AUG.RSE, both nodes now have parent node (3,6). Augmented nodes do not have a parent node.

Exact Cover Algorithm A cover algorithm is present in RSE.Token of some schemes, like LOG.RSE and AUG.RSE. It takes the query range input of RSE.Token and returns a list of labels covering the query range. Given a query range (*a*,*b*), **Alg**ExactCover returns the fewest nodes that cover the query range exactly.

Overcover Algorithm **Alg**Overcover takes the query range (*a*,*b*) as the input and returns a single number assigned to the node that covers the entire query range (*a*,*b*). That means for the most part, **Alg**Overcover returns the assigned number of a node (*p*,*q*) where $p \leq a \leq b \leq q$, where ciphertexts outside the query range will also be returned by RSE.Search.

For the full schemes and cover algorithms, refer to appendix A.

3 Volumetric Analysis Framework and Security Comparison Results

3.1 Summary

Const.RSE is the least secure, QUAD.RSE is highly secure. LOG.RSE and AUG.RSE using **Alg**ExactCover are moderately secure, **Alg**OverCover significantly improves scheme security.

3.2 Context

Some information is leaked to adversaries for each entry accessed in the EDS, as they observe operations on encrypted data. For example, if query (1,4) was made in CONST.RSE, followed by (1,2), an adversary sees that some ciphertexts from both queries overlap. If the adversary knows both queries, they can deduce which ciphertexts belong to (3,4). Given more information,

adversaries better understand the structure and contents of a database, making it more prone to attacks. Hence, we compare the information leaked by each scheme. Another form of information leakage is the frequency of data access. For example, if the same query is made many times, it can be deduced that ciphertexts within that range are important, which the adversary then focuses on decrypting. Lastly, frequent access to certain ranges also imply that information within that range is more common, revealing data distribution. Adversaries can match that with auxiliary information and infer the contents of encrypted data without decryption. Evidently, many kinds of qualitative information leakage exist, making it difficult to compare the security of schemes. Hence, our solution consolidates different forms of leakage into a single measure, for direct comparison. We take the information leakage associated with each entry in the EDS as 1 volume and measure each scheme's volume leakage.

3.3 Methodology

We ran a simulation calculating the percentage of all volumes leaked when RSE.max = 2048 and 1000000 unique queries were made. We plotted the results (refer to Fig 5), taking the average percentage of the maximum number of volumes leaked in 200 runs, for each number of query made, to thin out the impact of any outliers or the order of queries. No encryption is needed in a simulation, so the number of volumes leaked is the number of entries returned from an unencrypted data structure. To count the volumes leaked, the simulation checks if the entry returned has not been returned already, and adds that entry to a counter of the volumes leaked.

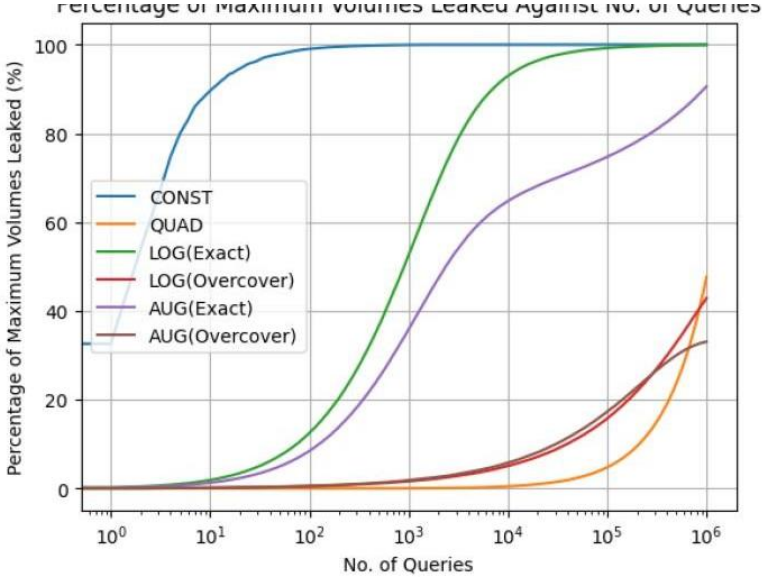


Fig.3: Graph of the Percentage of Maximum Volumes Leaked Against Number of Queries Made

3.4 Discussion

QUAD In QUAD.RSE, the increase in volume leakage is minimal from 0 to 10000 queries, as each query returns a single entry, leaking 1 volume at a time. Additionally, the total number of volumes is high (2098176), as the labels in QUAD.RSE are all possible ranges from 1 to RSE.max (2048). Hence, QUAD.RSE has minimal volume leakage and a high level of security.

CONST In CONST.RSE, volume leakage is around 35% after the first query, because for a query range of size R, the server returns R number of entries. The gradient is initially sharp, before decreasing from 5 queries onwards. This is because fewer unique entries are returned, as the number of queries increases. Eventually, all volumes are leaked after around 100 queries, as all entries in the EDS have been accessed

LOG ExactCover Volume leakage increases sharply as the number of queries increases to 5000. From 5000 queries onwards, the gradient decreases. Though the number of queries grow exponentially, subsequent queries tend to return previous entries, as nodes at deeper parts of the binary tree are less likely to be returned, decreasing volume leakage significantly in later queries.

AUG ExactCover The gradient is sharp until 5000 queries, then it decreases from 5000 to 50000 queries. Lastly, the gradient increases after 50000 queries. The fluctuation is caused by 2 reasons. Firstly, the number of queries increases exponentially, hence there is a larger increase in volume leakage after 50000 queries. Secondly, for AUG.RSE, there are multiple ways queries can be covered with the same number of nodes. For example, range (3,8) is covered by {(3,4), (5,8)} and {(3,6), (7,8)}. AlgExactCover prioritises returning non-augmented nodes. Hence, entries of augmented nodes are less likely to be returned. Therefore, the initial plateau is because most nodes have already been accessed, while augmented nodes are not accessed. The gradient increase comes as augmented nodes get accessed by more unique queries.

LOG v.s. AUG ExactCover When comparing LOG.RSE and AUG.RSE using **Alg**ExactCover, the percentage of volumes leaked is higher for LOG.RSE than AUG.RSE. This is because AUG.RSE has more labels covering more unique ranges, requiring fewer labels to cover the queried range. For example, when query (4,5) is made, LOG.RSE using **Alg**ExactCover returns 2 entries with labels (4,4) and (5,5), leaking 2 volumes while AUG.RSE using **Alg**ExactCover returns only the entry with label (4,5), leaking 1 fewer volume.

ExactCover v.s. Overcover ExactCover schemes leak more volumes than **Alg**Overcover schemes. Per query, **Alg**Overcover returns 1 label, while **Alg**ExactCover returns multiple labels and is more likely to result in new unique entries returned, increasing volume leakage.

4 Space Efficiency

4.1 Storage

Context Space efficiency comprises storage and bandwidth. Storage efficiency is the space occupied by each scheme's data structure. Storage is based on the number of files stored in each scheme, which is represented in formulas with RSE.max (*n*) as the subject. From these formulas, storage complexity is derived, representing how the size of the data structure scales as *n* increases. For the sake of comparison, each characteristic only has 1 corresponding file.

4.2 Storage Efficiency Analysis

Scheme	CONST	QUAD	LOG	AUG
Total number of labels	n	$\frac{n(n+1)}{2}$	2n - 1	3n - ($\log_2$n + 2)
Total number of files	n	$\frac{n^3 + 3n^2 + 2n}{6}$	n ($\log_2$n + 1)	2n ($\log_2$n) - n + 2
Storage Complexity	θ(n)	θ(n^3)	θ(n $\log_2$n)	θ(n $\log_2$n)

Fig.4: Summary of Storage Analysis

Summary CONST.RSE is most storage efficient ($\theta(n)$), LOG.RSE and AUG.RSE to have moderate storage efficiency ($\theta(n\log_2 n)$) and QUAD.RSE to be least storage efficient ($\theta(n^3)$).

QUAD The number of files is the product of every possible range and the number of possible queries for that range. For query range n, there is 1 possible query (1,n). Hence the number of files for range n is n. For query range $n-1$, there are 2 possible queries, (1,$n-1$) and (2,n). Hence the number of files for range $n-1$ is $2(n-1)$. Continuing this, the total files in QUAD.RSE is:

$$
\begin{aligned}
& n + 2(n-1) + 3(n-2) + \cdots + n(1) \\
&= (0+1)(n) + (1+1)(n-1) + (2+1)(n-2) + \cdots + (n-1+1)(n-(n-1) \\
&= \sum_{k=0}^{n-1} (k+1)(n-k) \\
&= \frac{n^3 + 3n^2 + 2n}{6}
\end{aligned}
$$

LOG Each depth in the binary tree has n files. Number of depths in the binary tree = $\log_2 n + 1$. Hence, the total number of files is:

$$\text{No. of files in each depth} \times \text{No. of depths} = n(\log_2 n + 1)$$

AUG For every depth in the AUG.RSE tree except for the last depth (the leaves), depths are assigned a depth number i, where $i \in \{0,1,\cdots,\log_2 n - 1\}$. For every depth, the total number of nodes is the number of nodes in a LOG tree, in addition to the augmented nodes, which is:

$$2^i + (2^i - 1) = 2^{i+1} - 1$$

For every depth, the number of files contained per node is $(\frac{n}{2^i})$. Hence, for every depth, the number of files across the nodes at that level is the product of both expressions:

$$(2^{i+1}-1)(\frac{n}{2^i})$$

Hence, the total number of files excluding the last depth is the summation of all depths:

$$\sum_{i=0}^{log_2 n-1}(2^{i+1}-1)(\frac{n}{2^i})$$
$$= n(2log_2 n - 2 + \frac{2}{n})$$

For the last depth (the leaves), the number of files would be the number of leaves, *n*, since each leaf contains 1 file. Hence, the total number of files in the entire augmented tree would be:

$$n(2log_2 n - 2 + \frac{2}{n}) + n$$
$$= 2nlog_2 n - n + 2$$

For the full proof for the storage formulas of AUG.RSE, refer to appendix B.

4.3 Bandwidth

Context Bandwidth comprises upload and download. Upload bandwidth is measured with the number of attributes within the token of RSE.Token. Download bandwidth is measured with the number of ciphertexts from RES.Search after a query. Smaller bandwidths improve performance.

4.4 Bandwidth Efficiency Analysis

Summary The u**pload bandwidth** of CONST.RSE is the largest possible size and bounded by $\theta(R)$, followed by LOG.RSE and AUG.RSE using **Alg**ExactCover which are smaller and bounded by $O(log_2 R)$. QUAD.RSE, and LOG.RSE and AUG.RSE using **Alg**Overcover have the smallest upload bandwidth, bounded by $\theta(1)$.

The **download bandwidth** of CONST.RSE, QUAD.RSE, and LOG.RSE and AUG.RSE using **Alg**ExactCover is the smallest possible download size (bounded by $\theta(R)$). Download bandwidth of LOG.RSE and AUG.RSE using **Alg**Overcover is bounded by the size of the query ($\theta(R)$).

CONST and QUAD Upload CONST.RSE performs the worst as it creates a separate token for every characteristic within the query range, the upload size is always *R* and it is bounded by $\theta(R)$. On the other hand, QUAD.RSE performs the best, as any query is hashed into a single attribute in a token, hence QUAD.RSE's upload bandwidth is bounded by $\theta(1)$.

LOG and AUG Upload (ExactCover) For LOG.RSE and AUG.RSE using **Alg**ExactCover, the most attributes within the 1 token are ($2log_2 R - 2$) and ($2log_2 R - 3$) respectively (upload bandwidth bounded by $O(log_2 R)$). Given RSE.max = 2048, average sizes of the token for the schemes as a percentage of the size of CONST.RSE (largest possible query) are 3.31% and 3.15% respectively. Hence, average upload sizes for both schemes are a tiny percentage of the largest possible upload, as both schemes require only a few labels to cover a large range. Hence, LOG.RSE and AUG.RSE perform much closer to QUAD.RSE than CONST.RSE.

LOG and AUG Upload (Overcover) For LOG.RSE and AUG.RSE using **Alg**Overcover, the upload size is always 1 and the bound is $\theta(1)$ as AlgOvercover returns only 1 attribute.

CONST, QUAD and ExactCover Download CONST.RSE, QUAD.RSE, and LOG.RSE and AUG.RSE using AlgExactCover always return only ciphertexts within the query range. Hence, the download size of these schemes is R and is bounded by the size of the query ($\theta(R)$).

Overcover Download However, for LOG.RSE and AUG.RSE using **Alg**Overcover, ciphertexts outside of the query range can be returned. Hence, both schemes are bounded by their largest possible download size. Excess is the number of ciphertexts returned that are outside the query range. The largest possible download for LOG.RSE using **Alg**Overcover has an excess of $(n - 2)$ ciphertexts. For AUG.RSE using **Alg**Overcover, it is an excess of ($\frac{3n}{4} - 2$) ciphertexts (refer to appendix C for proof). Hence, the download bandwidth of both schemes is bounded by O(n).

With RSE.max as 2048, we ran simulations to find the average excess for LOG.RSE and AUG.RSE using **Alg**Overcover as a percentage of the size of the query range (smallest possible download size). For LOG.RSE and AUG.RSE, the average excess is 173% and 83.6% respectively. This means 173% extra files are downloaded on average per query for LOG.RSE using **Alg**Overcover. This shows that Augmented.RSE using **Alg**Overcover performs better and has a smaller average download size, because Augmented.RSE has augmented nodes covering additional ranges, hence **Alg**Overcover can more tightly cover given ranges, returning fewer excess ciphertexts. Scaling up to bigger values of RSE.max, the percentage excess for both also increases. Hence, the number of excess ciphertexts would increase, increasing the download size.

5 Time Efficiency

Context Time efficiency of an RSE scheme is evaluated by comparing the runtimes of setup and query, which is based on the time complexity of algorithms in each process. The setup process is RSE.Setup and the query process is composed of RSE.Token, RSE.Search and RSE.Dec.

5.1 Setup Runtime Analysis

Summary CONST.RSE has the shortest setup runtime ($\theta(n)$), LOG.RSE and AUG.RSE have moderate setup runtimes ($\theta(n\log_2 n)$) and QUAD.RSE has the longest setup runtime ($\theta(n^3)$).

CONST For all files in the cloud ($\theta(n)$ iterations), each file is encrypted ($\theta(1)$ step) and its corresponding label is evaluated ($\theta(1)$ step). Hence, time complexity of CONST.RSE's setup is:

$$\theta(n) \text{ steps}$$

QUAD Iterating i starting from 1 up to to *RSE.max* ($\theta(n)$ iterations), for each value of i, iterate over j starting from i up to to *RSE.max* ($\leq \theta(n)$ iterations). For each pair of i and j, combine all the files in the range i to j into 1 file ($\leq \theta(n)$ steps), encrypt the combined file ($\theta(1)$ step) and evaluate the range i to j ($\theta(1)$ step). Hence, time complexity of QUAD.RSE's setup is:

$$\theta(n) \times \theta(n) \times \theta(n) = \theta(n^3) \text{ steps}$$

LOG/AUG For all depths in the binary tree($\theta(log_2 n)$ iterations), and for each node on each depth ($\theta(n)$ iterations), generate its left child node (label) ($\theta(1)$ step) and its right child node (label) ($\theta(1)$ step). For AUG.RSE, the formation of the EDS involves all nodes having "3 children nodes" now, except for the depth above the leaves. Hence, given the depth of the binary tree ($\theta(log_2 n)$ iterations) and for each node at each depth ($\theta(n)$ iterations), the 2 children nodes and the augmented node in between both children nodes is generated ($\theta(1)$ steps) Hence, setup time complexity for LOG.RSE and AUG.RSE is:

$$\theta(log_2 n) \times \theta(n) = \theta(n log_2 n) \text{ steps}$$

5.2 Query Runtime Analysis

The query process of an RSE scheme consists of RSE.Token, RSE.Search and RSE.Dec. To obtain the query time complexity, each algorithm was analysed for each RSE scheme to find the algorithm of the highest time complexity. For the RSE schemes compared, RSE.Token and RSE.`Search all have time complexities` $\leq O(R)$. Additionally, time complexity of RSE.`Dec is determined by the number of ciphertexts decrypted, which is` $\geq O(R)$. Hence, RSE.Dec is the highest complexity algorithm in the query process, thereby determining the query runtime.

CONST.RSE, QUAD.RSE, schemes using **Alg**ExactCover return ciphertexts within the query range (RSE.Dec runtime bounded by $\theta(R)$). However schemes using **Alg**Overcover have **longer query runtimes**, as these schemes can return ciphertexts outside of the query range. As such, time complexities for the Overcover schemes ($O(n)$) are bounded by the maximum number of ciphertexts returned, which is ($n - 2$) `and` ($\frac{3n}{4} - 2$) respectively (refer to 4.4).

Scheme	Space Efficiency			Time Efficiency		Security
	Storage	Bandwidth (Upload)	Bandwidth (Download)	Runtime (Setup)	Runtime (Query)	Volume Leakage
CONST	$\theta(n)$	$\theta(R)$	$\theta(R)$	$\theta(n)$	$\theta(R)$	$O(R)$
QUAD	$\theta(n^3)$	$\theta(1)$	$\theta(R)$	$\theta(n^3)$	$\theta(R)$	$O(1)$
LOG (Exact)	$\theta(n \log_2 n)$	$O(\log_2 R)$	$\theta(R)$	$\theta(n \log_2 n)$	$\theta(R)$	$O(\log_2 R)$
LOG (Overcover)	$\theta(n \log_2 n)$	$\theta(1)$	$O(n)$	$\theta(n \log_2 n)$	$O(n)$	$O(1)$
AUG (Exact)	$\theta(n \log_2 n)$	$O(\log_2 R)$	$\theta(R)$	$\theta(n \log_2 n)$	$\theta(R)$	$O(\log_2 R)$
AUG (Overcover)	$\theta(n \log_2 n)$	$\theta(1)$	$O(n)$	$\theta(n \log_2 n)$	$O(n)$	$O(1)$

Legend
Green: Excellent
Light Green: Good
Yellow: Fair
Red: Terrible

R: Size of query range
n: Number of unique characteristics
n ≥ R (n is the maximum user can query for)

O: Upper bound (scheme's worst case)
θ: Tight bound (scheme's average case)

Fig.5: Overall Security-Efficiency Tradeoffs

6 Final Recommendations and Future Work

Schemes Not Recommended for Use From our results, CONST.RSE takes up the least storage space, and the least time to set up and run. However, CONST.RSE has the largest upload size and

is the least secure scheme. This renders CONST.RSE unable to secure sensitive data (e.g. health records). On the other hand, QUAD.RSE is a very secure scheme and has the smallest upload and download sizes. However, QUAD.RSE is the least storage efficient, resulting in the longest time taken for setup. This means QUAD.RSE is resource-intensive, as it increases server operating costs in implementation. Hence, we do not recommend both CONST.RSE and QUAD.RSE, given the significant trade-offs in security and storage respectively.

Recommended Schemes LOG.RSE and AUG.RSE, are balanced across all metrics and are suited for most uses. Due to the added augmented nodes in AUG.RSE, the scheme has smaller upload sizes and less information leakage than LOG.RSE (when using both cover algorithms). However, this comes at the expense of more storage space needed and a longer setup time for AUG.RSE, increasing server costs. Hence, we recommend AUG.RSE for implementations prioritising security and can accommodate the higher operating costs. We recommend LOG.RSE for developers prioritising cost efficiency.

Lastly, between **Alg**ExactCover and **Alg**Overcover, we recommend **Alg**ExactCover for frequent querying as it reduces download size and time needed for decryption. Conversely, we recommend using **Alg**Overcover for developers prioritising security, as it massively reduces information leakage. Hence, schemes using **Alg**Overcover should be used to store highly sensitive data (e.g. financial records)

Future Work The devised metrics can be applied to more RSE schemes in existing literature, to enable more scheme recommendations to make better choices.

7 Acknowledgements

We would like to thank our mentor Ruth Ng Ii-Yung for teaching us about her guidance and advice throughout our research project. We would also like to thank our mentor Phang Yan Feng Benito for his support and assisting us in programming for the project.

References

1. Ioannis Demertzis, Stavros Papadopoulos, Odysseas Papapetrou, Antonios Deligiannakis, and Minos N. Garofalakis. Practical private range search revisited. In Fatma Ozcan, Georgia Koutrika, and Sam Madden, editors,¨ *Proceedings of the 2016 International Conference on Management of Data, SIGMOD Conference 2016, San Francisco, CA, USA, June 26 - July 01, 2016*, pages 185–198. ACM, 2016.
2. Sky Faber, Stanislaw Jarecki, Hugo Krawczyk, Quan Nguyen, Marcel Rosu, and Michael Steiner. Rich queries on encrypted data: Beyond exact matches. In *European symposium on research in computer security*, pages 123–145. Springer, 2015.

Appendix

A RSE Algorithms

CONST.RSE scheme is built using **SE** scheme (SE) and **CHF** scheme CHF. It defines:

AlgCONST.Setup(K,(f_1,f_2,$\cdots$,$f_{\text{CONST.max}}$))	**Alg**CONST.Token(K,a,b)	**Alg**CONST.Search(EDS,tk)	**Alg**CONST.Dec(K,C)
$K_{SE}\|\|K_{CHF} \leftarrow K$ Initialise EDS array For $i \in (1,2,\cdots,\text{CONST.max})$: ct $\leftarrow_s$ SE.Enc(K_{SE},file$_i$) attr $\leftarrow_s$ CHF.Ev(K_{CHF},i) EDS[attr] $\leftarrow$ ct Return EDS	$K_{SE}\|\|K_{CHF} \leftarrow K$ Initialise tk set For $j \in (a,a+1,\cdots,b)$: attr$_j$ $\leftarrow$ CHF.Ev(K_{CHF},j) tk $\leftarrow$ tk $\cup$ attr$_j$ Return tk	C $\leftarrow$ {} For $v \in$ tk : C $\leftarrow$ C $\cup$ EDS[v] Return C	$K_{SE}\|\|K_{CHF} \leftarrow K$ Files $\leftarrow$ {} For $c \in$ C Files $\leftarrow$ Files $\cup$ {SE.Dec(K_{SE},c)} Return Files

An **Quadratic**.**RSE** (QUAD.RSE) scheme is built using **SE** scheme (SE) and **CHF** scheme CHF. It defines:

AlgQUAD.Setup(K,(f_1,f_2,$\cdots$,$f_{\text{QUAD.max}}$))	**Alg**QUAD.Token(K,a,b)	**Alg**QUAD.Search(EDS,tk)	**Alg**QUAD.Dec(K,C)
$K_{SE}\|\|K_{CHF} \leftarrow K$ Initialise EDS array For $i \in (1,2,\cdots,\text{QUAD.max})$: For $j \in (i,i+1,\cdots,\text{QUAD.max})$: For $h \in (i,i+1,\cdots,j)$: file$_{i-j}$ $\leftarrow$ file$_{i-j}$ $\cup$ file$_h$ ct $\leftarrow_s$ SE.Enc(K_{SE},file$_{i-j}$) attr $\leftarrow_s$ CHF.Ev(K_{CHF},(i,j)) EDS[attr] $\leftarrow$ ct Return EDS	$K_{SE}\|\|K_{CHF} \leftarrow K$ q $\leftarrow$ (a,b) tk $\leftarrow$ CHF.Ev(K_{CHF},q) Return tk	C $\leftarrow$ {} C $\leftarrow$ C $\cup$ EDS[tk] Return C	$K_{SE}\|\|K_{CHF} \leftarrow K$ Files $\leftarrow$ {} Files $\leftarrow$ Files $\cup$ {SE.Dec(K_{SE},c)} $f_a\|\|f_{a+1}\|\|\cdots\|\|f_b \leftarrow$ Files Return {f_a,f_{a+1},$\cdots$,f_b}

LOG.RSE scheme is built using **SE** scheme (SE) and **CHF** scheme CHF. It defines:

AlgLOG.Setup(K,(f_1,f_2,$\cdots$,$f_{\text{LOG.max}}$))	**Alg**LOG.Token(K,a,b)	**Alg**LOG.Search(EDS,tk)	**Alg**LOG.Dec(K,C)

$K_{SE}\|\|K_{CHF} \leftarrow K$ Initialise EDS array start = 1 end = LOG.max $= \left\lfloor \frac{(\text{start} + \text{end})}{2} \right\rfloor$ mid Until start = end Find children nodes of (start,mid) Find children nodes of (mid + 1,end) EDS ← EDS ∈ children nodes Return EDS	$K_{SE}\|\|K_{CHF} \leftarrow K$ query list ← Cover(*a*,*b*) `For q ∈ query list :` tk ← CHF.Ev(K_{CHF},q) Return tk	C ← {} C ← C ∪ EDS[tk] Return C	$K_{SE}\|\|K_{CHF} \leftarrow K$ Files ← {} Files ← Files ∪ {SE.Dec(K_{SE},*c*)} $f_a\|\|f_{a+1}\|\|\cdots\|\|f_b \leftarrow$ Files Return $\{f_a, f_{a+1}, \cdots, f_b\}$

Alg ExactCover (a,b)

if $a = b$
  Return a
if $b - a = 1$ and $a \bmod 2 = 1$
  Return $\frac{\frac{a}{2} + \frac{b}{2}}{2}$

cover list = {}
if $a \bmod 2 = 1$:
  cover list ← cover list ∪ a
cover list ← cover list ∪ **Alg** Cover$\left(\frac{a+1}{2}, \frac{b-1}{2}\right)$
if $b \bmod 2 = 0$:
  cover list ← cover list ∪ b
Return cover list

Alg Overcover(a,b)

if $a = b$
  Return a
if $b - a = 1$ and $a \bmod 2 = 1$
  Return $\frac{\frac{a}{2} + \frac{b}{2}}{2}$

cover list = {}
cover list ← cover list ∪ **Alg** Cover$\left(\frac{a}{2}, \frac{b}{2}\right)$
Return cover list

AUG.RSE scheme is built using **SE** scheme (SE) and **CHF** scheme CHF. It defines:

AlgAUG.Setup(K,(f_1,f_2,$\cdots$,$f_{AUG.max}$))

$K_{SE}||K_{CHF} \leftarrow K$

Initialise EDS array

start = 1 end = LOG.max

$= \left\lfloor \frac{(start + end)}{2} \right\rfloor$ mid Until start = end

Aug node $= \left(\left\lfloor \frac{start + mid}{2} \right\rfloor + 1, \left\lfloor \frac{mid + 1 + end}{2} \right\rfloor \right)$

Find children nodes of (start,mid)

Find children nodes of (mid + 1,end)

EDS ← EDS ∈ children nodes

EDS ← EDS ∈ Aug node

Return EDS

AlgLOG.Token(K,a,b)

$K_{SE}||K_{CHF} \leftarrow K$

query list ← Cover(a,b)

`For q ∈ query list :`

tk ← CHF.Ev(K_{CHF},q)

Return tk

AlgLOG.Search(EDS,tk)

C ← {}

C ← C ∪ EDS[tk]

Return C

AlgLOG.Dec(K,C)

$K_{SE}||K_{CHF} \leftarrow K$

Files ← {}

Files ← Files ∪ {SE.Dec(K_{SE},c)}

$f_a||f_{a+1}||\cdots||f_b \leftarrow$ Files

Return {f_a,f_{a+1},$\cdots$,f_b}

B Proof for AUG.RSE Storage Formula

For every depth in the Augmented tree except for the last depth (the leaves), depths are assigned a depth number i, where $i \in \{0,1,\cdots,log_2 n - 1\}$. For every depth, the total number of nodes in the depth (binary + augmented) would be:

$$2^i + (2^i - 1)$$

$$= 2_{i+1} - 1$$

Which For every depth, the number of files contained per node is $\frac{n}{2^i}$. Hence, for every depth, the number of files across the nodes at that level would be the product of both expressions:

$$(2^{i+1} - 1)(\frac{n}{2^i})$$

Hence, the total number of files in the augmented tree excluding the last depth would be the summation of all depths:

$$\sum_{i=0}^{log_2 n - 1} (2^{i+1} - 1)(\frac{n}{2^i})$$

Which can be simplified:

$$\sum_{i=0}^{log_2 n-1} (2^{i+1}-1)(\frac{n}{2^i})$$

$$= \sum_{i=0}^{log_2 n-1} (2n - \frac{n}{2^i})$$

$$= n\left[\sum_{i=0}^{log_2 n-1} (2 - \frac{1}{2^i})\right]$$

$$= n\left[\sum_{i=0}^{log_2 n-1} 2 - \sum_{i=0}^{log_2 n-1} \frac{1}{2^i}\right]$$

$$= n(2log_2 n - (1 + (1 - \frac{1}{2^i}))$$

$$= n(2log_2 n - 2 + \frac{2}{n})$$

For the last depth (the leaves), the number of files across the nodes would be the number of leaves since each leaf contains 1 file. Hence, since there are *n* leaves, the number of files = *n*. Hence, the total number of files in the entire augmented tree would be:

$$n(2log_2 n - 2 + \frac{2}{n}) + n$$

$$= 2nlog_2 n - 2n + 2 + n$$

$$= 2nlog_2 n - n + 2$$

C Proof for Worst case possible (Upload)

For LOG.RSE, to have the worst upload possible, the query made would be (2,*n* − 1). This would make **Alg**ExactCover take 2 nodes at each depth except the depths with the root node and its direct children. Hence the number of nodes taken to fulfill the query would be 2(Number of layers) = 2($log_2 n$ − 1). For AUG.RSE, the same query would be made.**Alg**ExactCover would follow the same process as in LOG.RSE, taking 2 nodes at each depth except for the depths with the root node and its direct children. There, instead of taking the 2 nodes adjacent to each other, it will take the first augmented node to fulfill the query range. Hence, AUG.RSE would return 1 less node than LOG.RSE, making its formula (2$log_2 R$ − 3).