NEW OPTIMAL ALGORITHMS FOR COMPUTING BINARY TREE OVERCOVERS IN RANGE SEARCHABLE ENCRYPTION

Richard Ong Jun Quan[1], Guan Keer[2] , Ruth Ng Ii-Yung[3]

[1]NUS High School of Mathematics and Science, 20 Clementi Avenue 1, Singapore 129957

[2]Nanyang Girls' High School, 2 Linden Drive, Singapore 288683

[3]DSO National Laboratories, 20 Science Park Dr, Singapore 118230

Abstract Range Searchable Encryption is a data storage method that allows for the outsourcing of search of large, encrypted data sets to an untrusted server. Many state of the art schemes make use of binary trees and break query processing down into two parts: generating a "cover" for a given range and accessing the tree data structure to return relevant records.

Our work provides novel contributions on both parts. In terms of cover-generating algorithms, we focus on "overcover" algorithms as there exist no generic optimal ones in the literature. We developed two optimal algorithms for different variants of overcover generation and analyzed them via mathematical proofs and benchmarking. In terms of tree datastructures access, we designed and benchmarked an improved approach using recursive hashing which reduces server side storage.

1. Introduction

Range Searchable Encryption (RSE) is a data storage method that has been described in previous studies[1,2]. and addresses a setting where a client *C* outsources search for an encrypted document set within a given range in an encrypted database to an untrusted server *S*. Ideally, *C* can search for a collection of documents without revealing information about what is being searched and returned beyond the size of the query and returned range to adversaries (untrusted S, attackers) and S can authorize range queries based on the size of the queried range without learning the actual endpoints of the range.

Our research is impactful since the outsourcing of data storage has become popular in modern applications due to advantages in cost, scalability and accessibility. However, there is demand for end-to-end encryption from users who wish their data to be hidden from the cloud service provider, eavesdroppers on the network or hackers who breach the cloud. This is the motivation behind cryptographic schemes such as RSE.

Existing state of the art RSE schemes make use of binary trees to represent the encrypted database and search through it for requested ranges [1,2]. They are all able to return descendant nodes containing relevant encrypted documents for a given node higher up in the tree. A set of such nodes where the union of their descendants give a certain continuous range is known as a cover. Cover-generation algorithms which compute a cover for a specified range must be used in conjunction with a RSE scheme for arbitrary ranges to be queried. Currently, there exists four variants of cover-generation algorithms in the literature. Two variants, classed as overcovers which are covers that return false positives, offer the additional advantages in hiding information from adversaries but are relatively unexplored in literature

The use of RSE involves 2 parts: a tree search algorithm and a cover-generation algorithm and we will present our novel contributions to both parts.

1.1 Our contributions

- We have designed and implemented **two novel optimal generic overcover-generations algorithms** in both the **"non-universal" and "universal"**

settings. Previous studies have only presented heuristic approaches to these, but our dynamic programming solutions are optimal. We demonstrate this optimality both mathematically (via proofs) and experimentally (via implementation and benchmarking). Due to this, our algorithms are substantially faster than a brute force approach and reduce overheads compared to the heuristics of prior work in large ranges.

- We present **two new mathematical proofs** relevant to the problem of overcover generation that prove the optimality of the above algorithms and bind the run time of one algorithm.
- In the "universal" setting above, we provide **new heuristics** for the special cases of **overcovers of size 4 and 5**. These heuristics run much faster than our optimal algorithm and may be of interest in high-efficiency RSE implementations. We present experiments which show that these perform as well as the optimal algorithm on ranges of size 100 or less . Finally, we described and implemented an **novel RSE scheme R-RSE** which improves upon schemes in the literature through the use of ratcheted hashing, resulting in reduced server-side storage. This RSE achieves memory complexity that scales similarly to a cleartext implementation (no encryption), which cannot be said for RSE schemes in the literature.

2. Preliminaires

2.1 Binary Trees: Our algorithms are defined in terms of infinite binary trees. Figure 1 shows a subsection of this tree. **Nodes** are referred to in the notation **(h, p)** where h is the **height** of the nodes which is defined as the layer the node is on, counted from the bottom of the tree and p is the **position** of the node in its layer, counted from the left. A **leaf** is defined as a node with no nodes connected below it (ie. At the bottom most layer of the tree) and has a height of 1. **Height profile** is a set of heights referring to the height of each node in a set of nodes. We refer to node x as a **descendant** of node y and y as a **predecessor** of x if y is on the sho
rtest path between x and the root 1. We refer to node x as a **child** of node y and y as a **parent** of x if y is a predecessor of x and is separated from x by one vertex. In a binary tree, a set of nodes is known as a **cover** of range (a,b) where (1,a) and (1,b) are leaves if the union of their leaf descendants contain all leaves between a and b (inclusive).

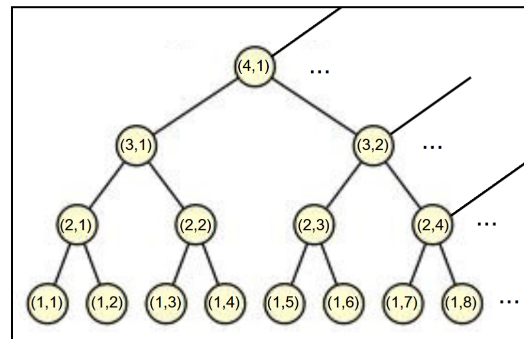

Figure 1: Infinite binary tree

3. Range Searchable Encryption

Simply put, RSE allows for the safe outsourcing of large encrypted databases to untrusted cloud providers while allowing for the data to be searched.

3.1 Background:

Current state of the art Range Searchable Encryption (RSE) schemes [1,2] are able to store encrypted data with a binary tree and return encrypted data within a given range with a tree search. They consist of a setup phase and query phase.

During the setup phase, the client *C* provides the encrypted database in a pre-defined format to the server *S* such that *S* can search through it. This includes a lookup table for individual

documents encrypted with a Symmetric Encryption (SE) Scheme based on their hashes generated by Cryptographic Hash Functions (CHF) and a lookup table for search functionality needed for certain schemes.

During the query phase, C uses a variant of a cover-generation algorithm to find a suitable cover for the range it wants to query and sends nodes in it as tokens which are hashes generated by the same (CHF) as above to hide information from S about the actual nodes in cover. A cryptographic key K is used for both encryption and tokenization operations and is kept private by C. S receives the tokens which represent encrypted nodes in the tree and must find the encrypted documents within the given range which are represented as descendants of the encrypted nodes in cover without knowing the actual range of search and returns them to C. C receives the set of encrypted documents and decrypts it with K, if an overcover is used, causing extra unwanted documents to be returned, C can filter them out as it has knowledge of the range.

3.2 Novel Improved RSE scheme

Our RSE scheme, **Ratcheting RSE (R-RSE)** is an improvement of a scheme proposed by [1] which we will refer to as **F-RSE** that has a second lookup table for nodes to reference each other. Specifically, F-RSE has the second lookup table for search with values being the encrypted form of each node in the binary tree and the entries being the encrypted descendant nodes of the node in the value which refer to encrypted documents. Given an encrypted node by C, S returns entries in the lookup table based on it. We will highlight the primary differences and improvements on F-RSE in our schemes.

Our improved scheme leverages the uniqueness of cryptographic hashes to reduce storage size. Nodes are assigned a cryptographic hash based on their position in the tree which does not have to be stored and can be computed with knowledge of the secret cryptographic key and node number. The hashes of bottommost nodes which represent documents can be derived from cryptographic hashes of its predecessors without knowledge of cryptographic key by S. Hence, the scheme is able to function without storing the search lookup table, significantly reducing storage size. For a detailed description of our scheme, it can be found in section 1 of the appendix.

3.3 Benchmarking

We implemented our improved scheme as well as the original scheme and benchmarked their storage sizes against plaintext as shown in figure 2. We used synthetic datasets of a given depth with all documents of length 100 bytes and measured the storage size as the byte length of minimum data needed to be stored on the server for the scheme to function. We used AES-CTR as the SE for encryption of the dataset and HMAC-256 as the CHF for token generation. The storage size of the improved R-RSE scheme is smaller than that of the original F-RSE for all values as expected. From depth 6 to 14, there is a constant 19.5 % improvement in storage size by our scheme for such parameters. The storage size of our improved scheme scales the same as that of plaintext data storage which is minimal/baseline. Storage size improvements are most obvious for

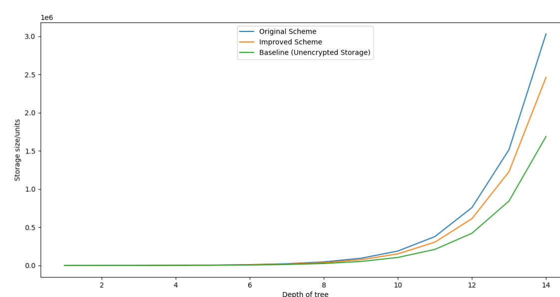

Figure 2: Graph of storage sizes against depth of the tree for different schemes

storage of shorter documents and there stands to be more the most benefits.

4. Binary Tree Covers

All current RSE schemes[1,2], including our new scheme, require cover-algorithms to enable arbitrary range search. There are four variations of covers focused for different purposes which we will define. Two are well-studied in the literature and optimal algorithms exist, while the other two we will be focusing on which offer more advantages for information hiding have been overlooked in the literature.

4.1 Background:

A set of nodes is known as an **exact cover** of range (a,b) if the union of their leaf descendants contains exclusively all the leaf nodes between a and b (inclusive). (e.g.) An arbitrary number of nodes are in the set for an exact cover as no false positives are allowed. A set of nodes is known as an **overcover** of range (a,b) if the union of their descendants contains all leaves between a and b (inclusive) and other false positives. An overcover can consist of a fixed number of nodes. (e.g.) Overcovers are the variant of covers that are relatively unexplored in literature but can improve security and privacy by allowing queries for ranges of the same size to appear indistinguishable to the server.

(a,b) denotes a range (i.e., a sequence of contiguous values) of leaf nodes to be queried which includes all nodes between (1,a) and (1,b) (inclusive), a<=b. **c** denotes the number of nodes in a cover. **e** is defined as the number of nodes in a cover's descendants not in the range it covers (i.e. false positives) which we will also refer to as error or overhead. **r** is the range size of a given range (a,b), r=b-a+1.

4.2 General Cover Algorithms

Input: Range **(a,b)**, Output: Cover $\{(h_1,p_1),\ldots,(h_c,p_c)\}$ of (a,b) where b - a +1 >= c

Now we will introduce general cover algorithms which are a group of algorithms that take an input of a range of nodes (a,b) (inclusive) and an integer c and return a c-cover of that range, a cover with a set of c number of nodes in it. The idea is that the union of the leaf descendants of the cover should include the leaves from (1,a) to (1,b). We say that a cover is exact if we do not allow leaves outside this range, and it is an overcover otherwise. The output from a cover algorithm enables a RSE scheme to return files in the range. If overcovers are returned, *C* will filter out false positives after decryption as it has knowledge of the range.

Cover algorithms may be heuristic or optimal. An optimal algorithm minimizes some parameter, which differs depending on the cover variant. Below, we discuss four variants and the relevant parameters for each. An algorithm that cannot be proven to be optimal is called a "heuristic".

4.2.1 Non-Universal and Universal Cover Algorithms

A **Non-Universal Cover** covers a range (a,b) minimally (with the least false positives) with a c-cover. The **Non-Universal Cover Algorithm** returns a Non-Universal Cover. A **Universal Cover** is a cover that has the same ordered height profile as any other universal cover for all ranges (a,b) with the same range size r = b-a+1.

As shown in diagram 1, there are 4 variants of covers obtained by considering combinations of the characteristics of "Exactness" and Universality.

Optimal Exact Non-Universal Cover Algorithms have been presented by Demertzis et al [1] and optimal Exact Universal Cover Algorithms have been presented by Faber et al [2].

For **Non-universal Overcover Algorithms (NUOA)**, there exists only a Heuristic Algorithm for c=1 with an augmented tree presented by [1]. In this work, we contribute novel optimal algorithms to find generic non-universal c-Overcovers for arbitrary c as well as novel optimal algorithms for finding non-universal 1,2-overcovers faster than the generic algorithm.

<table>
<tr><th>Cover Algorithms</th><th>Non-Universal Overcover Algorithms (NUOA)</th><th>Universal Overcover Algorithm (UOA)</th></tr>
<tr><td>Exact cover:
Minimize c, e = 0</td><td colspan="2">[1] and [2]: Optimal Algorithms</td></tr>
<tr><td rowspan="3">Overcover:
Fixed c, Minimize e</td><td>[1]: Heuristic Algorithm for c=1 (w/ modified tree)</td><td>[2]: Heuristic Algorithm for c=3</td></tr>
<tr><td colspan="2">[Ours]: Optimal Algorithms for arbitrary c</td></tr>
<tr><td>[Ours]:
Optimal Algorithm for c=1,2</td><td>[Ours]: (Fast) Heuristic Algorithm for c =4,5</td></tr>
</table>

For **Universal Overcover Algorithms (UOA)**, only heuristic UOAs for c =3 have been presented [2]. Universal overcovers cover all ranges of the same size in an indistinguishable manner by not disclosing the exact mapping of the results in a subtree to the server. When documents are of the same size, the size of encrypted data sent on the network for responses will also be constant for the same r, allowing for the most information hiding out of all the variants and is hence the most interesting to explore. In this work, we contribute novel generic (c can be arbitrary) optimal UOAs.

Table 1 below provides a summary of prior work and our contributions on cover algorithms.

4.3 Novel optimal and heuristic cover algorithms

We have produced novel optimal and heuristic algorithms to compute non-universal and universal covers with the input of range (a,b) and range size, r, respectively.

Both optimal algorithms are based on a programming technique known as Dynamic Programming. We noticed that the problem of computing optimal covers can be broken down into smaller problems and solved recursively. In both algorithms, a table of base cases is set up and covers with larger parameters are computed based on covers in the table with smaller parameters based on their relationships. Thus, memoization is used to prevent repeated calculations and optimize the program.

4.3.1 Optimal Non-universal c-Overcovers Algorithm (NUOA)

We now define the NUOA, which computes optimal Non-universal n-Overcovers.

NUOA is an algorithm whose input is a range (a,b) and positive integer c, the number of nodes in the cover. It should return a valid c-overcover for the range with minimal error e.

The table T built is as follows:

T [a, b, c] → cover, where (a,b) is the range of the query and the entry is the c-overcover with minimum e and c number of nodes in it. The base cases for such a table would be all entries where c = 1 and a , b being valid positions for nodes with height of 1, $a \leq b$

Our algorithm follows from the observation that all optimal covers can be expressed as the union of smaller sets of optimal non-overlapping(range being covered does not overlap) covers. Hence, given a certain range (a, b) with a c-cover, there exists a node (1, x) within the range which will allow the cover to be separated into a range (a, x) with (c-1)-cover and range (x+1, b) with a 1-cover which do not overlap. Their union gives the original c-cover. We will be presenting the proof of optimality in a later section.

With the existence of the table T, a non-universal overcover with minimal e is given by x where $a \leq x < b$ such that:

$T[a, b, c] \rightarrow T[a, x, c-1] \cup T[x+1, b, 1]$ produces this cover

This is the key idea which we base our novel algorithm on. First, the algorithm creates the base cases for the table based on a, b. There is a recursive function which takes in input (a, b, c) and will return the cover if a cover of given parameters (a, b, c) is in the table. If not, it will construct the cover by iterating through all possible x and finding the cover constructed as per the above operation with minimal e adds the cover to the table before returning it. Note that all table requests can be replaced with this recursive function as they have the same input so there is no need for direct table calls in the cover building operation.

4.3.2 Proof of NUOA Optimality

Theorem 1 (Optimality of non-universal Algorithm): Our non-universal overcover algorithm is optimal.

Proof (Sketch). While this may seem to follow directly from the dynamic programming logic, there is an implicit assumption therein which assumes optimal covers never return "overlapping" nodes. We prove this with Lemma 1, then the optimality follows by induction on c. Both of these proofs are detailed in Appendix B. □

4.3.3 Optimal Universal c-Overcovers Algorithm (UOA)

With a height profile of the Universal cover, we can easily obtain the universal cover for an arbitrary range of given r on a given tree so the algorithm only has to return the height profile of the universal cover instead of a specific cover which can be saved and reused for valid ranges in future. Note that the universal cover must work independent of the size of the tree, thus the problem is looking for a height profile of a universal cover that remains universal even on an infinite tree.

Our UOA algorithm takes three integer inputs: r, the size of the range, c the number of nodes in the cover and b', a leaf. We assume that $r \leq c \leq b'$. This algorithm should return a height profile that can cover any range (a,b) of size r where $1 \leq a < b \leq b'$ without covering any node beyond b'. Intuitively, our algorithm considers all (a,b) of size r within a finite subtree and guarantees that the height profile works for all these ranges. We later give a mathematical proof that there exists a finite b', given in terms of r which will ensure that this height profile will also work on all (a,b) in the infinite binary tree.

A height profile is returned instead of a specific cover for a specific range because a height profile is functionally equivalent to the cover and can be applied on ranges of the same size in trees of different depth. A range is provided because each range is associated with the height profile of the universal cover and will allow the algorithm to be used more flexibly. Minor changes are needed to modify this algorithm to fit the general cover algorithm definition. Because all possible ranges of size r must have a cover of that height profile, we have to check an infinite number of ranges on an infinite tree to ensure the universality of cover.

However, the algorithm is computed over a finite subtree because results on a subtree sufficiently large can be extended to an infinite tree. Later, we will prove the results on a subtree with this algorithm can remain optimal on an universal tree.

The table T built is as follows:

T [a, b, c, **e**] → cover where (a,b) is the range of query, $1 \leq a \leq b \leq b'$, c is the number of cover nodes to be returned, **e** is a new parameter in the table and is the overhead of the cover in the entry. The entry contains a list of valid left-aligned covers (meaning that the leaf descendant node of the cover with the highest position is b and there is no e on the right of the cover) that cover the range with given e. The base cases for such a table would be all entries where c = 1.

As the height profile of universal covers for a r is the same, e is also constant with r. As covers can be broken down as an union of subcovers, all universal covers of c > 1 must consist of 2 smaller subcovers of e_1 and e_2 respectively where $e_1 + e_2 = e$ (error on the left and right). Hence, given a certain range (a,b) with an optimal universal c-cover with the minimal e, there exists there exists a node (1,x) within the range such that an the union of a left-aligned (c-1) cover with e_1 overhead that covers range (a,x) (ie no false positives after (1,x) and a right-aligned 1-cover with e_2 overhead that covers range (x+1,b) (ie false positives before (1,x+1) gives the original optimal universal c-cover. The proof that such an operation gives an optimal cover will be found in a later section. It can also be expressed as such a table operation.

With the existence of the table T, a universal overcover is given by x where $a \leq x < b$, e is that of a valid universal cover and n is an arbitrary integer representing an element in the list such that:

$$T[a,\ b,\ c,\ e] \rightarrow T[a,\ x,\ c - 1, e_1][n_1] \cup T[x + 1,\ b + e_2,\ 1,\ 0][n_2],\ e_1 + e_2 = e$$

produces a universal cover if x, e_1, e_2, n_1 and n_2 are correct. Note that $T[x + 1,\ b + e_2,\ 1,\ 0][n_2]$ contains a right-aligned cover for range (x+1,b) with e_2. Note also that there may be more than one entry in the table for a given a, b , c,e due to the nature of the table and that larger Table operations must consider different combinations of entries with given a,b,c,e and put all valid possibilities in itself.

The algorithm is broken down into 2 stages: Table Construction and Cover Finding

1. **Table Construction:** The algorithm constructs T and its base cases. It then fills in entries where 1< c <= inputted c-1 in an ascending order (2,3…c-1).
2. **Cover Finding:** The algorithm then starts with e=0 and tries to find at least one valid cover for each a, b, $r = b - a + 1,\ 1 \leq a \leq b \leq b'$ using the table operation shown above with varying x, e_1, e_2, n_1 and n_2. If it fails to do so, e is incremented until it succeeds and returns the height profile of a cover which exists for all (a, b). Such a cover is universal and optimal with a minimal e as no other cover which covers all ranges with a smaller e exists. Note that there can be more than one universal cover with the same e as there can be a list of more than one cover in one entry of T.

Using our optimal UOA with a suitable b' allows results from this algorithm which computes on a subtree that fits at least the node (1,b'), to be extended to that computed with an infinite tree and give the true universal cover without computing on a universal subtree. Given r, $h = \lceil log_2 r \rceil$, this $b' = 2^h + 3(2^{\lfloor log_2 r \rfloor})$ which is the number of leaves needed in the subtree as

shown in the proof later. Input b’ calculated from r inputted to the algorithm will allow the algorithm to compute a height profile for a universal cover that can be extended to the infinite tree.

4.3.2 Mathematical Analysis and Optimality of UOA

Before proving optimality, we must first parametrize the UOA algorithm above with b’, then show that it will terminate with a height profile that is optimal and universal to not just the subtree, but the infinite tree as well. Our result below shows that $b' = 3(2^{\lfloor log_2 r \rfloor}) + 2^h$, where $h = \lceil log_2 r \rceil$, is sufficient.

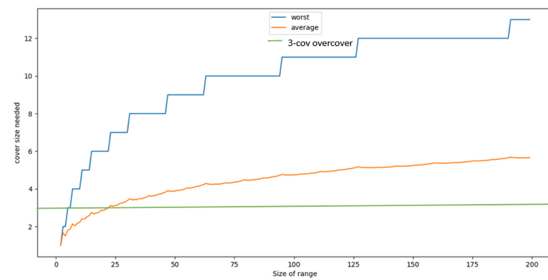

Figure 3: Graph of cover size growths against range size for exact non-universal covers and 3-covers

Theorem 2: Let $h = \lceil log_2 r \rceil$. When the UOA algorithm is run on a subtree with $3(2^{\lfloor log_2 r \rfloor}) + 2^h$ leaves, the output is an optimal height profile over the infinite binary tree.

Proof (Sketch). This proof proceeds by first bounding the size of the universal overcover (including false positives), which we do in Lemma 2. We then show that covers within our subtree can be “moved” to an analogous cover exactly 2^h leaves away in Lemma 3. With this, we can use both lemmas to prove this result by induction on groups of 2^h leaves. We state and prove Lemmas 2 and 3, then provide a full proof of this in Appendix B. □

4.5 Optimal non-universal 1,2-overcover algorithm.

Presently, Demertzis et al. have presented only a heuristics algorithm for non-universal 1-overcovers on an augmented tree. Our novel optimal NUOA for c=1,2 can be found in Appendix C.

4.6 Heuristic universal 4,5-overcovers algorithms

We have also constructed heuristic universal 4,5-overcovers algorithms which were constructed based on observation and experimentation. These heuristics run much faster than our optimal algorithm and may be of interest in high-efficiency RSE implementations. These algorithms have been tested to be empirically equivalent to the optimal dynamic programming algorithm until values of r = 200. The algorithms use manipulation of the binary string of r, similar to the universal 3-overcover heuristic algorithm by [2], but uses a more complicated approach due to the complexities of universal 4,5-overcovers. The algorithms are thus linear time and are extremely fast. Further details on the algorithms can be found in the Appendix D.

4.7 Evaluation of algorithms

Cover size growth We benchmarked the worst case and average cover size growth of non-universal exact covers. As shown in figure 3 in section 6 of the appendix, both cases grew logarithmically with the worst case having a large constant. At r=200, e of worst case = 13 and e of average = 5.8 whereas fixed cover size allows for constant and significantly

smaller cover sizes. The differences between the worst and average case implies there is a very large variance in cover size of exact covers which gives an extra identifier about a query to an adversary.

Overhead of non-universal overcovers. We benchmarked the average overhead percentage of non-universal overcovers over a range of a fixed size. Overhead percentage is e/r and we modeled the minimum subtree needed to cover desired range size and found the average overhead percentage over ranges of that fixed size. As shown in Figure 4, the average overhead percentage of all cover sizes plateaus suggesting that the tradeoff is less significant for larger ranges. Overcovers of larger sizes also perform significantly better than those of smaller sizes as the average overhead percentage decreases with an increase in cover size.

Overhead of Universal overcovers We benchmarked the growth of overhead percentage with the increase in r for universal overcovers as shown in figure 5. Again, it can be observed that larger overcovers perform strictly better although for some values of r, the e of a c cover may be equal to that of its c-1 counterpart. There is also an average increasing improvement in overhead percentage as r increases. Notice that the graph for each cover is not a neat curve but a line that dips up and down. We believe this due to the constant e for universal overcovers. This also allows certain ranges to be particularly advantageous (eg. range 25-30 has a roughly halved overhead percentage compared to range 20-25 for c=2 despite having larger r).

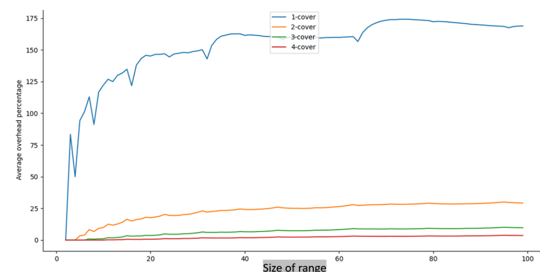

Figure 4: Graph of overhead (e/r) against range size for non-universal c-overcovers

Speed of computation We benchmarked the average time of computation in seconds of a universal 3-overcover of a naive brute force algorithm against our optimal UOA as shown in figure 6 of the appendix and found that our algorithm performs significantly better. When r=25, the brute force took 10.23s while our optimal UOA took 0.91s to find the right cover and found the correct cover 11 times faster . We believe that our UOA will be even more times faster than the brute force for higher values of r.

Lastly, we also benchmarked the heuristic 3-overcover algorithm of [2] against our optimal UOA and found that it is optimal until r=200.

4.7.1 Discussion

To summarize, our algorithms show that overcovers are better than exact covers in terms of information hiding and that overcovers can be optimally found. Hence, our benchmarking has shown that the tradeoff in overhead and computation time when using the more secure overcover algorithms is acceptable and that overcovers can be applied practically with RSE.

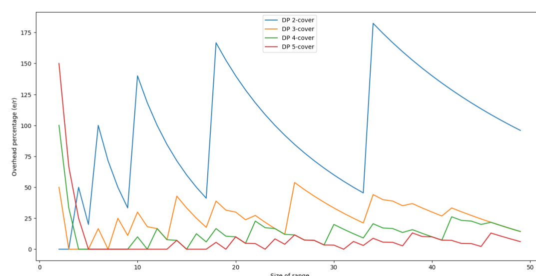

Figure 5: Graph of overhead percentage (e/r) against range size for universal c-overcovers

5. Future Work

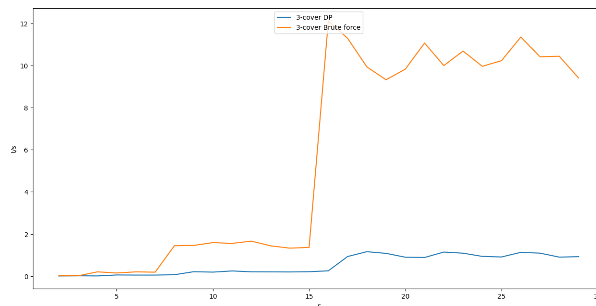

Figure 6: Graph of computation time against range size for our UOA and brute-force for c=3

While we have proven the optimality of our novel algorithms, they can be further optimized. Further algorithmic optimizations could include better selection of table entries generated. A possibility is reducing search size by tightening the bound for r+e for UOA. We believe that r+e can be further minimized for c greater than 2.

We have also considered other approaches for optimal algorithms. An approach could be to build a c-cover from (c-2)-cover, which may be faster. It may be possible that a provable non-dynamic algorithm that could run in linear time can be created, not unlike the heuristics we designed.

We believe our heuristics for 4,5-overcovers can be proven optimal in future work. Even though we believe universal 4,5-overcovers are the best in terms of balancing information hiding and overhead size, larger cover sizes may be useful for certain applications.

An interesting improvement could be to create algorithms that consider documents of non-equal size. This would generalize the problem to minimize overhead in RSE schemes where different amounts of information are indexed at each leaf.

6. Conclusion

RSE is used in the outsourcing of large amounts of sensitive data to an untrusted cloud. In this work, we first improve upon an RSE scheme from the literature using recursive hashing, and experimentally demonstrate its improved storage efficiency. We then revisit overcover generation, an algorithm used within this and other RSE schemes, and provide the first provably optimal generic algorithms for generating these covers. We provide mathematical proofs of the algorithms' optimality, and experimental evaluation of the algorithms' efficiency, concluding that they are able to achieve better security with low overheads in practical settings. We conclude with several linear-time novel heuristics for fixed-size overcovers which we prove optimal on limited ranges. Through these novel techniques, our research enables the adoption of RSE for more secure and functional Cloud storage.

Acknowledgements

We would like to thank our mentor, Dr Ruth Ng Ii-Yung, for her continued support and guidance throughout this project. We would also like to thank Mr Choo Jia Guang and Mr Daren Khu Boon Tat for their support.

References

[1] Demertzis, I., Papadopoulos, S., Papapetrou, O., Deligiannakis, A. & Garofalakis, M. Practical private range search revisited. p3-7 *Proceedings of the 2016 International Conference on Management of Data* (2016). doi:10.1145/2882903.2882911

[2] Faber, S. *et al.* Rich queries on encrypted data: Beyond exact matches. p3-8 *Computer Security -- ESORICS 2015* 123–145 (2015). doi:10.1007/978-3-319-24177-7_7

Appendix A: Description of R-RSE

Our R-RSE Scheme is a type of document lookup scheme.

Note that in the description, nodes in a binary tree are referred to with an integer. Depth of a node is given by $\lfloor \log 2(i) \rfloor + 1$ where i is the node's integer.

Document lookup trees are defined for fixed depth binary trees, for convenience we use this notation

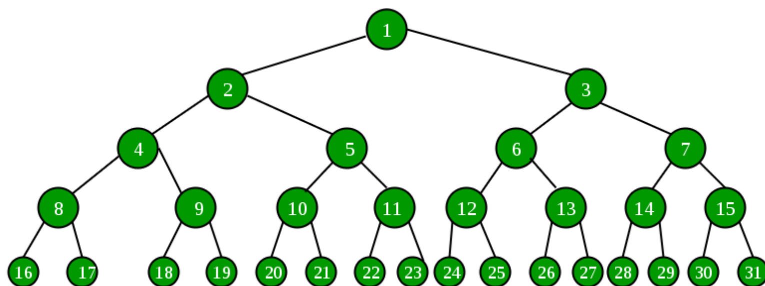

Figure 7: Alternative Binary Tree

A document lookup scheme DL defines the following:

- the key length: DL.kl
- The token length: DL.tkl
- The document length: DL.dl
- Number of documents: Dl.n
- An encryption Algorithm: DL.Enc(K,DS) → EDS
  - K is in $\{0, 1\}^{DL.kl}$
  - DS is in $(\{0, 1\}^{DL.dl})^*$ #set of documents to be encrypted
    - We refer to $d=\lceil \log_2(|DS|) \rceil$ as the "depth" of the tree
  - EDS is in $\{0, 1\}^*$
- A Token Algorithm: DL.Token(K,a,b) → tk U $\{\perp\}$
  - a,b are positive integers, a where 1<= a <= b <= 2^d
  - tk is in $\{0, 1\}^{Dl.tkl}$
- Search Algorithm: DL.Search(tk,EDS) → CS
  - EDS is the output of DL.Enc
  - tk is the output of DL.Token under the same key where tk$\neq\perp$
  - CS is a in $\{0, 1\}^*$
- Decryption Algo: DL.Dec(K,CS) → D
  - D is in $(\{0, 1\}^{DL.dl})^*$

Additionally, it should satisfy the following correctness condition:

- DL.Dec(K,DL.Search(DL.Token(K,a,b),DL.Enc(K,DS))) = (DS[a],...,DS[b]) if DL.Token(K,a,b) $\neq\perp$
  - Where DS[a] is the ath document in DS

When a DL scheme is used with a cover-generation algorithm, it becomes a RSE scheme. R-RSE provides the functionality of returning descendants of a node and it is up to the discretion of the client to choose what cover-generation algorithm to use. Nodes in cover generated can be tokenized and then sent to the server.

R-RSE is an example of a Document Lookup Scheme:
Ratcheting Document Lookup Scheme (R) is built using Cryptographic Hash Function CHF and Symmetric Encryption scheme SE:

- R.kl = CHF.kl+SE.kl
- R.tkl=CHF.ol
- R.dl can be any fixed constant
- K||K_m ← K where R.kl=|K|

cover(a,b,c) → $\{n_1, \ldots, n_c\}$ is a cover-generation algorithm given by user
get_hash(K_m,i) # i is a node number

- If H[i] not empty
    - return H[i]
- If i=1
    - h ← CHF(K_m,0)
- Else If i % 2 == 0
    - h ← CHF(get_hash(⌊i/2⌋,2))
- Else
    - h ← CHF(get_hash(⌊i/2⌋,1))
- H[i] ← h
- Return h

R.Enc(K, DS)

- Initialize dictionary EDS where EDS[tk]=⊥ for all tk
- (D_1,D_2,...,D_n) ← DS
- d ← $\lceil \log_2(n) \rceil$ +1
- Initialize dictionary H
- For i=1…n :
    - Pick IV randomly
    - EDS[get_hash(K_m,2^(d-1)+i)] ← IV||SE.Enc(K,IV,D_i||n)
- Return EDS

R.Token(K_m,a,b) → tk

- c = cover(a,b)
- get_hash(K_m,c) → tk

R.Dec(K,C) → DS

- Split C string back into list of individual C_i
- SE.K||CHF.K ← K
- Initialize an empty list DS
- For all i:
    - If C_i = ⊥ then return ⊥

- IV||C_SE ← C_i
- D_m ← SE.Dec(SE.K,IV,C_SE)
- if SE.Dec throws an error
    - return ⊥ if SE.Dec throws an error
- else:
    - D_i||n ← D_m
    - If a <= n <= b:
        - Add D_i into DS
- Return DS

R.Search(tk,EDS):

- Return recurse(⌈$\log_2$|EDS|⌉),[tk])

recurse(d,[list]):

- If d=0 then return error
- If EDS[list[1]] not equal to ⊥ then return [EDS[l] : l in list]
- Else return recurse (d-1, [CHF.Ev(l,1): l in list] ++ [CHF.Ev(l,0):l in list])

Additionally, notice that correctness is achieved because R.Dec(K,R.Search(R.Token(K,a,b),R.Enc(K,DS))) = (DS[a],...,DS[b]) if DL.Token(K,a,b) ≠⊥

- Where DS[a] is the ath document in DS

Appendix B: Mathematical Analysis of Algorithms

Our NUOA algorithm can be proven to be optimal (i.e. return a c-overcover with minimal error). While this may seem to follow directly from the dynamic programming logic, there is an implicit assumption therein which assumes optimal covers never return "overlapping" nodes. More precisely, define a cover C as overlapping if there exists two nodes with a common descendent. Note that the additional copy of this common node is included in the overhead of the cover C.

Lemma 1 (Non-overlapping lemma): Let C be a c-overcover of range (a,b) with error e. Then there exists a non-overlapping c-overcover of (a, b), C', with error e'< e.

Proof. Let n, n' be two overlapping nodes in C of heights h, h' respectively where $h \geq h'$. Notice that n' must be a descendant of n, which means that $C/\{n'\}$ is a c-1-overcover of (a,b) with error $e - 2^{h'}$. To get a c-overcover, we either replace any node in C of height >1 with its two children (such a node must exist because we assume $c \leq b - a + 1$). We construct C' by repeatedly doing this until the cover is non-overlapping. □

Now we can show that our first algorithm generates optimal non-universal overcovers.

Theorem 1 (Optimality of non-universal Algorithm): Our non-universal overcover algorithm is optimal.

Proof. We can prove this result by induction on c. By the algorithm's base case, it is optimal for 1-overcovers.

Now suppose the algorithm is optimal for all overcovers of size less than c. Let C be an optimal c-overcover for range (a,b) which achieves error e and C' be the c-overcover of (a,b) returned by the algorithm achieving error e'. If C=C' then we are automatically done. otherwise, since we know that C (by lemma 1) and C' (by the algorithm definition) are non-overlapping, there must exist x such that C = D ∪ {n} where D is an c-1-overcover of (a,x) with error e_1 and {n} is a 1-overcover of (x,b) with error e_2=e-e_1. Similarly, x', D', n',e_1',e_2' must analogously exist for C'. By the inductive hypothesis, when computing C' in the algorithm, T[a,x,c-1], T[x,b,1], T[a,x',c-1], T[x',b,1] are all populated with optimal overcovers. Since C' will be computed by taking the minimum e over the covers associated with each value of x, then it must be true that e=e', or the algorithm would have returned C instead of C'. Therefore, the algorithm is optimal for c-overcovers. □

We move on to the analysis of the universal overcover algorithm. To demonstrate its optimality, we must first parametrize the UOA algorithm above with a finite subtree size, then show that it will terminate with a height profile that is optimal and universal to not just the subtree, but the infinite tree as well.

Lemma 2 (Error upper bound): Let H be the height profile of a universal c-overcover of range r with error e. Then we have the following tight (i.e. RHS cannot be reduced further) bound.

$r + e < 3(2^{\lfloor log_2 r \rfloor})$

Proof. Let (a,b) be such that b-a+1=r. Then, we define an exact cover C={n,n'} of (a',b') where$a' = y(2^{\lfloor log_2 r \rfloor})$ and y is the highest integer such that a'<=a. Intuitively, a' is the left-most node of the height $\lfloor log_2 r \rfloor$ subtree that a is in. Then set $b' = a' + 3(2^{\lfloor log_2 r \rfloor}) - 1$. Notice then that (a',b') spans exactly the leaves of three height $\lfloor log_2 r \rfloor$ subtrees which means that {n, n'} can be selected to have a height profile $\{\lfloor log_2 r \rfloor, \lfloor log_2 r \rfloor + 1\}$. This height profile provides a universal 2-overcover (not necessarily of minimum error) which bounds r+e as above. Hence, any optimal universal c-overover of size r must also satisfy the above.

This bound is tight when we set c=2 and $r = 2^{k+1} - 1$. Notice that the lowest error 2-cover of the range $(2^k - 1, 3(2^k) - 2)$ (2^k-1,3*2^k-2) is {(k+1, 1), (k, 3)}, which achieves the above bound, so the universal 2-cover must do so too. □

Lemma 3: There a cover C of range (a,b) if and only if there is a cover C' with the same height profile of range $(a + 2^h, b + 2^h)$ where $h = \lceil log_2 r \rceil$ and r = b-a+1

Proof. Let C={(h_1,v_1),...,(h_n,v_n)} be an exact cover of (a',b') and note that these nodes are part of the subtree of ancestors of (a',b'). We design C' to take on the same positions within the analogous subtree for $(a + 2^h, b + 2^h)$. We do this by setting C' = {(h_i,v_i + 2^{h-h_i+1}) | (h_i,v_i) ∈ C}. Notice that this cover is well-defined since 2^h>=r by the definition of h and $r \geq 2^{h_i}$

since C covers r so, by Lemma 1, each node in C covers a partition of the range. Notice that if (h_i,v_i) covered the partition (x, y) of (a',b'), then

(h_i, v_i + 2^{h-h_i+1}) covers (x+2^h,y+2^h). Thus, C' is an exact cover of (a'+2^h, b'+2^h) and an overcover of $a + 2^h, b + 2^h$).

With these we have the parameters which will guarantee that the UOA algorithm above will terminate with an optimal result: a subtree with no more than $3(2^{\lfloor log_2 r \rfloor}) + 2^h$ leaves, where $h = \lceil log_2 r \rceil$. This is shown below.

Theorem 2: Let $h = \lceil log_2 r \rceil$. When the UOA algorithm is run on a subtree with $3(2^{\lfloor log_2 r \rfloor}) + 2^h$ leaves, the output is an optimal height profile over the infinite binary tree.

Proof. We begin by noting that the algorithm must terminate with some height profile being returned because Lemma 2 bounds the size of r+e in an (infinite tree) universal c-overcover by $3(2^{\lfloor log_2 r \rfloor})$. Since our subtree exceeds this size, some (subtree) universal c-overcover will be returned with error at most $3(2^{\lfloor log_2 r \rfloor}) - r$.

The optimality of the UOA algorithm on the subtree follows from a similar inductive logic as in theorem 1 for each value of e. The only difference is that we must incorporate the algorithm's consideration of all possible splits of e into e_1,e_2 into the inductive reasoning. If we know that only a finite number of e will be considered (from lemma 2), this means the UOA will terminate for some value of e, and when this happens the result is an optimal (minimal overhead) universal c-overcover from the above inductive logic.

What remains is to show that the UOA algorithm on the subtree is sufficient to generate a universal height profile for the infinite tree. To see this, let (a,b) be any range of size r in the infinite tree with error e. Notice that an optimal universal c-overcover of (a,b) is an exact cover of range (a',b') of size r+e. We do this by showing that there must exist a range $(a' - k(2^h), b' - k(2^h))$ for some integer k which is entirely contained within the subtree. We set $a' - k(2^h)$ such that $1 \leq a \leq 2^h$ and note that that places $b' - k(2^h) \leq 3(2^{\lfloor log_2 r \rfloor}) + 2^h$ by lemma 2. By applying lemma 3 k times, we see that an optimal universal c-overcover of (a',b') is also an c-overcover of $(a' - k(2^h), b' - k(2^h))$ with identical error. For this reason, by considering all possible ranges in the subtree, we consider all possible height profiles for all ranges in the infinite tree. □

Appendix C: Optimal non-universal 1 and 2-overcover algorithms

We will now present a novel optimal non-universal 1-overcover algorithm.

Observing the relation of node numbers with its position on the binary tree, reading from the left of the number in binary form, a 0 shows a left in the tree and 1 to the right. Thus, the similarity between the 2 most extreme nodes in a range (a,b) from the left to the right gives the smallest 1-cover. Eg. (given (8,11) binary of 8 is 1000 and binary of 11 is 1011. Their 1-cover node is 10 in binary, which is 2 in decimal)

We will now present a novel **optimal non-universal 2-overcover algorithm** based on the previous algorithm.

Given an optimal 1-overcover O for (a, b), if we split O into the 2 nodes directly connected below it by 1 vertex, we will obtain 2 1-covers which covers the ranges (a+e1, x), (x+1, b+e2) respectively. The union of the optimal non-universal 1-overcover for ranges (a,x), (x+1,b) gives an optimal 2-overcover as the 1-overcovers cover continuous ranges.

Appendix D: Heuristic Universal 4 and 5-overcover algorithms

Ranges covered by a universal overcover are represented by a set of cover heights. A cover height hence represents a subrange of size 2^h of r+e. The entire range can hence be expressed as a sum of all 2^h, where h ∈ cover. Similarly, one bits in the binary string of r represent powers of 2, and a sum of these powers of 2. Hence, this gives the intuition that manipulating the binary string of range r will give a universal overcover

Our heuristic algorithms are **4-universal**(r) and **5-universal**(r), where r is the range size. Below, lists position_one and position_zero refer to the position of all ones and all zeros in a binary string respectively, arranged in a descending order. Additionally, the position of a bit is defined as 0 for the rightmost bit and 1 for the second rightmost bit and so on. Multiple ones may exist in the same position, while only one zero can exist in each zero position and the position of ones and zeroes are mutually exclusive.

position_counter(r):

Checks and converts r to even, converts r to binary and stores 1 and 0 positions into position_zero and postion_one

- **if** r is odd, r = r-1

convert r to binary string

- position_one = [position of all ones in binary string]
- position_zero = [position of all zeros in binary string]
- **return** position_one, position_zero

split(position_ones)

Split a one in the highest position into two ones of lower by one position.

- duplicated = position_one[0] -1
- **remove** position_one[0]

position_one = [duplicated] + [duplicated] + position_one

sort position_one in descending order

return position_one

merge(position_ones, position_zeroes) #Merge two ones in the lowest positions into a one of a higher by one position of the second lowest one

```
replacement = position_one[-2] + 1
remove the two ones in lowest position
add their position value into zero_position
position_one.append(replacement)
return position_one, position_zero
```

zero_chain(position_one, position_zero) #Move 1s across 0s of one higher position from right to left, recursively repeats

```
for all i in position_one
    if i+1 in position_zero
        replace i with i+1
        remove i+r from position_zero
        return zero_chain(position_one, position_zero)
return position_one
```

4-universal(r)

```
if r<=15
    return universal(r,4) #the dynamic programming algorithm
get position_one, position_zero from position_counter(r)
if position_zero[0] = position_one [0] -1 and position position_zero[1] = position[0] -2
    position_one = split(position_one)
if len(position_one) > 3
    if position_zero[0] == position_one[0] -1:
        position_one = split(position_one)
    do merge(position_one, position_zero) until len(position_one) = 3
    position_one.append(0)
    position_zero = [x for x in position_zero if x not in position_one]
    position_one = zero_chain(position_zero, position_one)
    universal_cover = position_one
elif len(position_one) < 3
```

```
        do split(position_one) until len(position_one) = 3
        position_one.append(0)
        position_zero = [x for x in position_zero if x not in position_one]
        universal_cover = zero_chain(position_zero, position_one)
elif len(position_one) = 3
        position_one.append(0)
        position_zero = [x for x in position_zero if x not in position_one]
        universal_cover = zero_chain(position_zero, position_one)
return universal_cover
```

5-universal(r)

```
if r<=24
        return universal(r,5) #the dynamic programming algorithm
get position_one, position_zero from position_counter(r)
if position_zero[0] = position_one [0] -1 and position position_zero[1] = position[0] -2
        do split(position_one) until the above is no longer true
sort position_one in descending order
if len(position_one) > 4
        if position_zero[0] == position_one[0] -1:
                position_one = split(position_one)
        else do merge(position_one) until len(position_one) = 4
        position_one.append(0)
        position_zero = [x for x in position_zero if x not in position_one]
        position_one = zero_chain(position_zero, position_one)
        universal_cover = position_one
elif len(position_one) < 4:
        do split(position_one) until len(position_one) = 3
        position_one.append(0)
```

```
        position_zero = [x for x in position_zero if x not in position_one]
        universal_cover = zero_chain(position_zero, position_one)
elif len(position_one) = 4
        if position_zero[0] == position_one[0] -1:
                do split(position_one)
                do merge(position_one, position_zero)
        position_one.append(0)
        universal_cover = zero_chain(position_zero, position_one)
return universal_cover
```