THE AUGMENTED BINARY TREE RECONSTRUCTION PROBLEM: NEW ALGORITHMS AND DIRECTIONS

Julian Tay Yu Sheng, Tey Yik Jin Ryden, and Ruth Ng Ii-Yung
Dunman High School, 10 Tanjong Rhu Rd, Singapore 436895
Anglo Chinese School (Independent), 121 Dover Road, Singapore 139650
DSO National Laboratories, 12 Science Park Drive, Singapore 118225

Abstract. In this project, we define and explore the problem of reconstructing augmented binary trees. We provide two algorithms which allow an adversary to infer random identifiers associated to leaf nodes of such a tree from information about groups of nodes associated to "covers" of leaf node ranges. We conclude by linking these algorithms to adversarial attacks on range searchable encryption, a cryptographic scheme used to make range queries to outsourced data on untrusted Cloud storage providers, thereby demonstrating the practical uses of our work in the security evaluation of such schemes.

1 Introduction

In graph theory, tree graphs are the subject of many interesting algorithms with far reaching applications. In this work, we focus our effort on fixed-depth augmented binary tree graph and define the new algorithmic problem of "reconstructing" such trees from "node queries".

More concretely, an augmented binary tree adds new nodes to the intermediate levels of a complete binary tree. These nodes have no parents and have two consecutive children in the layer below who do not have a common parent in the binary tree. The problem we define assumes that a unique random identifier is assigned to each leaf node which is unknown to the adversary. By observing node queries to this tree, the adversary gains information about which identifiers are associated to descendants of particular nodes. The adversary then tries to make inferences from a series of such queries about the identifier of each leaf node, thereby reconstructing the tree. In our work, we provide two algorithms to address this reconstruction problem, and do a mathematical and experimental analysis of them.

This seemingly random problem has a direct application in the security evaluation of Range Searchable Encryption (RSE), a type of cryptographic scheme used for outsourcing a database, and the processing of range queries to said database to an untrusted Cloud server.

Our problem statement and algorithms can be applied by an eavesdropper adversary (e.g. a nosy Cloud administrator) on real-world RSE encryption schemes [1, 2] to try to glean information about sensitive stored data, in spite of end-to-end encryption applied by the client. However, depending on the RSE application's characteristics, such as frequency of queries and distribution of data, the success rate of such an adversary may differ greatly. As such, our algorithms are impactful as they can be used in the security analysis of real-world cryptographic schemes, and understand better their practical security in realistic use-cases.

1.1 Our Contributions

Our main research contributions can be summarized as such:

– We designed an algorithm which incrementally reconstructs the tree from the inferences that can be drawn from adjacent nodes. We call this the Recursive Neighbour Search (RNS) algorithm. We demonstrate that this algorithm is only a heuristic, as there are a family of inferences that it is unable to draw from these adjacency rules, so the algorithm output is suboptimal.

– We then provide another, completely distinct algorithm for the same problem, which only makes one pass across the leaf nodes and exploits union and intersection properties of the queries made. We call this the One-pass Union-Intersection Search and it is optimal.
– We implemented both algorithms in Python, then ran simulations to test for accuracy and time, concluding that the One-pass algorithm is both more accurate (owing to its optimality) and more efficient than RNS.
– We applied our problem to the cryptanalysis of RSE, and demonstrated the potential impact of our research in the security evaluation of RSE schemes. People who use RSE can use our algorithms to stress-test their implementations, better understand their security against realistic adversaries in different settings.

2 Augmented Binary Tree Definition

Binary trees are hierarchical data structures with nodes, with two children nodes per parent node, connected by lines defined as edges. Every node has a maximum of two children nodes, with nodes higher on the tree labelled as parent nodes and the nodes at the final depth being leaf nodes. Leaf nodes have no children nodes, signifying a termination of the binary tree. In our notation, we label the topmost node as node 1, and notate the rest of the nodes numerically, horizontally from left to right, up to down. Every node x has two children nodes $2x$ and $2x + 1$. For example, in Figure 1 node 2 has children 4 and 5.

We define the depth of the tree as d, the number of "rows" in the tree, ignoring the root, and the depth of a node x as the length d_x of the minimal path between it and the root. Note that depth d of a tree contains the nodes from 2^d to 2^d-1. For example, the depth of node 5 in Figure 1 is 2.

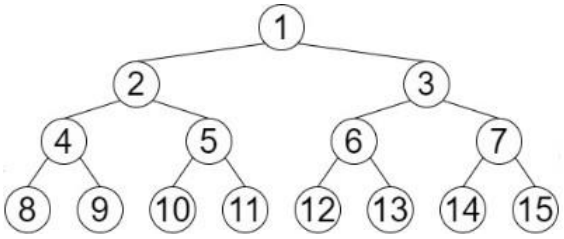

Fig. 1. A diagram of a complete binary tree with depth d = 3.

Our research studies augmented binary trees, which modifies the above by adding new "augmented" nodes. These nodes are denoted with decimal number of the form $i + 0.5$ (for integer i) and are added "between" the nodes i and $i + 1$, for all i except those of the form $i = 2^d - 1$ for all integers d. The children of each such node are the two integer nodes "below" it with no common parent. Figure 2 illustrates the analogous augmented binary tree for $d = 3$.

In this whole paper, we will be dealing with augmented binary trees, as our reconstruction problem will be centred upon the nature of augmented binary trees.

2.1 Covers

Another large aspect of the project is the term of range covers. A cover of a range (a, b) is defined as the set of nodes, where the union of the descendent leaf nodes is exactly the set of leaves $\{a, a + 1, \ldots, b\}$. As the term might insinuate, the set of nodes must "cover" all of

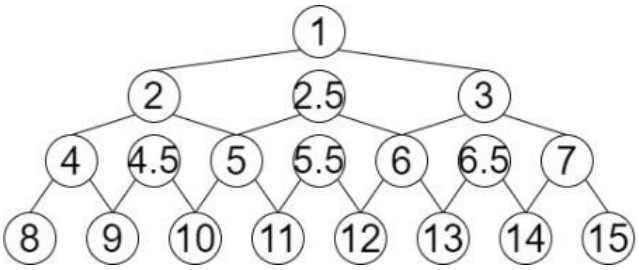

Fig. 2. A diagram of an augmented binary tree at depth d = 3

the range that is required, by ensuring that the descendent leaf nodes of each node in the set of nodes are all within the range.

We say that C, a subset of the nodes in an augmented binary tree is a cover of range [a,b] if

$$\bigcup_{n \subseteq C} \{\text{n'} \mid \text{n' is a leaf descendent of n}\} = \{a, \ldots, b\}$$

While a range might have multiple covers, applications minimize bandwidth by preferring the minimal one (i.e. the one with the least nodes). For example, in figure 2, the range [8,12] has minimal cover $\{2,12\}$, as node 2 has descendent leaf nodes 8 to 11, and the final node 12 is to finish covering the last section of the range, which is 12.

3 Reconstruction Problem

Now, we will discuss the actual problem of reconstruction. To a full and complete depth d augmented binary tree, we randomly assign 2^d unique identifiers (notated as capital-letter alphabets) to the leaf nodes of the tree (e.g. node 4 is mapped to B). We define a query on a node as a tuple (x, P), where x is the node number and P is the set of the unique identifiers associated to the descendent leafs of that node, i.e. querying node 2 gives us (2,{A, B}). As such, our problem is to design an algorithm that when inputted a depth d and q queries, e.g. (n_1, P_1), the output should be the list of as many leaf nodes and their identifiers as possible computed based on the queries given.

E.g., in Figure 2, given node 5 and 5.5, with identifiers {H,J} and {H,K} respectively, a way an algorithm can work is to recognise that 5 and 5.5 share a child 11 and deduce information about the identifier of leaf node 11 which may not be queried, hence the identifier of 11 should be {H}, which is the intersection of identifiers of 5 and 5.5. Thus, the algorithm can have an input of 5 and 5.5 and their respective identifiers, with an output of (11, {H}).

3.1 Recursive Neighbour Search Algorithm (RNS)

The intuition behind our first algorithm is to incrementally draw inferences about nodes in the tree via simple equations, gradually reconstructing the whole tree. Our equations all relate to how each node is associated to its neighbours, in hope that these inferences will coalesce into inferences about the entire tree. To better understand the algorithm, We describe these equations then discuss how to extend this toa reconstruction algorithm.

SHARED CHILD EQUATION In an ordinary binary tree, each node has a single parent. However, in an augmented binary tree, nodes are now able to have two parents, due to the existence of augmented nodes. Thus, it can be concluded that for every two parent nodes x and $x + 0.5$ who share the same child $2x + 1$, $P_{2x+1} = P_x \cap P_{x+0.5}$, as x and $x + 0.5$ both cover the same identifiers of the shared child. Referring to Figure 2, if the algorithm was given two nodes 5 and 5.5, e.g. $5 : \{A, B\}$ and $5.5 : \{B, D\}$, our algorithm will determine that node 11 will contain the intersection of the identifiers from its two parent nodes, and our algorithm will update nodedictionary with $11 : \{B\}$. It is important to note that this equation is the basis of the 2nd algorithm, which we will go into further detail in Section 3.3.

SHARED PARENT EQUATION (ONE CHILD ONE PARENT) As every two nodes share a parent, logically, all identifiers from the parent include the identifiers of both children. Hence, when the identifiers from one parent and one child are known, we are able to compute the identifiers of the other child. Mathematically, this is depicted as $P_{2x} = P_x \setminus P_{2x+1}$. Referring to Figure 2, if the algorithm was given two nodes 5 and 11, e.g. $5 : \{E, F\}$ and $11 : \{B\}$, the algorithm will use the identifiers of the parent node to subtract the set of identifiers of the child node, computing that node 10 has identifiers {F}.

SHARED PARENT EQUATION (TWO CHILDREN) Similar to the previous equation, due to the fact that both of the children's identifiers added up equals to the identifiers of the parent node, mathematically, $P_x = P_{2x} \cup P_{2x+1}$. Referring to Figure 2, if given nodes 14 and 15 and their respective identifiers {W} and {Y}, the algorithm is able to infer that node 7 has identifiers {W,Y} by adding the identifiers of both its children.

EXTENDING EQUATIONS TO RNS The algorithm repeatedly searches the tree for relations between nodes, using a node's descendant leaves' identifiers to infer its neighbours' identifiers. Our algorithm is built around these three equations representing these inferences. Using these three equations, the algorithm repeatedly scans the tree for any scenarios where these equations can be applied, then modifies the tree by updating new nodes which can be computed from the equations. With these new nodes and information, the algorithm can in turn infer more about the neighbours, and repeats until no further nodes can be computed. The details of this algorithm can be found in the appendix A, with the detailed pseudocode. Uses dictionary to keep track of inputted nodes

The final step entails the extraction of all depth $d-1$ nodes and their subsequent identifiers, by measuring the length of all P in the dictionary and outputting all P of length 2.

MATHEMATICAL ANALYSIS OF RNS Algorithms usually are described as optimal, or heuristic. We refer to optimal algorithms as always returning the correct answers and algorithms that do not meet this criteria as heuristic. Returning to our algorithm, it has a limitation of a certain specific edge cases which we observed in Figure 3. From the figure, we can observe that for the first case, assuming the algorithm is given the nodes 4, 1, 2.5, similar to the shared parent equation (one child, one parent), this case can be treated as one parent, two children. Node 1 is the parent, and one can deduce that the identifiers in 1 which are not in 4, 2.5, must be in node 7. However, due to the infinite number of these edge cases, our algorithm is unfortunately unable

to account for all of these cases, making this algorithm a heuristic and approximate algorithm, instead of the optimal one we desire. Thus, we attempted to create a second algorithm which we could define as optimal.

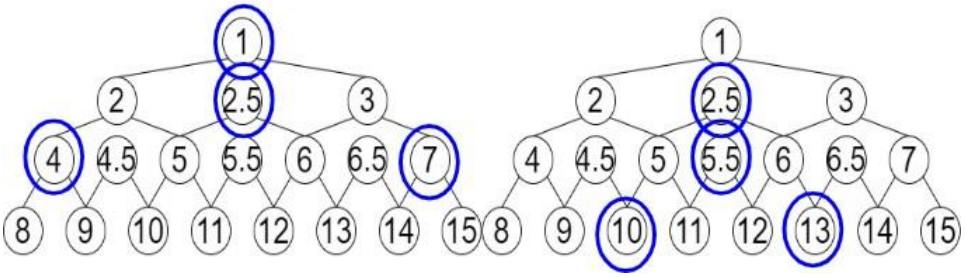

Fig. 3. Example edge cases demonstrating that RNS is a heuristic algorithm

3.2 One-pass Union-Intersection Search Algorithm

Next, we discuss the details of our second algorithm, which we construct due to the fact that RNS is not optimal. Our aim is to design an algorithm which is optimal, and we attempt to design one that is more efficient than RNS, to find the best solution to our problem. Our next algorithm is called the One-pass Union-Intersection Search Algorithm as it exploits certain properties of nodes.

Firstly, x is descendent of y if there exists a series of nodes connected by edges who are parents of each other up until y. z is ancestor of y if there exists a series of nodes connected by edges who are children of each other up to y. Secondly, every leaf node's identifier appears in all of its ancestor nodes' identifiers, and also does not appear in any of its non-ancestor nodes' identifiers

The main intuition behind this algorithm is that every child node has more than one ancestor which cover the node. This would mean that all we require is to find the intersection of all identifiers belonging to these nodes and subtract the union of all identifiers belonging to nodes that do not cover the specified node. The intersection of identifiers of parents of a specified node would return the identifiers of that specified node. Hence, expanding this to the whole tree, searching the tree for as many ancestor nodes of the specified node and taking their intersection would return a set which contain the identifiers of the specified node, but yet may also contain other identifiers which are not in the specified node, if the ancestor nodes selected are too high in the tree. This is because nodes which are higher on the tree contain more identifiers than those lower on the tree. Thus, subtracting the identifiers of nodes which do not cover the specified node are important as well to obtain solely the identifiers of the specified node. The mathematical equation is as follows:

$$P_x = \bigcap_{n \in I} P_n \setminus \bigcup_{n \in U} P_n$$

For each leaf node, we register all the higher nodes which cover each leaf node, which we can denote as a set I where I_I again refers to the length of set I. In addition, all nodes which do not cover the specified leaf node can be denoted as set U, and the length of set U as I_U.

To provide an example, referring to Figure 2, we observe node 10. It has an augmented node parent 4.5 as well as other nodes which covers node 10, which are 5, 2, 2.5, 1. The algorithm would note the nodes provided, which e.g. could be [2, 2.5]. As such, to calculate P_{10}, the

algorithm would find the intersection of P_2 and $P_{2.5}$, which are P_5. However, we require P_{10} instead, but the algorithm does not end here, continuing instead, which takes the intersection of identifiers and conducts a set subtraction of the intersection and union of all other nodes which are given and not covering node 10. E.g. we are also given node 5.5. Thus, the algorithm computes the intersection of identifiers, which is P_5, and then computes $P_5 \setminus P_{5.5}$, hence obtaining a subtraction of P_{11}, giving P_{10} as the final solution.

One of the main reasons why this algorithm is so favourable is due to the fact that the time taken to run this program at a fixed depth d is constant for as many numbers of queries, as the program repeats for 2^d times no matter the number of queries given. It also makes use of a lookup table to efficiently store ancestor nodes of every leaf node, as well as nodes that are non-ancestors.

In conclusion, this algorithm allows inputs of similar parameters to the previous algorithm and outputs depth $d-1$ identifiers, and it is optimal due to the fact that it scans the whole tree, using all available information to deduce each node's identifiers.

4 Methodology

Firstly, we discuss how we implement our algorithms and how we extract data from experiments done. Our main implementation was done through programming, where we program our algorithms and test for the time taken, as well as how various factors affect the success rate. Although mapping each leaf node to the correct file is most important, we instead define our success rate to be the percentage of nodes at depth $d-1$ which were reconstructed instead of depth d. This factor was chosen because ranges are inputted into the algorithm instead of actual nodes, the probability of receiving depth d augmented nodes is rather small, with only depth d regular nodes being returned, which limits the orderings of identifiers to two different possibilities. Thus, unfortunately, with this limitation, our only option is to test for depth $d-1$ nodes. As such, our research would be discussing how various factors affect the percentage reconstructed.

Using the Python random module, we are able to construct a function which generates identifiers, append them into a list, and shuffles the items of the list using the random module. Next, identifiers are mapped to each leaf node, and stored in an "answer key" list, to check the program's accuracy. Ranges are randomly generated as well using Python's randint function with replacement, where two integers are randomly generated and both are used for the inputted ranges and subsequent nodes from the cover of the range were used to be inputted in the two algorithms. As such, querying a range will lead to a computed cover, consisting of a set of nodes, and these nodes and their subsequent identifiers are inputted into the algorithm through a Python dictionary, labelled as nodedictionary. Items in the nodedictionary are notated as $n : P$, where n is the node number, and P_n is the set of identifiers covered by n in the form of a list.

In addition, we constructed a program named pair, which calculates the minimum cover based on the randomly generated ranges. Both programs have the same setup.

5 Results

5.1 Overview

In order to obtain our results from the Python programs, we used for–loops to repeat the algorithm with increasing depth number to investigate its effect on the percentage of depth $d-1$ nodes reconstructed, as well as investigate the relationship between number of queries and

percentage reconstructed. Regarding these experiments, since both algorithms are fundamentally similar, for simplicity, all our data from this section were generated from the Recursive Neighbour Search Algorithm. However, when comparing the two algorithms by testing for time taken, both algorithms were experimented upon.

All our graphs were plotted either with Python's matplotlib.pyplot module or with charts in Google Sheets.

5.2 Overall Statistics

Looking at all iterations of both algorithms, the percentage of iterations which resulted in 100% reconstruction of depth $d - 1$ nodes is almost 0, but both our algorithms are successful in inferring more nodes from the queries provided. Detailed statistics will be analysed and evaluated in the next few sections.

5.3 Effect of Depth Number

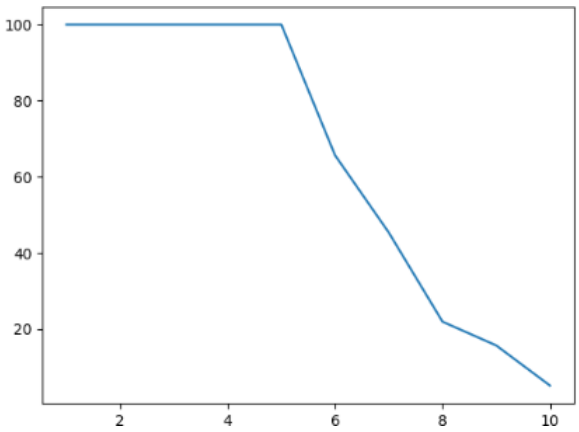

Fig. 4. Graph of percentage reconstructed (%) against depth number at fixed query number

From the graph (Figure 4), the trend is rather distinct, which is that at smaller depths, from 1-5, 100% of depth $d-1$ nodes were reconstructed, and it decreases rather swiftly as depth increases. The reason being is that due to a fixed number of queries of around 250, since 2^d from $d = 1$ to $d = 5$ is under 250, the probability of obtaining unique queries greater than 2^d is higher. However, there is a sharp decrease because the number of nodes in the tree is increasing exponentially, thus the same number of queries is inadequate to provide the same amount of information at increasing d.

5.4 Effect of Number of Queries

Next, we investigate the effect of the number of queries inputted to the algorithm and how it affects the success rate of the algorithm. We decided to plot our results on a scatter chart and it demonstrated the results as seen in Figure 5, and we derived an equation which modelled this trend, which is $y = 100(1-e^{-0.012456x})$. With this equation, it is not difficult to estimate the percentage of $d-1$ nodes reconstructed at a specific number of queries. Observing the scatter plot, it can be observed that as the number of queries inputted to the algorithm increases, the percentage reconstructed increases, but at a decreasing rate when the graph tends to 100%

reconstructed. We chose the values of $d = 8$ and 250 queries to allow a spread of data which showcases how the data points tend to be 100%. Originally, only 50 queries were utilised, which when observed from the graph, appeared to showcase a linear relationship, however, as one might observe from the graph, the graph ends up plateauing at approximately 150 queries. We believe that the reason this might be is due to the manner in which our setup is constructed. Queries are generated randomly *with replacement*, thus this scenario illustrates how with a greater number of queries, the number of novel unique queries inputted into the algorithm will decrease, as most unique queries have been accepted by the algorithm or have been computed thus far.

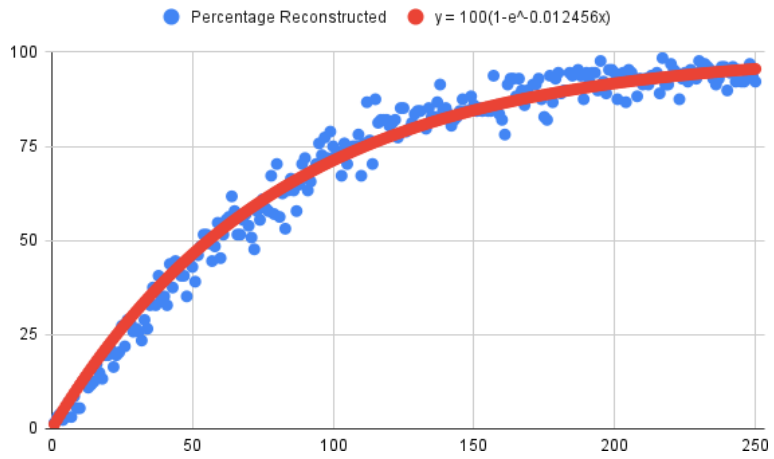

Fig. 5. Graph of percentage reconstructed (%) against number of queries at a fixed depth of d = 8

5.5 Time Comparison

Our first metric for comparing the algorithms is their runtime. In Figure 6, we plot the average time taken for each algorithm as the number of nodes given as input increases. We see that both algorithms grow, but RNS has at a significantly faster rate than the one-pass algorithm.

To explain this, note that RNS' runtime is related to the number of queries since this determines the number of adjacency relations that will be explored. For one-pass, each leaf node is computed from a similar equation whose efficiency does not degrade as much when the number of queries is increased. As such, we conclude that one-pass is more efficient, especially when there are a large number of queries.

5.6 Accuracy Comparison

Next, we compare the accuracy differences between the two algorithms. We define accuracy as our success rate, with that being the percentage reconstructed of depth $d - 1$ nodes. We used a Python loop which increases the number of queries per iteration and computes the percentage reconstructed for each algorithm during that iteration. From Figure 7, it can be observed that the one-pass algorithm generally has higher accuracy than the recursive neighbour search for most iterations, as well as having a greater number of 100% reconstructions. This is unsurprising since RNS is heuristic while One-pass is optimal, and this experiment demonstrates that this suboptimality is significant in practice.

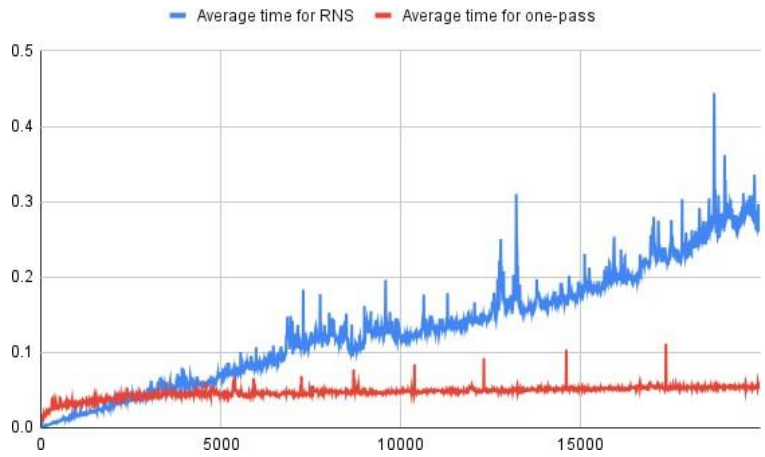

Fig. 6. Graph of time taken per algorithm against number of queries at a fixed depth of d = 8

We note that this graph is similar to Figure 5, but with the results of both algorithms superimposed on the same grid, to accurately showcase the contrasts of accuracy between both algorithms. This also depicts the similarities in functionality and structure between the two algorithms, as both still produced similar trend-line and graph shapes. Thus, it can be observed how both algorithms are fundamentally the same, yet it is important to note that the speed and efficiency of both are not entirely the same.

Next, we discuss the reasons behind this trend. The one-pass algorithm is more accurate due to the fact that it has no limitations in its structure. It computes all nodes, and thus would have no possibility of missing a case or a combination of nodes and would hence utilise all available information to compute the outputs, making it extremely efficient and comprehensive.

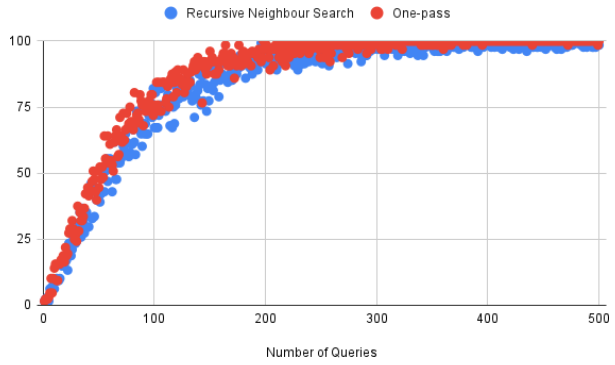

Fig. 7. Graph of percentage reconstructed against number of queries at a fixed depth of d = 8

6 Application to Cryptography

Augmented binary trees are used in Range Searchable Encryption (RSE) schemes to outsource databases to the Cloud for storage and query processing. Suppose we have a doctor (our client) with a list of sensitive information (salary, hospital records etc.) arranged by age and wishes to outsource this data to Amazon Web Services. Suppose he later wants to make queries

to find patients within a range of ages (say 10-24), but does not want to either pass the server the key (because he does not trust it) nor download the whole database. To support such an application, one can use RSE to construct and encrypt an augmented binary tree, representing the files as leaves in the tree. Queries are supported by breaking down the query into a cover, then accessing those nodes in the RSE scheme to retrieve the associated files at the descendant leaves. The client then collects, decrypts and interprets these files as the query response. Some of these schemes in the literature make use of augmented binary trees to achieve this [1, 2].

In the process of using such a scheme, an adversary can observe the nodes queried and the encrypted files that are returned. In the application above, the adversary may be an eavesdropper on the network, or a nosy employee of Amazon. Since the files are encrypted, they cannot directly learn about the information in the file. However, by observing the nodes queried and the patterns of encrypted files, they can apply our algorithms to deduce the range value associated to each file, thereby "reconstructing" the binary tree. This is a form of a Leakage Abuse Attack, an wide area of research in the literature.

First, he observes the queries, which as mentioned before are sets of nodes, then splits them into individual nodes and notes down the files associated with each node. This is where the adversary implements our algorithms, where input is $[(n_1), (P_1), ..., (n_q), (P_q)]$ and output is $[l_n, id_n]_{n \in N}$, where N is the maximum subset of leaf nodes and identifiers that can be deduced from the input and $N \leq 2^d, ..., 2^{d+1} - 1$. This output essentially gives sensitive information about which file corresponds to which range value.

7 Conclusion

In conclusion, we have defined the new Augmented Binary Tree Reconstruction problem, and designed two algorithms to address it. We mathematically and experimentally evaluate these algorithms, showing that the less-intuitive one-pass algorithm is optimal and efficient while the more-straightforward RNS is only a heuristic and suffers from efficiency blowup.

We then demonstrate a practical and impactful application of our algorithms in the security evaluation of RSE schemes deployed on untrusted Cloud servers. Future work could explore testing our algorithms on actual cryptography schemes, in order to conduct security evaluations of these schemes, or alternatively, future work could also explore alternate or optimized algorithms to solve this problem of reconstruction.

8 Acknowledgements

We would like to thank our mentors, Dr Ruth Ng for her advice and guidance on our project, and Mr Choo Jia Guang, for all of his assistance during the programming of our algorithms.

References

1. Ioannis Demertzis, Stavros Papadopoulos, Odysseas Papapetrou, Antonios Deligiannakis, and Minos Garofalakis. Practical private range search revisited. In *Proceedings of the 2016 International Conference on Management of Data*, pages 185–198, 2016.
2. Sky Faber, Stanislaw Jarecki, Hugo Krawczyk, Quan Nguyen, Marcel Rosu, and Michael Steiner. Rich queries on encrypted data: Beyond exact matches. Cryptology ePrint Archive, Report 2015/927, 2015. `https://eprint.iacr.org/2015/927`.

Appendix

A Pseudocode

A.1 RNS and One-pass

Below is our pseudocode for both algorithms, with RNS on the left, and the pseudocode for the one-pass algorithm on the right.

Alg RNS($\{n_1, G_1\}...\{n_x, G_x\}$)

Initialise dictionary **T**
For i = 1...x: set **T**[n_i] = G_i
Set bool = True
While (bool):
  Set bool = False
  For i = 1, 1.5, 2, 2.5 … 2^d where **T**[i] is defined:
    If **T**[i+0.5] is defined:
      Set **T**[2i+1] = **T**[i] ∩ **T**[i+0.5]
      Set **T**[2i] = **T**[i] \ **T**[2i+1]
      Set **T**[2i+2] = **T**[i+0.5] \ **T**[2i+1]

    If **T**[i/2] is defined:
      Set **T**[i+1] = **T**[i/2] \ **T**[i]

    If **T**[i//2] is defined:
      Set **T**[i+1] = **T**[i//2] \ **T**[i]

    If i is even and **T**[i+1] is defined:
      Set **T**[i/2] = **T**[i] ∪ **T**[i+1]
  If **T** has changed in size, set bool = True

For i = 2^d ... $2^{d+1} - 1$, if **T**[i] is defined, append to list extracted_files

Output length of extracted_files / 2^d multiplied by 100 to output percentage reconstructed

Alg One – pass($\{n_1, G_1\}...\{n_x, G_x\}$)

Initialise dictionary **T**
For i = 1...x: set **T**[n_i] = G_i
Initialise dictionary **L**

For y in **T**:
  Compute K_y, where K is the set of descendent leaf nodes of node y.
  For c in K_y:
    Set **L**[c] += y
Let the set of nodes not in **L**[x], where x is a node, be N_x
For i = 2^d ... $2^{d+1} - 1$:

  Set **T**[i] to as follows

$$\bigcap_{n \subseteq \mathbf{L}[i]} \mathbf{T}[n] \setminus \bigcup_{n \subseteq N_i} \mathbf{T}[n]$$

  Append **T**[i] to list extracted_files
Output extracted_files

Fig. 8. Pseudocode for both Algorithms