

Secure Range-Searching Using Copy-And-Recurse

Eyal Kushnir*, Guy Moshkovich†, Hayim Shaul‡

IBM Research

June 23, 2023

Abstract

Range searching is the problem of preprocessing a set of points P , such that given a query range γ we can efficiently compute some function $f(P \cap \gamma)$. For example, in a 1 dimensional *range counting* query, P is a set of numbers, γ is a segment and we need to count how many numbers in P are in γ . In higher dimensions, P is a set of d dimensional points and the query range is some volume in \mathbb{R}^d . In general, we want to compute more than just counting, for example, the average of $P \cap \gamma$. Range searching has applications in databases where some SELECT queries can be translated to range searches. It had received a lot of attention in computational geometry where a data structure called *partition tree* was shown to solve range searching in time sub-linear in $|P|$ using only space linear in $|P|$.

In this paper we consider partition trees in a secure setting where we answer range queries without learning the value of the points or the parameters of the range. We show how partition trees can be securely traversed with $O(n^{1+\epsilon} + t \cdot n^{1-\frac{1}{d}+\epsilon})$ operations, where $n = |P|$, t is the number of operations needed to compare to γ and $\epsilon > 0$ is a parameter. As far as we know, this is the first non-trivial bound on range searching and it improves over the naïve solution that needs $O(t \cdot n)$ operations.

Our algorithms are independent of the encryption scheme but as an example we implemented them using the CKKS FHE scheme. Our experiments show that for databases of sizes 2^{23} and 2^{25} , our algorithms run $\times 2.8$ and $\times 4.7$ (respectively) faster than the naïve algorithm.

The improvement of our algorithm comes from a method we call copy-and-recurse. With it we efficiently traverse a r -ary tree (where each inner node has r children) that also has the property that at most ξ of them need to be recursed into when traversing the tree. We believe this method is interesting in its own and can be used to improve traversals in other tree-like structures.

*eyal.kushnir@ibm.com

†guy.moshkovich@ibm.com

‡hayim.shaul@ibm.com

1 Introduction

The problem of *range searching* has been studied extensively. In this problem we are given a finite set of points $P \subset \mathbb{R}^d$ and a volume (range) $\gamma \subset \mathbb{R}^d$ and wish to quickly compute some function of $P \cap \gamma$. For example, $P \subset \mathbb{R}^2$ can represent a 2-column database, where $p \in P$ is a record and the columns are mapped to the x and y coordinates. Then, for example, this query:

```
SELECT COUNT(*) FROM table tab
WHERE (x1 < tab.x) and (tab.x < x2)
and (y1 < tab.y) and (tab.y < y2);
```

can be answered by counting the number of points in P that fall in the axis-parallel box $\gamma = \{(x, y) \mid x_1 < x < x_2 \text{ and } y_1 < y < y_2\}$.

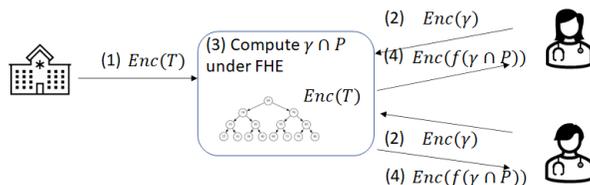


Figure 1: A FHE-based system for delegating a database to an untrusted cloud. (1) The data owner (hospital) uploads a database to an untrusted cloud. (2) A client (doctor) encrypts their query and send it to the untrusted cloud. (3) the cloud performs the query under FHE and (4) returns the result to the client.

The privacy preserving range searching problem is to compute a function over $P \cap \gamma$ without sharing the input (i.e. P or γ). This is useful if for example P is a database of medical data that is kept at an untrusted cloud. Such problems can be solved by secure-multi party computation (MPC) in which several parties compute an output without sharing their private input. Solutions such as that by Beaver [7] have the disadvantage that they have a large communication complexity. Also they are *interactive* which requires the parties to remain online for the entire computation. There are *non-interactive* methods, e.g. based on Fully Homomorphic Encryption (FHE) which is a public key encryption where addition and multiplication can be applied on ciphertexts. Using FHE we can send an encrypted input to a server and run entirely at the server side. The methods and techniques in this paper are generic and can be implemented in any MPC solution, however, for simplicity we present them for FHE.

The naïve implementation checks for every point $p \in P$ whether $p \in \gamma$ for a total of $O(t \cdot n)$ operations, where $n = |P|$ and t is the number of operations to check whether $p \in \gamma$. In plaintext, efficient solutions group points together, check whether the entire group is contained in γ and recursively continue only in groups that are partially contained in γ . These solutions rely heavily on branching in the code. This is something that is impossible under FHE (or MPC in general). Under FHE it is impossible to branch based on a condition

that depends on the input. More specifically, knowing whether a condition on the input is met contradicts the semantic security of FHE and therefore is impossible. Naïvely, when running a privacy-preserving protocol a branch based on a condition C is replaced by: (1) computing a ciphertext $Enc(c)$ where $c = 1$ if C is met and $c = 0$ otherwise; (2) computing both branches; (3) multiplexing by multiplying one branch by $Enc(c)$ and the other branch by $(1 - Enc(c))$. Effectively, this leads to computing all possible branches of an algorithm which is why solutions that work well in plaintext do not (naïvely) extend well to FHE.

In this paper we consider the secure range searching problem for ranges that can be described with a constant number of parameters (formally this is called “constant description complexity”. See definition in Section 7.1). In this case we show how to solve the privacy-preserving range searching problem in $O(n^{1+\epsilon} + t \cdot n^{1-\frac{1}{d}+\epsilon})$ time, where t and n are as before, d is the dimension $P \subset \mathbb{R}^d$ and $\epsilon > 0$ is arbitrarily small (as usual there is a multiplicative factor that grows as ϵ gets smaller). Our algorithm outperforms the naïve algorithm asymptotically and also in practice as our experiments show. We note that under FHE the problem has a lower bound of $\Omega(n)$ operations. This stems from a $\Omega(n)$ lower bound on private information retrieval (PIR) ([8]) which can be solved by 2D range-searching problem. In addition, in plaintext the best known solution for range searching with near linear storage is $O(n^{1-\frac{1}{d}+\epsilon})$ ([2]).

In plaintext, range-searching can be solved using linear storage with partition trees (see [20]). This tree-structure is built for a family of ranges Γ and a set of points P . In the construction we partition $P \subset \mathbb{R}^d$ into m subsets P_1, \dots, P_m such that for any range $\gamma \in \Gamma$ it can be quickly determined whether a subset P_i is contained ($P_i \subset \gamma$), disjoint ($P_i \cap \gamma = \emptyset$) or crosses γ . The geometric structure admits a partition where only a small number of subsets cross γ and need further “attention”. In plaintext, this leads to an improved running time of $O(t \cdot n^{1-\frac{1}{d}+\epsilon})$.

Naïvely implementing a partition tree under FHE leads to $O(t \cdot n)$ operations because as mentioned earlier, the naïve approach visits all branches and nodes of the tree. This leads to an algorithm whose running time is $O(t \cdot n)$. Our improved time bound $O(n^{1+\epsilon} + t \cdot n^{1-\frac{1}{d}+\epsilon})$ comes from using a method we call “copy-and-recurse” with it we traverse a partition tree efficiently under FHE. The copy-and-recurse method can be applied when traversing a r -ary tree (i.e. each inner node has r children) that also has a bound $\xi < r$ on the number of children that need to be recursed into. The method copies ξ children and their subtrees (under FHE) and recurses into the copied subtrees. The time spent by algorithm copying subtrees is linear in the tree size, however, the function that compares a range is called only at a sublinear number of nodes. This leads to an algorithm that needs only $O(n^{1+\epsilon} + t \cdot n^{1-\frac{1}{d}+\epsilon})$ operations. The choice of r and the bound ξ determine the value of ϵ . We demonstrate our copy-and-recurse method on partition trees that solve the range searching problem, but our method can be used in other tree based solutions and we expect it to be of interest for other solutions as well.

To demonstrate the performance of our algorithm we provide a C++ code

that implements it. We use the HElayers library [3] and our experiments show that our method is $\times 4.6$ faster than the naïve implementation for a database of 32M records. (We note that modern databases often have significantly more records.) We expect our algorithm to have impact in practice as well as in theory.

In Figure 1 we show a system that implements privacy preserving range searching. At the left is a hospital. To save IT and maintenance costs they upload their database to an untrusted (honest but curious) cloud. Physicians of that hospital can then query that database to get various statistics and views of the database by sending an encrypted query range to the cloud, receiving the encrypted result and decrypting it.

1.1 Our contribution

We list below our contributions in this paper.

- **The copy-and-recurse method.** We show how to efficiently traverse a full r -ary tree (i.e. each inner node has r children) with n leaves, where there is a bound $\xi < r$ on the number of children we need to recurse into at each node. Here r is a parameter $0 < r < n$ that affects the running time. In a nutshell the method traverses a tree as we now (briefly) describe. When visiting a node:
 - Compute (under FHE) an indicator vector indicating which children need to be recursed into. (There are at most ξ such children.)
 - Use the indicator vector to copy ξ children and their subtrees.
 - Recurse into the copies of ξ children.
- **An algorithm to answer privacy preserving range searching queries.** We show how to build a FHE-friendly partition tree that efficiently answers range searching queries (see [1, 2, 20]). In a nutshell, a partition tree is built with a parameter $r > 0$, a family of ranges Γ and points $P \subset \mathbb{R}^d$ where each inner node has $O(r)$ children. When traversing the tree with a query $\gamma \in \Gamma$ at most $\xi = O(r^{1-\frac{1}{d}})$ children need to be recursed into at each node. Our algorithm can be implemented by an arithmetic circuit of size $O(n^{1+\epsilon} + t \cdot n^{1-\frac{1}{d}+\epsilon})$, where t is the size of an arithmetic circuit that checks whether γ contains or intersects a point or a simplex and $\epsilon > 0$ is a parameter that can be arbitrarily small (as usual, there is a multiplicative factor that grows when ϵ decreases).
- **Generalized range searching problem definition.** We generalize the classic range searching problems (counting and reporting) to output $f(P \cap \gamma)$ for a large set of functions. Specifically, our protocol works with any function f , that can be computed in a divide and conquer way, i.e., there exists another function g such that $f(A \cup B) = g(f(A), f(B))$. This is summarized in the following theorem:

Theorem 1. *Let $P \subset \mathbb{R}^d$ be a set of n points, $\Gamma \subset 2^{\mathbb{R}^d}$ a family of semi-algebraic ranges, T a full partition tree as output from Algorithm 4, a function f that can be computed in a divide and conquer manner and t and ℓ are the size and depth of the circuit that compares a range to a simplex, then given $\gamma \in \Gamma$, $PPRangeSearch$ (Algorithm 2) securely evaluates $f(\gamma \cap P)$ in a circuit whose size is $O(n^{1+\epsilon} + t \cdot n^{1-\frac{1}{d}+\epsilon})$ and depth is $O(\ell \cdot \log n)$.*

- **Implementation and experiments for privacy preserving range searching.** We implemented our algorithm into a system that answers privacy preserving range searching queries. Our algorithm is generic and can be implemented with any scheme. In this paper we used the HElayers framework [3] with HEAAN [15] as the cryptographic library.

1.2 Paper organization

The rest of the paper is organized as follows: in Section 2 we review the related work and in Section 3 we give some preliminaries and notations (the ones that relate to computational geometry we defer to Section 7.1). In Section 4 we state the main problem this paper solves. In Section 5 we describe our copy-and-recurse method. In Section 6 we describe a FHE-friendly partition tree to solve the 1-dim privacy preserving range searching counting problem. In Section 7 we show how to extend our solution to d dimensions. In Section 8 we analyze our algorithm, with respect to the size and depth of the arithmetic circuit that implements it and its security. In Section 9 we show how to extend our solution from counting to more generic functions. In Section 10 we describe the experiments we have done and finally we conclude in Section 11.

2 Related Work

Range searching. In a range searching problem (in plaintext) we are given a set of n points $P \subset \mathbb{R}^d$ and a family of ranges Γ (usually of infinite size) and wish to efficiently compute $P \cap \gamma$ (or some function on it) for any $\gamma \in \Gamma$. The problem has been studied extensively in computational geometry. In a seminal work Matoušek [20] showed how to build a data structure called *partition tree* of $O(n)$ size where for any γ , which is a halfspace in \mathbb{R}^d bounded by a hyperplane (i.e., Γ is the infinite set of all such halfspaces), $P \cap \gamma$ can be computed in $O(n^{1-1/d+\epsilon})$ time. Later, Agarwal and Matoušek [1] extended this result to ranges of constant description complexity in time $O(n^{1-1/c+\epsilon})$, where $c = d$, for $d = 2, 3$ and $c = 2d - 4$, for $d \geq 4$. Using tools from algebraic geometry Agarwal, Matoušek and Sharir [2] improved this running time to $O(n^{1-1/d+\epsilon})$.

In the privacy preserving context, the private information retrieval (PIR) problem can be answered with range searching as we describe now. In PIR, one party (the server) has an array T and a second party (the client) has an index $x \in \mathbb{Z}$. The goal is to output $T[x]$ to the client while hiding x from

the server and $T[i]$, for $i \neq x$ from the client. To answer PIR with range searching replace each entry in the table, $T[i]$, with a 2D point $(i, T[i])$, for $i = 1, 2, \dots$. To answer the PIR problem, report the (single) point in the range $\gamma_x = \{(a, b) \in \mathbb{R}^2 \mid x - 0.5 < a < x + 0.5\}$. In [8] a lower bound of $\Omega(n)$ was shown for the PIR problem, which holds for the privacy preserving range searching query as well. Another line of works related to private range query is based on symmetric searchable encryption (SSE) e.g., [16, 17, 18]. SSE schemes offer a tradeoff between efficiency and revealing some well defined information about queries and stored data and are less secure than homomorphic encryption schemes.

As far as we know, this is the first non-trivial work that answers general range searching queries under FHE.

Traversing a tree under FHE One of our contributions is the copy-and-recurse method to efficiently traverse a tree under FHE. A recent work by Azogagh et al. [6] describes how to efficiently traverse a decision tree using FHE. In their implementation they evaluate a single path instead of the entire tree. At each node of the decision tree they compare a variable to a parameter, where they blindly copy the variable and the parameter from an array of variables and parameters. Here, the index of the node that the traversing reached is encrypted and used to fetch the right variable and parameter from the arrays. However, it is not clear how to extend their ideas to answering range searching queries. Another recent work by Cong et al. [14] discussed *homomorphic traversing* where they show how to efficiently traverse a decision tree. To use their techniques the values at each decision node needs to be bit-wise encrypted i.e., each bit in its own ciphertext. The threshold in the tree need to be given in plaintext. Then they express the conditions at decision node as boolean polynomials and compute all polynomials together while applying a heuristic that finds mutual subpolynomials and computing them once. In the worst case, their technique leads to $O(n)$ conditions being evaluated. Their technique relies heavily on the input being encrypted in binary and also the thresholds being given in plaintext. Furthermore, it is not clear how to extend their techniques to answer range searching. In [5] Akavia et al. showed how to train a decision tree and use it for prediction, however their method evaluates the conditions at all nodes of the tree leading to $O(t \cdot n)$ operations, where t is the number of operations to compute a condition at a node. In [21] the authors also considered prediction using decision trees but here as well they tested the conditions at each node of the tree again leading to $O(t \cdot n)$ operations. In [22] the authors considered a solution that uses garbled circuits and ORAM to achieve sub-linear prediction time with a decision tree, however their solution is interactive. We summarize these works in Table 1.

Computing a function over database records that match a query function. Given a set $P \subset \mathbb{Z}^m$ of n points and a function $h : \mathbb{Z}^m \rightarrow \mathbb{Z}$, Iliashenko et al. showed in [19] how to preprocess P s.t. given an encrypted value x ,

	[6]	[5]	[14]	copy-and-recurse
Encoding agnostic	✓	✓		✓
Sublinear	✓		✗	✓
Encrypted	✓	✓		✓
Range searching				✓

Table 1: Comparing our copy-and-recurse method to other works. *Encoding agnostic* means input is not restricted to be encrypted bit-wise. *Sublinear* means the number of comparisons performed is sublinear in the number of leaves (a crossed checkmark means a heuristic that may result in a sublinear number of comparisons). *Encrypted* means encrypted trees are also supported. *Range searching* means range searching is efficiently supported.

it can efficiently compute the number of points whose image is x , i.e. report $|\{p \in P \mid h(p) = x\}|$. This can be formulated as a range searching query where a range is of the form $\gamma_x = \{p \in \mathbb{Z}^m \mid h(p) = x\}$ that doesn't necessarily have a finite VC-dimension and then reporting $|P \cap \gamma_x|$, i.e. report $f(P \cap \gamma)$, where $f(A) = |A|$. However, they require that γ_x be given in plaintext. Moreover, they do not show how to compute $f(P \cap \gamma)$ for functions f or ranges γ_x other than what mentioned above.

Cheon et.al [13] considered encrypted queries over an encrypted database. They propose a search-and-compute method, where they first search for records that match a query and then compute a function f (addition, average, min, max, etc.) on those records. However, to find the records that match a query they apply a *IsMatch* function all of the records, for a total of $O(t \cdot n)$ operations, where n is the number of records and t is the number of operations to evaluate *IsMatch*.

3 Preliminaries

To improve readability we split the preliminaries into two parts. We give here the preliminaries that are needed to understand the 1-dim case and in Section 7 we give more preliminaries in computational geometry that are needed for the d -dim case.

Number representation. Our algorithms work over the reals and are stated in this paper as such. Computers use finite space to represent numbers and therefore cannot truly work over the reals. There are several number representations (e.g. fixed point representation) to address this problem. In this paper we are not concerned with how numbers are represented and require only the existence of addition and multiplication operations.

Divide and conquer functions. In this paper we are concerned with functions that take a set as input. We say such a function, f can be computed in a divide and conquer manner if $f(A \cup B)$ can be computed from $f(A)$ and $f(B)$, when A and B are (non-empty) disjoint sets. For example, for the cardinality function we have $|A \cup B| = f(A \cup B) = f(A) + f(B) = |A| + |B|$. More generally,

there exists a function g such that $f(A \cup B) = g(f(A), f(B))$.

Selection matrix. A selection matrix $M \in \{0, 1\}^{\xi \times d}$, with $\xi < d$ is a matrix that has the property that it *selects* elements of a vector. More formally: $M \cdot x^T = (x_{i_1}, \dots, x_{i_\xi})$, where $x = (x_1, \dots, x_d)$. Intuitively, to construct M we set $M_{r,c} = 1$ if $i_r = c$, i.e. if the c -th element in x is the r -th element in $M \cdot x^T$, otherwise we set $M_{r,c} = 0$.

Trees. We use v to denote a node in the tree. We use dot (".") to denote members of v , so for example, $v.child[1], \dots, v.child[r]$ are the children of v . We also denote the root of the tree by *root*. The height of a node v is the number of nodes on the path from v to the root and the height of a tree is the maximal height of its nodes.

3.1 Fully homomorphic encryption

Our algorithms and protocols can be implemented using FHE or other MPC schemes that supports addition and multiplications. However, to improve readability we describe our protocol using FHE.

FHE (see e.g. [10, 11]) is an asymmetric encryption scheme that also supports $+$ and \times operations on ciphertexts. More specifically, a FHE scheme is the tuple $\mathcal{E} = (Gen, Enc, Dec, Add, Mult)$, where:

- $Gen(1^\lambda, p)$ gets a security parameter λ and an integer p and generates the keys pk and sk .
- $Enc_{pk}(m)$ gets a message m and outputs a ciphertext $\llbracket m \rrbracket$.
- $Dec_{sk}(\llbracket m \rrbracket)$ gets a ciphertext $\llbracket m \rrbracket$ and outputs a message m' .
- $Add_{pk}(\llbracket a \rrbracket, \llbracket b \rrbracket)$ gets two ciphertexts $\llbracket a \rrbracket, \llbracket b \rrbracket$ and outputs a ciphertext $\llbracket c \rrbracket$.
- $Mult_{pk}(\llbracket a \rrbracket, \llbracket b \rrbracket)$ gets two ciphertexts $\llbracket a \rrbracket, \llbracket b \rrbracket$ and outputs a ciphertext d .

Correctness is the requirement that $m = m'$, $c = a + b \pmod p$ and $d = a \cdot b \pmod p$. In an approximated FHE (e.g. CKKS [11]) we require that $m \approx m'$, $c \approx a + b$ and $d \approx a \cdot b$.

Semantic security is the requirement that given $pk, \llbracket m_1 \rrbracket, \dots, \llbracket m_{poly(\lambda)} \rrbracket, m_1, \dots, m_{poly(\lambda)}$, where $poly(\lambda)$ is a number that polynomially depends on λ , then given $\llbracket m_0 \rrbracket$, the value of m_0 is known in probability negligible in λ .

Using additions and multiplications we can construct any arithmetic circuit and compute any polynomial $\mathbb{P}(x_1, \dots)$ on the ciphertexts $\llbracket x_1 \rrbracket, \dots$. For example, in a client-server system, the client encrypts her data and sends it to the server who computes a polynomial \mathbb{P} on the encrypted input. The output is also encrypted and is returned to the client who then decrypts it. The semantic security of FHE guarantees the server does not learn anything on the content of the client's data.

When evaluating an arithmetic circuit, we are concerned with the size of C ($size(C)$), which is the number of gates in C and with the depth of C ($depth(C)$), which is the maximal number of multiplication gates on a path of C . The time

to evaluate a circuit is then $Time = overhead \cdot size(C)$, where in many schemes $overhead$ grows when $depth(C)$ increases.

3.2 Notation

Abbreviated syntax. To make our algorithms and protocols more intuitive to read we use $\llbracket \cdot \rrbracket_{pk}$ to denote a ciphertext. When pk is clear from the context we omit it. We use an abbreviated syntax:

- $\llbracket a \rrbracket + \llbracket b \rrbracket$ is short for $Add_{pk}(\llbracket a \rrbracket, \llbracket b \rrbracket)$.
- $\llbracket a \rrbracket \cdot \llbracket b \rrbracket$ is short for $Mult_{pk}(\llbracket a \rrbracket, \llbracket b \rrbracket)$.
- $\llbracket a \rrbracket + b$ is short for $Add_{pk}(\llbracket a \rrbracket, Enc_{pk}(b))$.
- $\llbracket a \rrbracket \cdot b$ is short for $Mult_{pk}(\llbracket a \rrbracket, Enc_{pk}(b))$.

3.3 Comparisons under FHE

A major obstacle when running under FHE is not being able to perform comparisons, i.e. to get a **plaintext** bit indicating whether one encrypted message is smaller than another. Informally, this is impossible because it breaks the semantic security of FHE. Such a comparison could have been useful, for example, when traversing a binary search tree under FHE to continue into only one child of a node (and not both).

Instead, there has been work (e.g. [12]) of implementing a $IsSmaller(\llbracket c_1 \rrbracket, \llbracket c_2 \rrbracket)$ function that returns a **ciphertext** that encrypts the indicating bit (whether or not $c_1 < c_2$). In a nutshell, such works implement a bi-variate polynomial

$$IsSmaller(x, y) = \begin{cases} 1, & \text{if } x < y \\ 0, & \text{otherwise.} \end{cases} \quad (\text{Or its approximated version in the case of}$$

CKKS). The implementation details of this polynomial depends on the underlying FHE scheme and the way numbers are represented. With this primitive, it is easy to construct more complicated tests, e.g., whether a value x is contained inside a range (a, b) .

3.3.1 Making Tests Under FHE - the 1-dim case

While our protocol works also in high dimensions, we start by explaining the easier 1-dim case. The extension to higher dimensions is explained in Section 7. We assume the existence of 2 functions $IsContaining$ and $IsCrossing$ (described in detail below). Our protocol uses these functions as black-boxes. We are not concerned with how these functions are implemented and express the complexity bounds of our protocol with respect to $IsContaining$ and $IsCrossing$ (see below the definition of t and ℓ , parameters that capture the “hardness” these functions).

For the 1-dim case, these functions are (see also Figure 2):

- $IsContaining(\llbracket\sigma\rrbracket, \llbracket\gamma\rrbracket)$. This function gets as input two encrypted segments $\sigma, \gamma \subset \mathbb{R}$. The value of $IsContaining$ is a ciphertext $\llbracket c \rrbracket$, where $c = 1$ if $\sigma \subseteq \gamma$ and $c = 0$ otherwise.
- $IsCrossing(\llbracket\sigma\rrbracket, \llbracket\gamma\rrbracket)$. This function gets as input two encrypted segments $\sigma, \gamma \subset \mathbb{R}$. The value of $IsCrossing$ is a ciphertext $\llbracket c \rrbracket$, where $c = 1$ if γ crosses σ (i.e. intersects but not contains) and $c = 0$ otherwise.

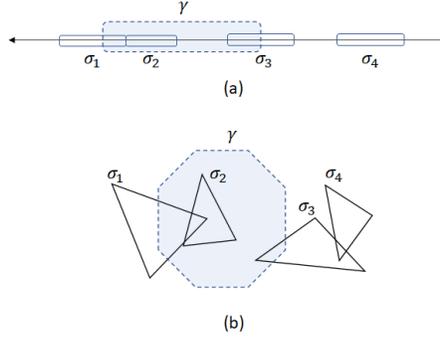


Figure 2: A range γ and simplices $\sigma_1, \sigma_2, \sigma_3$ and σ_4 . Above (a) is a setting in 1-dim, where the range is a segment and the simplices are non-intersecting segments. Below (b) is a setting in 2-dim, where the range is an octagon and the simplices are (possibly intersecting) triangles. In both cases σ_1 and σ_3 cross γ , σ_2 is contained in γ and σ_4 is disjoint from γ .

3.4 Size and depth of computing $IsContaining$ and $IsCrossing$ (t and ℓ)

We define t and ℓ as the size and depth of the circuit that computes $IsContaining$ and $IsCrossing$. These parameters capture the “hardness” of comparing a range to a simplex in FHE.

We note that the implementation of $IsCrossing$ and $IsContaining$ varies with the FHE scheme and the way numbers are represented (as does the primitive $IsSmaller$ they depend on). Looking ahead, in Section 7 we consider the problem in high dimensional settings. There, the implementation of these functions also depends on the shape of the query ranges. It therefore makes sense to show how the complexity of a range search protocol depends on the complexity bounds of $IsContaining$ and $IsCrossing$. Also, from the practical perspective most of the running time is spent executing these functions. Therefore it makes sense for the complexity to be expressed as a function of t and ℓ .

4 Problem Statement

In this paper we gradually introduce the ultimate problem we consider. We start by a simple problem of counting in 1-dim. Then we extend the problem to higher dimensions and finally, we extend the problem to more general functions.

Simple problem: Counting in 1-dim. First we consider the counting problem in 1-dim. In this problem we are given a set $P \subset \mathbb{R}$ of n numbers. We preprocess P in linear time into a data structure D of linear size, such that given a query segment $\gamma = [\gamma_a, \gamma_b]$ we efficiently compute $|\gamma \cap P| = |\{p_i \mid \gamma_a \leq p_i \leq \gamma_b\}|$.

Extension to higher dimensions. We then extend this problem to higher dimensions. That is, $P \subset \mathbb{R}^d$ and the query range γ is a volume that can be described with a constant number of parameters. Here γ is taken from a family of ranges Γ . The family of ranges Γ is known at the time D is constructed. See for example Figure 2 (bottom) for an example in 2-dim.

Extension to more general functions. Finally, we extend our solution to compute efficiently $f(P \cap \gamma)$, where f is a function that can be computed in a divide and conquer manner.

In all cases our privacy guaranty is that D does not leak anything on P except n and d . Also, during the computation of the query nothing leaks on P or on γ .

4.1 Security model

For simplicity, we consider here a model with 3 players: (1) data owner that owns P , (2) query owner that performs range search queries and (3) the cloud to which the database was uploaded to and performs the computation. We also assume the data owner and the query owner collude. This security model is motivated by the growing trend of outsourcing a database to the cloud (to save maintenance and other IT costs). In the example of Figure 1 the hospital is the data owner and a doctor is the query owner. We note that our protocols can be modified to support other models which we omit from this paper (e.g., where the data owner acts as a server, the query owner as a client and they do not collude).

Figure 1 shows an overall view of a system implementing our secure range searching protocol.

We consider computationally-bounded, semi-honest adversaries. We assume the data owner and the query owner collude but not with the cloud. This follows from our motivation of out-sourcing a database to the cloud. We note that our protocol can be modified to support other models but we use this simple model to present our method. The semantic security of FHE guarantees that the cloud learns nothing on the content of γ or P .

Protocol 1: RangeSearchingProtocol

Parties: Data owner, Query owner, Cloud.

Parameters: $d > 0$ the dimension of the space;

$\Gamma \subset 2^{\mathbb{R}^d}$ a family of ranges.

Data Owner Input: A set $P \subset \mathbb{R}^d$ of n points.

Query Owner Input: A pair (sk, pk) of secret and public keys.

Ranges $\gamma_1, \gamma_2, \dots \in \Gamma$.

The Cloud has no input.

Query Owner Output: $|P \cap \gamma_i|$, for $i = 1, 2, \dots$

The cloud and the data owner have no output.

```
1 Query owner Performs:
2   Send  $pk$  to the Cloud and to Data owner.
3 Data owner Performs:
4   Choose a parameter  $0 < r < n$ .
5    $(T', \xi, h) :=$  Build a partition tree for  $P$  and  $\Gamma$  with parameter  $r$ .
   // See Section 6
6    $T := \text{FillTree}(n, r, h, T')$ . // See Algorithm 4
7    $\llbracket T \rrbracket :=$  encrypt  $v.f, v.\sigma$  for every  $v \in T$ .
8   Send  $\llbracket T \rrbracket, n, \Gamma, \xi, r, h$  to Cloud.
9 foreach range  $\gamma$  the query owner has do
10  | Query-owner performs:
11  |    $\llbracket \gamma \rrbracket := \text{Enc}_{pk}(\gamma)$ 
12  |   Send  $\llbracket \gamma \rrbracket$  to Cloud.
13  | Cloud performs:
14  |    $\llbracket x \rrbracket := \text{PPRangeSearch}_{n,d,\Gamma,\xi,r,h}(T, \llbracket \gamma \rrbracket)$  // See Algorithm 2
15  |   Send  $\llbracket x \rrbracket$  to Data-owner.
16  | Query owner performs:
17  |    $x := \text{Dec}_{sk}(\llbracket x \rrbracket)$ 
18  | Output  $x$ 
```

5 Copy-and-Recurse

Before describing our solution to the range search problem we need to describe our copy-and-recurse method and how it is used to traverse a tree efficiently under FHE.

5.1 Prerequisites

Given a r -ary tree T (where each node has r children) that needs to be traversed, we can use copy-and-recurse if the following holds:

- T is full - i.e., all inner nodes have r children and all leaves have the same height.

- There exists an upper bound $\xi < r$ such that at each node at most ξ children need to be recursed into.

An example of such a tree is a full binary search tree, where at each node we recurse into exactly one child. Here $r = 2$ and $\xi = 1$. In Section 6 and Section 7 we describe partition trees and how they are efficiently traversed using copy-and-recurse. In Section 7.3 we discuss how to transform an arbitrary tree T to a full tree by adding “empty” nodes.

5.2 How it works

We now show how to efficiently traverse a tree T with the prerequisites in Section 5.1. We start at the root of T and perform:

1. Determine (under FHE) which children need to be recursed into. Specifically, we compute a binary vector of r indicator bits $\chi = (\chi_1, \dots, \chi_r)$, where $\chi_i = 1$ iff we need to recurse into the i -th child. The specifics of computing χ depends on the application. Looking ahead, when answering a range search we recurse into children that *cross* the query range (more details are given in Section 6).
2. Copy (under FHE) the children (and their subtrees) we need to recurse into. From the prerequisites mentioned in Section 5.1 only $\xi < r$ are copied since at most ξ children need to be recursed into. In the rest of the paper we think of copying the selected subset as multiplying by a $\xi \times r$ selection matrix. E.g., multiplying the vector of r children by M to get the ξ children we need to recurse into.
3. Recurse. After copying $\xi < r$ children we recurse into the copied subtrees by going back to Step 1 with the root of each subtree.

These steps are executed until a leaf is reached. See Figure 3 for an example of using copy-and-recurse on a binary search tree with n leaves. Applying copy-and-recurse at each node we determine which subtree to recurse into, copy that subtree and continue in recursion until we reach a leaf. Eventually, the decision function was called only $(\log n)$ times.

Discussion The complexity analysis of copy-and-recurse is given in Section 8 as part of the analysis of answering a range search. Intuitively, the total cost of copying subtrees in the process is linear because the size of the subtrees diminishes exponentially. Consider an algorithm that in plaintext traverses a tree and performs some additional work in each node it visits. It can be migrated to run under FHE using copy-and-recurse, such that the additional work will be performed on the same number of nodes as the plaintext algorithm. The extra cost of running under FHE with copy-and-recurse is only the cost of copying.

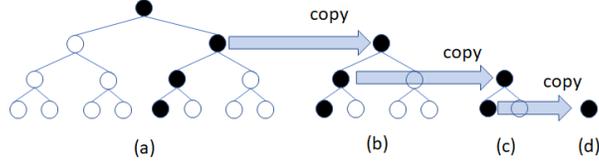


Figure 3: An example of copy-and-recurse method used with a full binary tree with n leaves. The black nodes are the nodes the plaintext algorithm visits as it traverses the tree. First (a) the FHE algorithm visits the root of the tree. It then determines (under FHE) it needs to recurse into the right child and copies the right subtree. Next (b) the algorithm continues in the copied subtree. Again it determines the nodes it needs to recurse into the left child and copies the left subtree. Similarly that happens at (c). In (d) the FHE algorithm reaches a leaf and reports it. Using copy-and-recurse the decision function was called $\log n$ times, as oppose to the naïve FHE algorithm that calls the decision function on all leaves.

6 1-dim Partition Trees

We now continue to describe how range searching can be answered efficiently. To simplify the description we consider here the 1-dim counting problem, i.e., to preprocess a set P of n numbers, $p_1 \leq \dots \leq p_n \in \mathbb{R}$, such that given a range $[a, b]$, we report the number of elements in the range, i.e. $|\{p_i \mid a \leq p_i \leq b\}|$. In Section 7 we explain how to extend this to higher dimensions and more complicated ranges. Our solution uses partition trees (introduced by Matoušek in [20]) and traverses them efficiently using the copy-and-recurse method. Next we show a partition for the 1-dim case with the property $\xi = 2$ (i.e., at each node, at most 2 children need to be recursed into).

6.1 The partition tree

A partition tree, T , is a tree data structure constructed for a set P . In this section, we assume each inner node of T has the same number of children, r . We also assume that all leaves have the same height. See Figure 4 for an example. A partition tree that meets these assumptions is easy to construct (in the 1-dim case) as we show below.

Each node of the tree is associated with a subset of P . For a node v , we denote by S_v the subset associated with it. Additionally we require:

- $S_{root} = P$. The root is associated with the entire set P
- $|S_{leaf}| = 1$. A leaf is associated with a single element
- the subsets associated with the children of v form a partition of S_v , i.e. $\cup_i S_{v.child[i]} = S_v$ and $S_{v.child[i]} \cap S_{v.child[j]} = \emptyset$, for $i \neq j$

We stress that S_v is used when building the tree but it is not kept at v . We now list the data that we do keep at each node v :

- (For inner nodes) $child[1], \dots, child[r]$ - the children nodes.
- $\llbracket |S_v| \rrbracket$ - the encryption of $|S_v|$. We note that for a full tree $|S_v| = r^{dist}$, where $dist$ is the number of nodes in a path (distance) from v to a leaf. This makes storing $\llbracket |S_v| \rrbracket$ unnecessary. We still mention it here because this is changed in Sections 7 and 9.
- $\llbracket \sigma \rrbracket = \llbracket [\sigma_{min}], [\sigma_{max}] \rrbracket$ - i.e., the encryption of $\sigma_{min} = \min S_v$ and $\sigma_{max} = \max S_v$. We call the segment $[\sigma_{min}, \sigma_{max}]$ the *bounding segment* of S_v because $x \in [\sigma_{min}, \sigma_{max}]$ for all $x \in S_v$.

6.2 Building a partition tree.

We now describe how to build a partition tree. Recall that we are given $p_1 \leq \dots \leq p_n \in \mathbb{R}$ and a parameter r . Also, recall that for simplicity we consider the case of a full tree which means $n = r^h$, for some $h \in \mathbb{N}$.

We start with $v = root$ and set $S_v = S_{root} = P$. Then we:

- Set $v.|S_v| = |S_v|$.
- Set $\llbracket v.\sigma_{min} \rrbracket = \llbracket [\min S_v, \max S_v] \rrbracket$.
- Partition S_v into r subsets $P_1 = (p_1, \dots, p_{n/r})$,
 $P_2 = (p_{n/r+1}, \dots, p_{2n/r}), \dots, P_r = (p_{n-r+1}, \dots, p_n)$
- Recursively build a sub-tree for $v.child[i]$ until we have $|S_v| = 1$.

6.3 Traversing a partition tree

In this section we describe how to traverse a partition tree is traversed to compute $|P \cap \gamma|$.

As we traverse the tree we keep a counter *count* of points in $|P \cap \gamma|$. At the beginning we set *count* = 0 then we start traversing at the root of the tree. When at a node v we consider each child $u_i = v.child[i]$, compare its bounding segment, $\llbracket u_i.\sigma \rrbracket$, to γ to determine whether it is *contained*, *disjoint* or *crosses* and act as follows:

1. *contained*. If $u_i.\sigma$ is contained in γ then all the points in S_{u_i} should be counted and we add $|S_{u_i}|$ to *count* without recursing into the subtree of u_i .
2. *disjoint*. If $u_i.\sigma$ and γ are disjoint then none of the points in S_{u_i} should be counted and we skip u_i without recursing into its subtree.
3. *cross*. If $u_i.\sigma$ crosses γ (i.e. it intersects but not contained) we need to recurse into the subtree of u_i to determine which of the points in S_{u_i} are contained in γ .

As already hinted, we can efficiently traverse the partition tree using copy-and-recurse. As mentioned in Section 5.1 to do that we need to show that at each node we need to recurse into at most ξ children for some bound $\xi < r$. In the next lemma we prove that a query segment can cross at most 2 of r segments (where any pair overlap in at most one point). This proves that for a 1-dim partition tree we have $\xi = 2$.

Lemma 1. *Let $\sigma_1 = [\sigma_{min_1}, \sigma_{max_1}], \dots, \sigma_r = [\sigma_{min_r}, \sigma_{max_r}]$ be r segments, where any pair overlap in at most one point. Specifically $\sigma_{min_1} \leq \sigma_{max_1} \leq \dots \leq \sigma_{min_r} \leq \sigma_{max_r}$. Then, a segment $\gamma = [\gamma_{min}, \gamma_{max}]$ crosses at most 2 segments of $\sigma_1, \dots, \sigma_r$.*

The claim is intuitive (see Figure 2) and we omit the proof. The implications of this lemma is that at most 2 children need to be recursed into and therefore we can use copy-and-recurse with $\xi = 2$.

Figure 4 shows an example of a partition tree that was built for the numbers 1, 2, 4, 4, 5, 7, 7, 8, 8 and $r = 3$ and how it is used to count the numbers that lie in a query segment $\gamma = [4, 7]$.

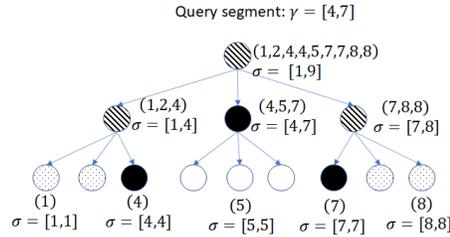


Figure 4: An example of a 1-dim partition tree that was built for the numbers 1,2,4,4,5,7,7,8,8 with $r = 3$. At each node v we show S_v and the bounding segment of S_v (for readability we omit this or some leaves). The figure also shows how the tree is traversed for counting all values in a query segment $\gamma = [4, 7]$. Nodes with $v.\sigma \subset \gamma$ are marked with **solid black**. All values in these nodes can be counted without recursing. Nodes with $v.\sigma \cap \gamma = \emptyset$ are marked with **white**. All values in these nodes can be ignored. Nodes with $v.\sigma$ crossing γ are marked with **black stripes**. These nodes should be recursed into to determine how many of the values they represent are in γ .

In Figure 2 we describe in pseudo-code how a partition tree is traversed efficiently to compute $|P \cap \gamma|$ for the 1-dim case.

In Section 7 we extend this algorithm to higher dimensions and in Section 9 we extend it to compute more general function $f(P \cap \gamma)$.

Algorithm overview Algorithm 2 implements privacy preserving range counting in 1-dim. The public **parameters** of the algorithm are: n and r , where $n = |P|$ is the number of points and r is the number of children each inner node has.

Algorithm 2: $PPCount_{n,r}(\llbracket T \rrbracket, \llbracket \gamma \rrbracket)$

Parameters: $n = |P|$ r the number of children each inner node has.**Input:** A full tree T where v is its root; an encrypted segment $\llbracket \gamma \rrbracket$ **Output:** $\llbracket x \rrbracket$, where $x = |P \cap \gamma|$.

```
1 if  $v$  is a leaf then
  | // check whether  $v.\sigma \subset \gamma$ 
2 |  $\llbracket Cont \rrbracket := IsContaining(\llbracket v.\sigma \rrbracket, \llbracket \gamma \rrbracket)$ 
3 | Output  $\llbracket Cont \rrbracket \cdot \llbracket v.f \rrbracket$ 
4 else
5 | foreach  $i = 1, \dots, r$  do
  | // check whether  $v.child[i].\sigma \subset \gamma$ 
6 |  $\llbracket Cont[i] \rrbracket := IsContaining(\llbracket v.child[i].\sigma \rrbracket, \llbracket \gamma \rrbracket)$ 
7 |  $\llbracket x \rrbracket := \sum_{i=1}^r \llbracket Cont[i] \rrbracket \cdot \llbracket v.child[i].f \rrbracket$ 
8 | foreach  $i = 1, \dots, r$  do
  | // check whether  $c.child[i].\sigma$  crosses  $\gamma$ 
9 |  $\llbracket Cross[i] \rrbracket := IsCrossing(\llbracket v.child[i].\sigma \rrbracket, \llbracket \gamma \rrbracket)$ 
10 |  $\llbracket M \rrbracket := BuildSeletionMatrix_{r,2}(\llbracket Cross \rrbracket)$  // See Algorithm 3
11 |  $\llbracket child' \rrbracket := M \cdot \llbracket v.child \rrbracket$ 
12 |  $\llbracket Cross' \rrbracket := M \cdot \llbracket Cross \rrbracket$ 
13 |  $\llbracket x \rrbracket := \llbracket x \rrbracket + \sum_{i=1}^2 \llbracket Cross'[i] \rrbracket \cdot PPCount_{n/r,r}(\llbracket child'[i] \rrbracket, \llbracket \gamma \rrbracket)$ 
14 | Output  $\llbracket x \rrbracket$ 
```

The **input** of the algorithm is $\llbracket T \rrbracket$ and $\llbracket \gamma \rrbracket$. T is a partition tree and γ is a segment. We use a ciphertext notation for T because the (private) content stored at each node (e.g. $v.\sigma$ and $v.f$) are encrypted. We note that the structure of T (e.g. which node is the child of which) is not encrypted.

The **output** of the algorithm is $\llbracket x \rrbracket$ where $x = |P \cap \gamma|$.

Algorithm 2 works recursively. When it is first called it operates on the root of the tree. Then it recurses into some of the children of the root by calling itself with the subtree of these children as input. For example, to recurse into $root.child[i]$ it calls $PPCount(\llbracket root.child[i] \rrbracket, \llbracket \gamma \rrbracket)$. The recursion stops when it reaches a leaf. While traversing the partition tree the algorithm sums $\llbracket v.f \rrbracket$ from various nodes (as we explain below). The improved efficiency of the algorithm comes from the property (proved in Lemma 1) that at most 2 children need to be recursed into. This is done under *FHE* using copy-and-recurse and more specifically, by making a copy of 2 children together with their subtrees. Lemma 1 proves that the children we actually need to recurse into are among the 2 children we copy.

Algorithm 2 starts by checking the stopping condition of the recursion (Line 1). For a leaf, v , it checks whether $v.\sigma \subseteq \gamma$ (Line 2). This is done by calling the function $IsContaining$ which returns $\llbracket c \rrbracket$, where $c = 1$ if $v.\sigma \subseteq \gamma$ and 0 otherwise. Then we use $\llbracket c \rrbracket$ as a multiplexer to output $|S_v|$ or 0 (Line 3).

When v is an inner node (Lines 4-14) the algorithm checks for each child of v whether its bounding segment, σ , contains γ (Line 6). If $S_{v.child[i]} \subset v.child[i].\sigma \subset \gamma$ we can count $v.f = |S_{v.child[i]}|$ without checking the points of $S_{v.child[i]}$. Then, the algorithm finds children whose bounding segment crosses γ (Line 9). These values are kept in a r -dimensional binary vector $Cross$. The copy-and-recurse method is implemented by multiplying by a selection matrix M . The matrix is generated (Line 10) using the *BuildSelectionMatrix* algorithm (see Algorithm 3). Then the algorithm copies the subtrees (Line 11) by computing $M \cdot v.child$ (here we regard $v.child$ as a vector with r elements ($v.child[1], \dots, v.child[r]$) where each element is a subtree). The output is a vector $child'$ with only 2 elements. Similarly we copy $Cross$ into $Cross'$. We then recurse into the subtrees in $child'$ to check a finer partition (i.e., into smaller sets) of their points Line 13. We add the output of these recursions into the output of the algorithm.

7 Range searching in d -dim

In this section we explain how partition trees are extended to answer range searching in d dimensions.

In d -dim the **input** $P \subset \mathbb{R}^d$ is a set of points (an not values in \mathbb{R}). The **query** range is some volume $\gamma \subset \mathbb{R}^d$ (e.g., a sphere, a polytope, etc. See more on this below). At each node v we keep a **bounding simplex**¹ $v.\sigma$ that contains all the points in S_v .

¹A simplex in \mathbb{R}^d is the convex hull of $d + 1$ points.

The concept of *partition tree* is generic and can be used in d dimensions as long as we supply a d -dim version of a partition theorem. Unlike the 1-dim case, in d -dim it is not trivial to find a partition to subsets of equal sizes bounded by simplices where the number of simplices crossing a query is bounded. Even the simple planar case with query ranges being the area under a query line received a lot of attention from the computational geometry community in the past. Looking ahead, the machinery we use can handle arbitrary ranges in d -dim as long as they are not “too complex” as we now explain.

In what follows we give some computational geometry preliminaries that were deferred until now to improve readability and then we describe the changes that need to be made in Algorithm 2 to support multiple dimensions.

7.1 Computational geometry preliminaries

In this section we explain some terminology in computational geometry. This terminology comes in handy in Section 7.

Range space. A range space is a pair (X, Γ) , where X is a set and $\Gamma \subset 2^X$ is a *family* of subsets, called ranges. We consider $X = \mathbb{R}^d$ for some d .

Range Searching. The range searching problem studied in computational geometry is: given a set of n points $P \subset \mathbb{R}^d$ and a family of ranges Γ , preprocess P into a data structure \mathbb{D} , such that given a range $\gamma \in \Gamma$ and using \mathbb{D} we can efficiently compute $|P \cap \gamma|$.

Algebraic range. A d -dimensional algebraic range is a subset $\gamma \subset \mathbb{R}^d$ defined by an algebraic surface given by a function that divides \mathbb{R}^d into two regions (e.g. above and below).

Semi-algebraic range. A d -dimensional semi-algebraic range is a subset $\gamma \subset \mathbb{R}^d$ that is a conjunction and disjunction of a bounded number of algebraic ranges. Simply put, a semi-algebraic range is the result of intersections and unions of algebraic ranges.

Constant description complexity. The description complexity of a range is the number of parameters needed to describe it. One example is a half-space range bounded by a plane in \mathbb{R}^3 $ax + by + cz + 1 = 0$ which has 3 parameters a, b , and c . Another example is a sphere $(x - a)^2 + (y - b)^2 + (z - c)^2 \leq r^2$, which has 4 parameters: a, b, c, r . The description complexity may depend on n (for example a star-shaped volume in \mathbb{R}^3 with n “spikes” has $O(n)$ parameters). There is a connection between the number of parameters and the “hardness” of the range searching problem. Intuitively, it is harder to answer queries when ranges have more parameters. The machinery we use requires the ranges to have a constant number of parameters in their description (i.e., to have a constant description complexity).

Elementary Cell Partition (or Simplicial Partition). Given a set $P \subset \mathbb{R}^d$ of n points, an elementary cell partition (or simplicial partition) is a collection $\Pi = \{(P_1, \sigma_1), \dots, (P_m, \sigma_m)\}$ where P_i ’s are disjoint subsets such that $\cup P_i = P$ and each $P_i \subset \sigma_i$, where σ_i is simplex. We say the size of the partition is m .

Crossing number. Given a simplicial partition $\Pi = \{(P_1, \sigma_1), \dots, (P_m, \sigma_m)\}$ and a range γ , the crossing number of γ with respect to Π is the number of simplices γ crosses, i.e. $|\{\sigma_i \mid \sigma_i \cap \gamma \neq \sigma_i \text{ and } \sigma_i \cap \gamma \neq \emptyset, \text{ for } i = 1, 2, \dots, m\}|$.

Partition Theorem. In a seminal work [20] Matoušek showed a non-trivial partition with small crossing number when Γ is the set of halfspaces bounded by hyperplanes. In [1] it was extended to general semi-algebraic ranges with constant description complexity. The most recent partitioning is due to [2] who improved the bound on the crossing number. Their result is summarized in the following theorem.

Theorem 2 (From [2]). *Given a set P of n points in \mathbb{R}^d , for some fixed d , a family of semi-algebraic ranges of constant description complexity Γ and a parameter $r \leq n$, an elementary cell partition $\Pi = \{(P_1, \sigma_1), \dots, (P_m, \sigma_m)\}$ can be computed in randomized expected time $O(nr + r^3)$ such that:*

1. $\lfloor n/r \rfloor \leq |P_i| < h \lfloor n/r \rfloor$ for every i and some constant h .
2. The crossing number of Π is $O(r^{1-1/d})$.

7.2 Changes to algorithms

We now describe the changes needed to be made in Algorithm 2 to support higher dimensions.

As mentioned above, in d dimensions we use Theorem 2 when constructing the partition tree. The tree constructed with this theorem has $r/h \leq m \leq r$ children at each node and the height of a leaf is $\lfloor \log_r n \rfloor \leq \text{height} \leq \lfloor \log_{r/h} n \rfloor$. This follows from the subset sizes at node v being $\lfloor |S_v|/r \rfloor \leq m \leq h \lfloor |S_v|/r \rfloor$.

The variable number of children and variable leaf height raises 2 problems: (1) the tree structure may leak information on the input and (2) the copy-and-recurse prerequisites are not met. To solve these 2 problems we describe in Section 7.3 how a partition tree can be filled by adding empty nodes. For the remainder of this section we assume the given tree is full.

Changes to *RangeSearchingProtocol* The changes needed to apply to *RangeSearchingProtocol* to support high dimension include getting d and Γ as parameters, where d is the dimension and Γ is the family of all possible query ranges. d and Γ are used when the partition tree is constructed (they are needed by the Theorem 2).

Changes to *PPCount* The changes *PPCount* include:

- The parameters now include $d, \Gamma, IsContaining$ and $IsCrossing$ (more precisely, *PPCount* needs only *IsContaining* and *IsCrossing* but their implementation depends on d and Γ).
- Call *BuildSeletionMatrix* with r and ξ as parameters. (Line 10)
- Recurse into the ξ children that were copied (Line 13).

7.3 Hiding tree structure

As mentioned above using Theorem 2 results in a partition tree that is not full. This may have security issues. Also, it does not meet the prerequisites of the copy-and-recurse method. In this section we give a recipe to “fill” trees by adding empty nodes to them (as we explain below) until the tree becomes full i.e.: (1) each node has the maximal number of children and (2) all leaf is at the maximal height. As our analysis below shows, this addresses the security problem because the structures of 2 full trees (built with the same parameters n and r) are indistinguishable. Our analysis also shows the size of the tree grows from $O(n)$ to $O(n^{1+\epsilon})$. We first define an *empty node* and then explain how they are added.

Definition 1 (Empty Node). *An empty node is a node v that is associated with an empty set, $S_v = \emptyset$ and its simplex is a degenerated empty simplex, $v.\sigma = \emptyset$.*

To hide the structure of a tree we add empty nodes until (1) all inner nodes have r children and (2) the height of each leaf is $\lceil \log_{r/h} n \rceil$. For completeness we describe this algorithm in Appendix C.

We conclude this section by stating 2 lemmas whose proofs are given in Appendix C.

Lemma 4. *Let P be a set of n points in \mathbb{R}^d , Γ a family of ranges, $r < n$ a parameter and h a parameter such that any simplicial partition of P' with respect to Γ , $\Pi = \{(P'_1, \sigma_1), \dots, (P'_m, \sigma_m)\}$ satisfies $|P'|/r < |P'_i| < h \cdot |P'|/r$ and let $T = \text{FillTree}(T', n, r, h)$, where T' is a partition tree built for P and Γ , then the height of T is $\lceil \log_{r/h} n \rceil$ and it has a total of $n^{\frac{1}{1-\lceil \log_{r/h} h \rceil}} = O(n^{1+\epsilon})$ nodes.*

Lemma 5. *Let $P_1, P_2 \subset \mathbb{R}^d$ be 2 sets of points with $|P_1| = |P_2| = n$ and T'_1, T'_2 be 2 partition trees built for P_1 and P_2 , respectively, with the same parameters r, h then T_1 and T_2 have the same structure, where $T_i = \text{FillTree}(n, r, h, T'_i)$.*

8 Size And Depth Analysis

In this section we analyze the size and depth of a circuit that implements *PPRangeSearch* (Algorithm 2) to compute $f(P \cap \gamma)$. We start by analyzing the size and depth of *BuildSeletionMatrix* which is used by *PPRangeSearch*.

8.1 Analyzing *BuildSeletionMatrix*

Depth and size analysis. The Analysis of the size and depth of a circuit implementing *BuildSeletionMatrix* $_{r,\xi}$ is summarized in the following lemma.

Lemma 2. *Computing $M[\text{col}, \text{row}]$ for $1 \leq \text{col} \leq \xi$ and $1 \leq \text{row} \leq r$ can be done with a circuit of depth $O(\xi \cdot \log r)$ and size $O(\xi \cdot r^2)$.*

Proof. We prove the lemma by induction on ξ . For $\xi = 1$ we have $M[1, row] := c[row] \cdot \prod_{i=1}^{row-1} (1 - c[i])$ which can be done for all $1 \leq row \leq r$ in a circuit of depth $O(\log row)$ and size $O(row)$. Computing for all rows in parallel we get a circuit of depth $O(\log r)$ and size $O(r^2)$.

Assuming it holds for all $\xi' < \xi$ we prove it holds for ξ . Since we have $M[\xi, row] = c[row] \sum_{k=1}^{row-1} \left(M[\xi - i, k] \prod_{h=k+1}^{row-1} (1 - c[h]) \right)$ this can be done for all $1 \leq row \leq r$ with a circuit whose depth is $O(\log r + (\xi - 1) \log r)$ and size is $O(r^2 + (\xi - 1)r^2)$, which proves the claim. \square

8.2 Analyzing *PPRangeSearch*

We now turn to analyze *PPRangeSearch* (Algorithm 2). As mentioned in Section 3.3.1 we denote by t, ℓ the size and depth of the circuit that realizes *IsContaining* or *IsCrossing*.

Analyzing the space of a tree is now easy.

Lemma 3 (Space). *Let P, T be as in Lemma 4, where $|P| = n$ and $r < n$ is a parameter, then T needs space of $O(n^{1+\epsilon})$, where the value of ϵ depends on r and can be made arbitrarily small.*

Proof. From Lemma 4 the number of nodes is $n^{\frac{1}{1-\log_r h}}$. Since we keep $O(1)$ data with each node the total space is $O(n^{1+\epsilon})$, where $\epsilon = \frac{\log_r h}{1-\log_r h}$ can be made arbitrarily small by choosing a large r . \square

We now turn to analyze the size and depth of the circuit that computes a range search query.

Theorem 1. *Let $P \subset \mathbb{R}^d$ be a set of n points, $\Gamma \subset 2^{\mathbb{R}^d}$ a family of semi-algebraic ranges, T a full partition tree as output from Algorithm 4, a function f that can be computed in a divide and conquer manner and t and ℓ are the size and depth of the circuit that compares a range to a simplex, then given $\gamma \in \Gamma$, *PPRangeSearch* (Algorithm 2) securely evaluates $f(\gamma \cap P)$ in a circuit whose size is $O(n^{1+\epsilon} + t \cdot n^{1-\frac{1}{d}+\epsilon})$ and depth is $O(\ell \cdot \log n)$.*

Informally, the correctness follows from the plaintext algorithm that Algorithm 2 implements. The bound on the circuit size is proved by solving the recursion formula of the circuit size. The circuit depth is proved by induction on the tree height. For lack of space, we give the full proof in Appendix B.

8.3 Privacy analysis

In this section we discuss the privacy of the inputs in the presence of dishonest adversaries. As mentioned above, we consider 3 parties: (1) the data owner; (2) the cloud and (3) the range owner who also holds the secret key and gets the output. We assume the data owner colludes with the query owner but not with the cloud and claim the cloud does not learn anything on the context of the encrypted input it receives. The range owner is the only party who receives

an output. As mentioned, this model is motivated by the growing trend to outsource databases to untrusted clouds to save maintenance and IT costs.

For the cloud, we consider a computationally bounded, semi-honest (a.k.a. honest but curious) adversary that follows the protocol but tries to infer additional information to what is stated above.

Informally, the security of our algorithm stems from the semantic security of FHE. For lack of space we discuss the security more formally in Appendix D.

9 extending to generic f

Until now we have discussed the counting problem in which we compute $|P \cap \gamma|$. In this section we show how we can modify *PPCount* to compute $f(P \cap \gamma)$ for functions f that can be computed in a divide and conquer manner. Note that computing $|P \cap \gamma|$ is a special case in which $f(A) = |A|$.

Looking at f we see that its input² is a subset of P and its output is a value $v \in V$ which we have no restriction on (e.g., for the counting problem we have $V = \mathbb{N}$). Additionally, we require that f can be computed in a divide and conquer manner. For the case of counting, we have $f(A \cup B) = f(A) + f(B)$.

In what follows we describe the changes that need to be made on *PPCount*. In Appendix E we give a few useful applications that use different functions f and g .

Changes needed to be made on *PPCount*:

- Set $v.f = f(S_v)$, i.e. each node in the partition tree keeps the value of f applied on S_v .
- The output when v is a leaf is $\llbracket Cont \rrbracket \cdot \llbracket v.f \rrbracket + (1 - \llbracket Cont \rrbracket) \cdot f(\emptyset)$, i.e. return $v.f$ if $Cont = 1$ and $f(\emptyset)$ otherwise.
- Adding the contribution of children that are contained in γ should use g and be: **foreach** $i = 1, \dots, r$ **do**
 $\llbracket x \rrbracket := \llbracket Cont[i] \rrbracket \cdot g(\llbracket x \rrbracket, \llbracket v.child[i].f \rrbracket, \llbracket \gamma \rrbracket) + (1 - \llbracket Cont[i] \rrbracket) \cdot \llbracket x \rrbracket$
- Adding the contribution of children that cross γ should use g and be: **foreach** $i = 1, \dots, \xi$ **do**
 $\llbracket x \rrbracket := \llbracket Cross[i] \rrbracket \cdot g(\llbracket x \rrbracket, PPCount(child'[i], \gamma)) + (1 - \llbracket Cross'[i] \rrbracket) \cdot \llbracket x \rrbracket$

In Appendix E we give a few useful applications for privacy preserving range searching.

²we avoid the more acceptable terms “domain” and “range” to avoid confusion

10 Experiments

In this section we report experiments we made to check our copy-and-recurse method with partition trees. We compared our method to the naïve solution (described below). As far as we know there are no better solutions. In what follows we describe the experiments and comparisons we made.

What we tested. Our method is generic for dimension d , range family Γ and function f as defined in previous sections, but for the experiments we set the parameters as follows. In the first experiment we set $d = 1$ (i.e. $P \subset \mathbb{R}$) and the range family Γ is the set of all segments. In the second experiment we set $d = 2$ (i.e. $P \subset \mathbb{R}^2$) and Γ is set to the set of all axis-parallel rectangles. In both experiments the goal was to count $f(P \cap \gamma) = |P \cap \gamma|$. This type of queries often arises in databases. When running our method we used several values of r . Specifically, we set $r = 3, 5, 7, 9$.

Setup. Our method is independent of a specific FHE scheme, and so in our experiments we used the CKKS scheme [11], which is a popular scheme. To implement the *IsSmaller* function we used the work of [12].

We used CKKS when implementing both the naïve and our methods. We used the HELayers framework [3] which uses the HEaaN library [15] as the implementation for CKKS. We set the parameters of the keys to have 128 bits of security with 2^{15} slots, chain length 12, integer precision of 18 bits and fractional precision of 42 bits. With these parameters the scheme also supported bootstrapping. The system we used had 64 cores (128 threads) of AMD EPYC 7742 CPU with 500 GB memory and NVIDIA A100-SXM4-80GB GPU.

Packing and SIMD Our technique is independent of SIMD, but in our experiments we stored each number $p \in P$ in a different slot for a total of 2^{15} elements of P in a single ciphertext.

The naïve solution. The naïve solution we implemented is similar to that of Cheon et al. [13] in the sense that given a set P of n points and a range γ we check for each point p whether $p \in \gamma$, for a total of $O(n)$ such checks. We remind that the points were packed in SIMD manner, so that only $O(n/\text{slots})$ checks are actually needed. In our experiments, checking whether $p \in \gamma$ was made by comparing $p \in \mathbb{R}$ to the two endpoints of γ . The output of these comparisons was a binary vector χ of size n with 1 for $p \in \gamma$ and 0 for $p \notin \gamma$. Summing the elements of χ yields $|P \cap \gamma|$.

What we measured. To compare our method to the naïve method we ran both algorithms and measured their running time over different database sizes.

10.1 Results in 1-dim

Our results are summarized in Table 2 and in Figure 5. Table 2 shows the running time (reported in minutes) of copy-and-recurse algorithm with different values of r and of the naïve algorithm. The algorithms were run multiple times and the reported result is the minimum of the measured running times (this is to eliminate skews in running times). Each column reports the running time of the algorithm specified in the column’s title. For example, the running times of *PPCount* using the copy-and-recurse algorithm with $r = 3$ are reported in the 2nd column. Each row reports the running time of the algorithms over database of of different size. For example, the 4th row shows the case where $|P| = 2^{21}$. These results also appear in Figure 5. The x -axis is the size of P and the y -axis is the running time of the algorithm.

We see that our method is faster when $|P| > 2^{23}$ (i.e., even for a relatively small database). For example, when $|P| = 2^{23}$ the naïve method took 49.36 minutes to run whereas our algorithm took less than 17 minutes when setting $r = 3$. The time difference grows in our favor as number of points increases. For example, when $|P| = 2^{25}$ the naïve took 197.04 minutes and our code took 42.71 minutes. This is expected and follows from our analysis as we now explain.

The naïve time is linear with a factor of $t \approx 6 \cdot 10^{-6}$ as expected since its time complexity is $O(tn)$. Indeed, the running times grows approximately four folds when we increase $|P|$ four folds. For example, the ratio between the running times for $|P| = 2^{19}$ and for $|P| = 2^{17}$ is 3.89.

The running time of the copy-and-recurse method is, largely speaking, linear. The running times grows approximately four folds when we increase $|P|$ four folds. This is expected and follows from our analysis that the running time is $O(n^{1+\epsilon} + tn^\epsilon)$. Indeed, for small values of n the part tn^ϵ is more dominant but it becomes less dominant as n grows. This is the reason we see sub-linear growth when $|P| > 2^{21}$. The results are also summarized in Figure 5.

$ P $	Partition tree using copy-and-recurse				Naïve
	$r = 3$	$r = 5$	$r = 7$	$r = 9$	
2^{15}	0.38	0.39	0.39	0.38	0.21
2^{17}	1.86	1.47	1.47	1.47	0.74
2^{19}	6.97	7.15	6.85	6.35	2.92
2^{21}	16.07	18.01	14.84	26.96	11.59
2^{23}	16.98	23.38	48.24	38.94	49.36
2^{25}	42.71	56.27	100.72	74.59	197.04

Table 2: The running time (in minutes) when running a range count query, i.e., finding $|P \cap \gamma|$ where $P \subset \mathbb{R}$. The first column shows the size of the database, $|P|$. The next 4 columns show the running time of answering range counting using copy-and-recurse with $r = 3, 5, 7, 9$. The last column shows the running time of the naïve method.

10.2 Results in higher dimensions

We also repeated our experiment in 2 dimensions. In this case Γ was the set of all axis-parallel rectangles. In this case, we built a 1-dim partition tree for P projected on the x -axis and then built a secondary tree at each node. This is a standard method (see for example Section 16.2 in [9]) to answer range queries when the ranges are conjunctions of algebraic ranges. The results are summarized in Table 3. For example, as before the naïve algorithm is linear while our algorithm is sub-linear. For example, when $|P| = 2^{25}$ our method took 700 minutes, while the naïve algorithm took 910 minutes.

$ P $	Partition tree using copy-and-recurse	
	$r = 4$	Naïve
4	13	3
16	64	14
64	161	54
256	335	213
1024	700	910

Table 3: The running time (in minutes) when running a range count query in 2 dimensions, i.e., finding $|P \cap \gamma|$ where $P \subset \mathbb{R}^2$. The first column is the size of P . The second column shows the running time of answering range counting using copy-and-recurse with $r = 3$. The last column shows the running time of the naïve method.

11 Conclusion

We showed a method we call copy-and-recurse and showed it can efficiently traverse a tree when there's a bound on the number of children we need to recurse into.

We showed how to apply this method to efficiently implement partition trees under FHE. Partition trees are a powerful tool for solving range query problems (we note the proliferation of problems in plaintext that were solved using partition trees). In this paper we have seen a few applications that can be stated as range searching problems. Specifically, many database queries can be stated as range queries.

In the general case, when the ranges are semi-algebraic with constant description complexity we showed how to answer a range search query with a circuit whose size is $O(n^{1+\epsilon} + t \cdot n^{1-\frac{1}{d}+\epsilon})$ and depth is $O(\ell \cdot \log n)$, where n is the number of points, d is their dimensionality and t is the time to check how a range interacts with a point simplex.

Since, more complex ranges mean a higher t value and since in practice these comparisons are the dominant part of the running time our result improves over the naïve implementation. We remind that $O(n)$ is a lower bound when

running under FHE, and $O(t \cdot n^{1-\frac{1}{d}+\epsilon})$ is the best bound known in plaintext when allowing near linear storage.

The efficiency in our result comes from the way we traverse the partition tree which takes advantage of its properties, namely there is a bound ξ on the number of children we need to recurse into. We then recurse into ξ children thus achieving similar results as the plaintext algorithm (which add to the $O(n^{1+\epsilon})$ overhead to allow this recursion). We believe the copy-and-recurse method can be useful in more tree and tree-like constructions.

We implemented a system to demonstrate the efficiency of our method. Our implementation shows that our method outperforms the naive implementation even for small values of n . In future work we will show more applications that are solved efficiently with range searching.

References

- [1] Pankaj K. Agarwal and Jiří Matoušek. On range searching with semialgebraic sets. *DISCRETE COMPUT. GEOM*, 11:393–418, 1994.
- [2] Pankaj K. Agarwal, Jiří Matoušek, and Micha Sharir. On range searching with semialgebraic sets. ii. *SIAM Journal on Computing*, 42(6):2039–2062, 2013.
- [3] Ehud Aharoni, Allon Adir, Moran Baruch, Nir Drucker, Gilad Ezov, Ariel Farkash, Lev Greenberg, Ramy Masalha, Guy Moshkovich, Dov Murik, Hayim Shaul, and Omri Soceanu. HeLayers: A Tile Tensors Framework for Large Neural Networks on Encrypted Data. *CoRR*, abs/2011.0, 2020.
- [4] Adi Akavia, Dan Feldman, and Hayim Shaul. Secure data retrieval on the cloud: Homomorphic encryption meets coresets. *IACR Transactions on Cryptographic Hardware and Embedded Systems*, 2019(2):80–106, Feb. 2019.
- [5] Adi Akavia, Max Leibovich, Yehezkel S Resheff, Roey Ron, Moni Shahaar, and Margarita Vald. Privacy-preserving decision trees training and prediction. *ACM Transactions on Privacy and Security*, 25(3):1–30, 2022.
- [6] Sofiane Azogagh, Victor Delfour, Sébastien Gambs, and Marc-Olivier Killijian. Probonite: Private one-branch-only non-interactive decision tree evaluation. In *Proceedings of the 10th Workshop on Encrypted Computing & Applied Homomorphic Cryptography, WAHC’22*, page 23–33, New York, NY, USA, 2022. Association for Computing Machinery.
- [7] Donald Beaver. Efficient multiparty protocols using circuit randomization. In Joan Feigenbaum, editor, *Advances in Cryptology — CRYPTO ’91*, pages 420–432, Berlin, Heidelberg, 1992. Springer Berlin Heidelberg.

- [8] Amos Beimel, Yuval Ishai, and Tal Malkin. Reducing the servers computation in private information retrieval: Pir with preprocessing. In *Annual International Cryptology Conference*, pages 55–73. Springer, 2000.
- [9] Mark de Berg, Otfried Cheong, Marc van Kreveld, and Mark Overmars. *Computational Geometry: Algorithms and Applications*. Springer-Verlag TELOS, Santa Clara, CA, USA, 2nd ed. edition, 2000.
- [10] Zvika Brakerski, Craig Gentry, and Vinod Vaikuntanathan. (leveled) fully homomorphic encryption without bootstrapping. In *Proceedings of the 3rd Innovations in Theoretical Computer Science Conference*, ITCS '12, pages 309–325, New York, NY, USA, 2012. ACM.
- [11] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. Homomorphic encryption for arithmetic of approximate numbers. In Tsuyoshi Takagi and Thomas Peyrin, editors, *Advances in Cryptology – ASIACRYPT 2017*, pages 409–437, Cham, 2017. Springer International Publishing.
- [12] Jung Hee Cheon, Dongwoo Kim, and Duhyeong Kim. Efficient homomorphic comparison methods with optimal complexity. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 221–256. Springer, 2020.
- [13] Jung Hee Cheon, Miran Kim, and Myungsun Kim. Search-and-compute on encrypted data. In *International Conference on Financial Cryptography and Data Security*, pages 142–159. Springer, 2015.
- [14] Kelong Cong, Debajyoti Das, Jeongeun Park, and Hilder V.L. Pereira. Sortinghat: Efficient private decision tree evaluation via homomorphic encryption and transciphering. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, CCS '22, page 563–577, New York, NY, USA, 2022. Association for Computing Machinery.
- [15] CryptoLab. HEaaN: Homomorphic Encryption for Arithmetic of Approximate Numbers, version 3.1.4, 2022.
- [16] 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.
- [17] Ioannis Demertzis, Stavros Papadopoulos, Odysseas Papapetrou, Antonios Deligiannakis, Minos Garofalakis, and Charalampos Papamanthou. Practical private range search in depth. *ACM Transactions on Database Systems (TODS)*, 43(1):1–52, 2018.
- [18] Francesca Falzon, Evangelia Anna Markatou, Zachary Espiritu, and Roberto Tamassia. Range search over encrypted multi-attribute data. *Cryptology ePrint Archive*, 2022.

- [19] Iliia Iliashenko, Malika Izabachene, Axel Mertens, and Hilder V L Pereira. Homomorphically counting elements with the same property. page 20.
- [20] Jiří Matoušek. Efficient partition trees. In *Proceedings of the Seventh Annual Symposium on Computational Geometry*, SCG '91, page 1–9, New York, NY, USA, 1991. Association for Computing Machinery.
- [21] Anselme Tueno, Yordan Boev, and Florian Kerschbaum. Non-interactive private decision tree evaluation. In Anoop Singhal and Jaideep Vaidya, editors, *Data and Applications Security and Privacy XXXIV*, pages 174–194, Cham, 2020. Springer International Publishing.
- [22] Anselme Tueno, Florian Kerschbaum, and Stefan Katzenbeisser. Private evaluation of decision trees using sublinear cost. *Proceedings on Privacy Enhancing Technologies*, 2019(1):266–286, 2019.

A Building The selection Matrix

In this section we describe $BuildSeletionMatrix_{r,\xi}(\llbracket c \rrbracket)$, a function that generates the selection matrix, M , used in Algorithm 2. The function has 2 parameters r and ξ . In addition, it gets as input an encrypted vector $\llbracket c \rrbracket$, where $c \in \{0, 1\}^r$ and ξ is an upper bound of the number of non-zero elements in c . The output of $BuildSeletionMatrix_{r,\xi}(\llbracket c \rrbracket)$ is a matrix $M \in \{0, 1\}^{\xi \times r}$, such that for any vector $x \in \mathbb{R}^r$ we have

$$(M \cdot x)[j] = \begin{cases} x[i] & \text{if } c[i] \text{ is the } j\text{-th value of 1.} \\ 0 & \text{if } c[i] \text{ has less than } j \text{ non-zero elements.} \end{cases}$$

Algorithm 3: $BuildSeletionMatrix_{r,\xi}(\llbracket c \rrbracket)$

Parameters: $\xi < r$.

Input: A vector $\llbracket c \rrbracket$, where $c \in \{0, 1\}^r$, s.t. $\sum c_i \leq \xi$.

Output: A selection matrix $\llbracket M \rrbracket$ that selects the non-zero elements in c .

```

1 for ( row = 1, ..., r )
2   |  $\llbracket M[1, row] \rrbracket := \llbracket c[row] \rrbracket \cdot \prod_{i=1}^{row-1} (1 - \llbracket c[i] \rrbracket)$ 
3 for ( col = 2, ...,  $\xi$  )
4   | for ( row = 1, ..., r )
5     |  $\llbracket M[col, row] \rrbracket :=$ 
6       |  $\llbracket c[row] \rrbracket \sum_{k=1}^{row-1} \left( \llbracket M[col - i, k] \rrbracket \prod_{h=k+1}^{row-1} (1 - \llbracket c[h] \rrbracket) \right)$ 
6 Output:  $\llbracket M \rrbracket$ 

```

To understand how Algorithm 3 works we note that $M[i, j] = 1$ iff $c[i]$ is the j -th cell with a value of 1. Algorithm 3 starts by setting (Line 2)

$$M[1, i] = c[i] \prod_{k=1}^{i-1} (1 - c[k]).$$

It is easy to see that $M[1, i] = 1$ iff $c[i]$ is the first non-zero element in c , i.e. $c[i] = 1$ and $c[k] = 0$ for $1 \leq k < i$. Then, Algorithm 3 continues by setting (Line 5)

$$M[j, i] = c[i] \sum_{k=1}^{i-1} \left(M[j-i, k] \prod_{h=k+1}^{i-1} (1 - c[h]) \right)$$

which we now explain. $M[j-i, k] = 1$ iff $c[k]$ is the $(j-i)$ -st element with a value of 1. $\prod_{h=k+1}^{i-1} (1 - c[h]) = 1$ iff $c[k+1] = \dots = c[i-1] = 0$. Putting these together and summing for all values of $k < i$ we get that $\sum_{k=1}^{i-1} \left(M[j-i, k] \prod_{h=k+1}^{i-1} (1 - c[h]) \right) = 1$ if there are exactly $j-i$ values of 1 in $c[1], \dots, c[i-1]$. Multiplying this by $c[i]$ we get that $M[j, i] = 1$ iff $c[i]$ is the j -th value of 1.

B Proof Of Theorem 1

In this Section we give the proof to Theorem 1.

Theorem 1. *Let $P \subset \mathbb{R}^d$ be a set of n points, $\Gamma \subset 2^{\mathbb{R}^d}$ a family of semi-algebraic ranges, T a full partition tree as output from Algorithm 4, a function f that can be computed in a divide and conquer manner and t and ℓ are the size and depth of the circuit that compares a range to a simplex, then given $\gamma \in \Gamma$, $PPRangeSearch$ (Algorithm 2) securely evaluates $f(\gamma \cap P)$ in a circuit whose size is $O(n^{1+\epsilon} + t \cdot n^{1-\frac{1}{d}+\epsilon})$ and depth is $O(\ell \cdot \log n)$.*

Proof. Correctness. The correctness of the plaintext algorithm for range searching was proven in [20, 1, 2]. Our construction deviates from the plaintext algorithm in 3 ways: (1) it adds empty nodes; (2) it always recurses into ξ children (for inner nodes) and (3) it uses the *Cross* and *Cont* indicator arrays to conditionally aggregate values into the output. These do not change the functionality of algorithm.

Circuit Size. At each inner node, v , Algorithm 2: (1) computes *IsContaining* and *IsCrossing* r times; (2) builds a selection matrix M ; (3) copies ξ children of v and (4) recurses into ξ children of v . Computing all *IsContaining* and *IsCrossing* takes $O(t \cdot r)$ time. From Lemma 2, computing the selection matrix takes $O(r^2 \cdot \xi)$. The size of each child (including its subtree) is $O((n/r)^{\frac{1}{1-\log_r h}}) = O((n/r)^{1+\epsilon})$ and copying ξ children (out of r) takes $O(r \cdot \xi \cdot (n/r)^{1+\epsilon})$.

It follows that the time to compute a range query is given by the following recursion rule:

$$T(n) \leq O(r \cdot t) + O(r^2 \cdot \xi) + O(r \cdot \xi \cdot (n/r)^{1+\epsilon}) + \xi \cdot T(h \cdot n/r)$$

This solves to

$$\begin{aligned}
T(n) &= \sum_{i=0}^{\log_{r/h} n - 1} O((r \cdot t + r^2 \cdot \xi) \cdot \xi^i) + \sum_{i=0}^{\log_{r/h} n - 1} O(r \cdot \xi \left(\frac{n}{r^{i+1}}\right)^{1+\epsilon} \cdot \xi^i) \\
&= (r \cdot t + r^2 \cdot \xi) \frac{\xi^{\log_{r/h} n} - 1}{\xi - 1} + O(r \cdot n^{1+\epsilon} \cdot \frac{1 - (\frac{\xi}{r^{1+\epsilon}})^{\log_{r/h} n + 1}}{1 - \frac{\xi}{r^{1+\epsilon}}}) \\
&= O((r \cdot t + r^2 \cdot \xi) \xi^{\log_{r/h} n} + r \cdot n^{1+\epsilon}). \quad (1)
\end{aligned}$$

For the case $d = 1$ we have from Lemma 1 $\xi = 2$ and $h = 1$. Putting these into Equation 1 we get

$$O((r \cdot t + r^2) 2^{\log_r n} + r \cdot n^{1+\epsilon}) = O((r \cdot t + r^2) n^{\log_r 2} + r \cdot n^{1+\epsilon}) = O(n^{1+\epsilon} + t \cdot n^\epsilon)$$

For the case $d > 2$ we have from Theorem 2 $\xi = O(r^{1-1/d})$ and $h = O(1)$. Putting these into Lemma 1 we get

$$\begin{aligned}
&O((r \cdot t + r^2 \cdot r^{1-1/(2d-4)}) (r^{1-1/d})^{\log_{r/h} n} + r \cdot n^{1+\epsilon}) \\
&= O((r \cdot t + r^{3-1/d}) (r^{1-1/d})^{\log_r n \log_{r/2} r} + r \cdot n^{1+\epsilon}) \\
&= O((r \cdot t + r^{3-1/d}) \cdot n^{(1-1/d)(1+\log_{r/h} h)} + r \cdot n^{1+\epsilon}) \\
&= O(n^{1+\epsilon} + t \cdot n^{1-1/d+\epsilon})
\end{aligned}$$

Putting these together we get the circuit size is $O(n^{1+\epsilon} + t \cdot n^{1-1/d+\epsilon})$.

Circuit depth. We prove the circuit depth by induction on the height of T . For a tree T of height 1 the root has r leaf children. The circuit starts with r instances of *IsContaining* and r instances of *IsCrossing* in parallel whose depth is ℓ . Then the circuit has a *BuildSeletionMatrix* subcircuit whose depth is $O(\xi \log_r)$. Then the circuit has an instance of matrix multiplication whose depth is constant. The total depth is $\ell + O(\xi \cdot \log r)$.

Assuming the circuit depth of a tree of height $(d-1)$ is $(d-1)\ell + (d-1)O(\xi \cdot \log r)$ we prove for a tree of height d . For a tree of height $d > 1$ the circuit has r instances of *IsContaining* and r instances of *IsCrossing* in parallel. Then the algorithm has a *BuildSeletionMatrix* subcircuit followed by ξ subcircuits that compute range search queries on subtrees of height $(d-1)$. This yields a circuit depth of $d \cdot \ell + dO(\xi \log r) = O(\ell \cdot \log n)$. Since the tree height is $d = \log_{r/h} n$ and ξ is a parameter that depends on r . \square

C Hiding tree structure

In this section we describe Algorithm 4 which we call *FillTree* that adds empty nodes to an input tree T as mentioned in Section 7.3.

In Figure 6 we show an example of a partition tree and its full version after empty nodes have been added to it.

Algorithm 4: $FillTree(n, r, h, T')$

Input: Number of points $n = |S_{root}|$; the parameters $0 < r < n$;
 $0 < h$ as specified in the partition theorem; A tree T' .

Output: A full tree T where all inner nodes have r children
and all leaves have distance $\lceil \log_{r/h} n \rceil$ from the root.

- 1 **while** *There are nodes with less than r children or leaves whose distance from the root is less than $\lceil \log_{r/h} n \rceil$* **do**
 - 2 Add an empty child node to an inner node that has less than r children.
 - 3 Add r empty children nodes to leaves whose distance from the root is less than $\lceil \log_{r/h} n \rceil$.
-

Lemma 4. *Let P be a set of n points in \mathbb{R}^d , Γ a family of ranges, $r < n$ a parameter and h a parameter such that any simplicial partition of P' with respect to Γ , $\Pi = \{(P'_1, \sigma_1), \dots, (P'_m, \sigma_m)\}$ satisfies $|P'|/r < |P'_i| < h \cdot |P'|/r$ and let $T = FillTree(T', n, r, h)$, where T' is a partition tree built for P and Γ , then the height of T is $\lceil \log_{r/h} n \rceil$ and it has a total of $n^{\frac{1}{1-\lceil \log_{r/h} n \rceil}} = O(n^{1+\epsilon})$ nodes.*

Proof. From the partition theorem, at each node v we have a partition with $n_v/r \leq |P'_i| \leq h \cdot n_v/r$. It follows that the number of children at each node is at most r and the height of the tree is at most $\lceil \log_{r/h} n \rceil$. The number of nodes is therefore $r^{\lceil \log_{r/h} n \rceil} = r^{\lceil \log_r n^{\frac{1}{1-\lceil \log_{r/h} n \rceil}} \rceil} = O(n^{\frac{1}{1-\lceil \log_{r/h} n \rceil}}) = O(n^{1+\epsilon})$. \square

We conclude with a Lemma stating that the structure of a full tree does not leak information on P .

Lemma 5. *Let $P_1, P_2 \subset \mathbb{R}^d$ be 2 sets of points with $|P_1| = |P_2| = n$ and T'_1, T'_2 be 2 partition trees built for P_1 and P_2 , respectively, with the same parameters r, h then T_1 and T_2 have the same structure, where $T_i = FillTree(n, r, h, T'_i)$.*

Proof. The number of children in each node of T'_1 and T'_2 is at most r for both trees and does not depend on P . In addition, the height of T'_1 and T'_2 is at most $\lceil \log_{r/h} n \rceil$. Since $FillTree$ adds nodes to have a full tree of height $\lceil \log_{r/h} n \rceil$ where each inner node has exactly r children T_1 and T_2 have the same structure. \square

D Security Analysis

In this section we prove the privacy against the cloud (there is no need to prove privacy against the data owner or the query owner because we assume they collude).

Theorem 3. *Algorithm 2 is secure against a computationally bounded, semi-honest cloud.*

Before we prove this theorem we define the view of the cloud, i.e. the set of all messages it sees during the execution of the protocol. The view of the cloud is

$$\text{view}_C = (pk, d, n, \Gamma, h, \xi, r, \llbracket \gamma \rrbracket, \llbracket T \rrbracket, \llbracket f(\gamma \cap P) \rrbracket, \llbracket I_1 \rrbracket, \llbracket I_2 \rrbracket, \dots),$$

where pk is the public key as was received from the query owner, d, n, Γ, h, r and ξ are the parameters the partition tree was built with and were received from the data owner, $\llbracket \gamma \rrbracket$ and $\llbracket T \rrbracket$ which is the partition tree whose structure is given but the content in its nodes: $\llbracket v.f \rrbracket$ and $\llbracket v.\sigma \rrbracket$, for every node $v \in T$, is encrypted. In addition, the view includes $\llbracket f(\gamma \cap P) \rrbracket$ and all the intermediate values $(\llbracket I_1 \rrbracket, \llbracket I_2 \rrbracket, \dots)$ that are generated during the execution of Algorithm 2. For simplicity, we reorder

$$\text{view}_C = (pk, d, n, \Gamma, h, \xi, r, \text{structure of } T, \llbracket m_1 \rrbracket, \llbracket m_2 \rrbracket, \dots),$$

where $\llbracket m_1 \rrbracket, \dots$ are the ciphertexts in its view.

Before proving Theorem 3 we prove a lemma claiming that a more restricted view (one that does not include the structure of T)

$$\text{view}_C^{restr} = (pk, d, n, \Gamma, h, \xi, r, \llbracket m_1 \rrbracket, \llbracket m_2 \rrbracket, \dots).$$

is computationally indistinguishable from this view:

$$\text{view}_Z = (pk, d, n, \Gamma, h, \xi, r, \llbracket 0 \rrbracket, \llbracket 0 \rrbracket, \dots).$$

Lemma 6. view_C^{restr} is computationally indistinguishable from view_Z .

Proof. Consider this set of views:

$$\begin{aligned} \text{view}_C^{restr} &= \text{view}_C^0 = (pk, d, n, \Gamma, h, \xi, r, \llbracket m_1 \rrbracket, \llbracket m_2 \rrbracket, \dots) \\ \text{view}_C^1 &= (pk, d, n, \Gamma, h, \xi, r, \llbracket 0 \rrbracket, \llbracket m_2 \rrbracket, \dots) \\ \text{view}_C^2 &= (pk, d, n, \Gamma, h, \xi, r, \llbracket 0 \rrbracket, \llbracket 0 \rrbracket, \dots) \\ &\vdots \\ \text{view}_Z &= (pk, d, n, \Gamma, h, \xi, r, \llbracket 0 \rrbracket, \llbracket 0 \rrbracket, \dots), \end{aligned}$$

where $\text{view}_C^{(i)}$ is different from $\text{view}_C^{(i-1)}$ by replacing $\llbracket m_i \rrbracket$ with $\llbracket 0 \rrbracket$, for $i = 1, 2, \dots$. Then if view_C^{restr} and view_Z are distinguishable then there exists some i for which view_C^i is distinguishable from $\text{view}_C^{(i-1)}$, but this means that $\llbracket m_i \rrbracket$ is distinguishable from $\llbracket 0 \rrbracket$ without having sk , which is a contradiction to the semantic security of FHE. \square

The proof of Theorem 3 is now easy.

Proof of Theorem 3. To prove against a semi-honest computationally bounded cloud we construct a simulator \mathcal{S} whose output, when given only the public parameters $(pk, d, n, \Gamma, h, \xi, r)$ is computationally indistinguishable from an adversarial cloud's view in the protocol.

The simulator operates as follows: (1) generates a dummy set $P' \subset \mathbb{R}^d$ of n points; (2) build a partition tree τ' for P' with parameters n, ξ, r, h and applies $\tau := \text{FillTree}(n, r, h, \tau')$; (3) encrypts γ and $v.f$ and $v.\sigma$ for every node $v \in \tau$; (3) executes Algorithm 2 $PPRangeSearch_{n,d,\Gamma,\xi,r,h}(\llbracket \tau \rrbracket, \llbracket \gamma \rrbracket)$; (4) outputs the view of the simulator

$$\text{view}_{\mathcal{S}} = (pk, d, n, \Gamma, h, \xi, r, \text{structure of } \tau, \llbracket m_1 \rrbracket, \llbracket m_2 \rrbracket, \dots)$$

By Lemma 5 T and τ have the same structure. By Lemma 6 the restricted simulator view (without the structure of τ) $\text{view}_{\mathcal{S}}^{restr}$ is computationally indistinguishable from view_Z which is indistinguishable from view_C^{restr} . We conclude that the simulator's view is computationally indistinguishable from the cloud's view. \square

E Applications

In this section we give a few applications for our privacy preserving range searching.

Counting. Counting is the problem of computing $|P \cap \gamma|$, i.e. how many points of P are in γ . For this we set $f : 2^P \rightarrow \mathbb{N}$ be defined as $f(A) = |A|$ and $g : \mathbb{N} \times \mathbb{N} \rightarrow \mathbb{N}$ be defined as $g(a, b) = a + b$.

Reporting. Reporting is the problem of outputting the points in $P \cap \gamma$. Here we do not report the points explicitly. Instead we report $O(\log n)$ canonical subsets S_{v_1}, \dots, S_{v_m} such that $\cup_i S_{v_i} = P \cap \gamma$. As hinted, the canonical subsets are going to be the sets associated with nodes in the partition tree (more specifically, the nodes whose simplex is contained in γ and their father's simplex is not) and to report them we assign an id to each node and output the id of the node. For this we set $f : 2^P \rightarrow 2^{\mathbb{N}}$, that is, f maps a set $A \subset P$ into the set ids of canonical subsets whose union is A . f is defined as $f(S_v) = ID(S_v)$, where $ID(\cdot)$ is a function returning a unique id for each subset S_v associated with a node. Similarly g is set to be $g : 2^{\mathbb{N}} \times 2^{\mathbb{N}} \rightarrow 2^{\mathbb{N}}$ and is defined as $g(A, B) = A \cup B$.

We also note that techniques such as [4] can be used to efficiently output $P \cap \gamma$ if $|P \cap \gamma|$ is small.

Min. The *min* problem is to report $\min_{p \in P \cap \gamma} (\text{cost}(p))$, where $\text{cost} : P \rightarrow \mathbb{R}$ is some cost function. To report minimum we set $f : 2^P \rightarrow \mathbb{R}$ and define $f(A) = \min_{p \in A} (\text{cost}(p))$ and $g(a, b) = \min(a, b)$.

We note that computing minimum under FHE is a costly operation to compute. Using a circuit that realizes a partition tree using copy-and-recurse

method instantiates only $O(\log n)$ instances of the subcircuit implementing the minimum operation, as oppose to $O(n)$ instances using the naive way.

Averages and k -Means Clustering. The average of a set A is $Avg(A) = \frac{\sum_{p \in A} p}{|A|}$. Since division is costly under FHE, it is customary to return the pair $(\sum_A p, |A|)$. We set $f : 2^P \rightarrow P \times \mathbb{R}$ and define $f(A) = (Sum_A, Size_A)$, where $Sum_A = \sum_A a$ and $Size_A = |A|$. The average then can be computed $Avg(A) = Sum_A/Size_A$.

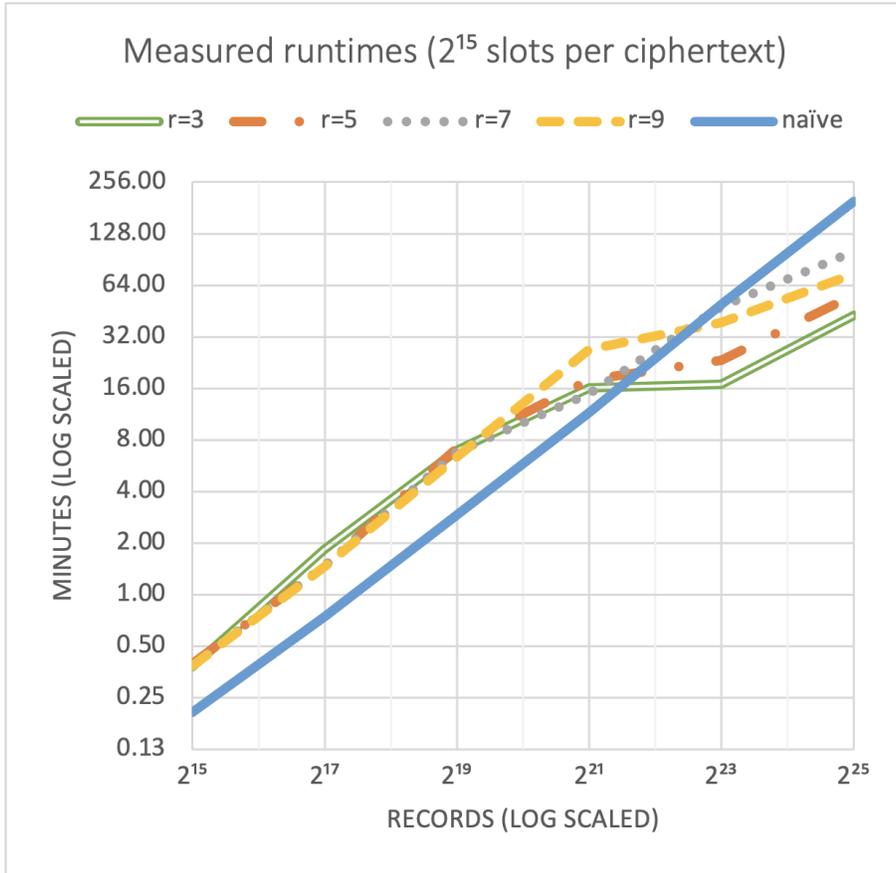


Figure 5: The running time of a count range search query, i.e., finding $|P \cap \gamma|$ where $P \subset \mathbb{R}$. The x -axis is the number size of P . The y -axis is the running time of the algorithm in minutes. The blue (solid) line shows the running time of the naive method. The other lines show the running time of *PPCount* using copy-and-recurse method using a partition tree with $r = 3, 5, 7, 9$, where r is the number of children of every inner node.

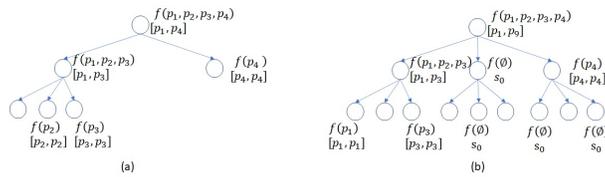


Figure 6: An example of a partition tree before (left) and after (right) applying *FillTree*. On the left is a partition tree for 4 points p_1, \dots, p_4 . On the right is a full version of the same tree. All the inner nodes need to have the same number of children, so empty nodes are added to the root. All leaves need to be at the same distance from the root, so empty nodes are added to the rightmost node (that represents p_4) and as children of the newly added child of the root. All empty nodes have the value $v.f = f(\emptyset)$ and empty bounding simplex s_0 .