Secure Range-Searching Using Copy-And-Recurse

Eyal Kushnir
IBM Research
Haifa, Israel
eyal.kushnir@ibm.com

Guy Moshkowich
IBM Research
Haifa, Israel
guy.moshkowich@ibm.com

Hayim Shaul
IBM Research
Haifa, Israel
hayim.shaul@ibm.com

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 of 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 queries. It had received a lot of attention in computational geometry where a data structure called *partition tree* was shown to solve range queries in time sub-linear in $|P|$ using space only linear in $|P|$.

In this paper we consider partition trees under FHE 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 $O(t \cdot n^{1-\frac{1}{d}+\epsilon} + n^{1+\epsilon})$ operations, where $n = |P|$, t is the number of operations needed to compare to γ and $\epsilon > 0$ is a parameter. When the ranges are axis-parallel hyper-boxes the running time is $O(t \cdot n^{\epsilon} + n \log^{d-1} n)$. As far as we know, this is the first non-trivial bound on range searching under FHE 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.

1 INTRODUCTION

The problem of *range queries* or *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 $f(P \cap \gamma)$. For example, a database with `sugar level`, `age` and `LDL level` columns, can be represented by $P \subset \mathbb{R}^2$, where a record with sugar level s, age a and LDL level l is represented by a point $(a, l) \in P$ (s is an additional datum associated with the point). Then, this query:

```
SELECT AVERAGE(sugar level) FROM patients
  WHERE (age > 30) and (age < 40)
    and (LDL level > 130) and (LDL level < 160);
```

can be answered by averaging the sugar levels associated with points in P that fall in the axis-parallel rectangle $\gamma = \{(x, y) \mid 30 < x < 40 \text{ and } 5 < y < 10\}$.

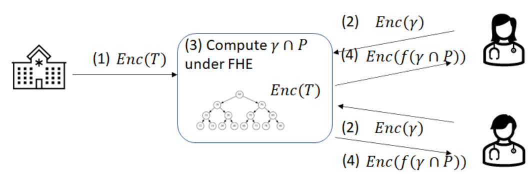

Figure 1: A FHE-based system for delegating statistics-as-a-service to an untrusted cloud. (1) The data owner (hospital) encrypts and uploads a database to an untrusted cloud. (2) A querier (doctor) encrypts their query and sends it to the untrusted cloud. (3) The cloud performs the query under FHE and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

The privacy preserving range searching problem is to compute $f(P \cap \gamma)$ without sharing the inputs (i.e. P or γ). This is useful if for example P is a database of sensitive data (e.g., medical, financial, etc.) that is kept at an untrusted cloud.

When the data owner and the querier collude but still use an untrusted cloud they can use a protocol based on oblivious RAM [34] or private information retrieval. In such solutions the querier needs to stay active (communicate and run computations) throughout the protocol. When the data owner and the querier do not collude multi-client ORAM [18, 31] can be used, but still the querier needs to stay active during the protocol. If some leakage is allowed, SSE [21–25] can be used. This hides the content of messages but the access pattern still leaks. It has been shown that this leakage is significant [26, 27, 30, 32].

In another line of solutions, which is more secure, the parties engage in a protocol that computes an arithmetic circuit to compute $f(P \cap \gamma)$ either interactively (e.g. using beaver triples [11]) or in a single round. The latter is done using Fully Homomorphic Encryption (FHE) which is a public key encryption where addition and multiplication can be applied on ciphertexts. Unlike SSE or ORAM, these solutions cannot compare 2 encrypted values and cannot make decisions based on comparisons. Intuitively, this means that these protocols cannot skip any point of P, leading to a $\Omega(n)$ lower bound. Although less efficient in CPU time, these solutions are more secure than SSE (unlike SSE, they do not leak the access pattern) and more efficient in network complexity than ORAM.

Proceedings on Privacy Enhancing Technologies YYYY(X), 1–19

https://doi.org/XXXXXXX.XXXXXXX

In this paper we focus on protocols that allow only Add and Mul to compute $f(P \cap \gamma)$. Specifically, we focus on FHE although the methods and techniques in this paper are generic.

As mentioned above, when using 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 comparisons and branching to skip entire groups of points. This is impossible under FHE where it is impossible to make a decision (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 converting a plaintext algorithm to run under FHE 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 and (3) multiplexing by multiplying one branch by $Enc(c)$ and the other branch by $(1 - Enc(c))$. This can be used to add p to the output only when $p \in \gamma$ and needs to be repeated for each $p \in P$ which is why solutions that work well in plaintext do not (naïvely) extend well to FHE. While the techniques described in this paper cannot "break" the $\Omega(n)$ lower bound, they apply the expensive test only sub-linear number of times. Our technique still "visits" every $p \in P$ but performs a cheaper operation on each point.

The ranges we consider in this paper are 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 plaintext, range-searching can be solved using linear storage with partition trees [33]. This tree-structure is built for a family of ranges Γ and a set of points $P \subset \mathbb{R}^d$. In the construction we partition P into m subsets $P_1, \ldots, P_m$ 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 $O(t \cdot n^{1-\frac{1}{d}+\epsilon})$, 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).

Naïvely implementing a partition tree under FHE leads to $O(t \cdot n)$ operations because, as mentioned earlier, the naïve approach ends up checking every path of the tree which is equivalent to checking whether $p \in \gamma$ for every $p \in P$. Our algorithm has improved time bound summarized in Table 1. In the general case it is $O(t \cdot n^{1-\frac{1}{d}+\epsilon} + n^{1+\epsilon})$[1]. Recall that $O(n)$ is a lower bound and $O(t \cdot n^{1-\frac{1}{d}+\epsilon})$ is the running time of the plaintext algorithm. Our improved time bound 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 uses selection matrices to generate a copy of ξ children and their subtrees (under FHE) and recurses into the copied subtrees. The time spent by the algorithm copying subtrees is linear in the tree size, however, the function that compares a range is used only a sublinear number of times. This leads to an algorithm that needs only $O(t \cdot n^{1-\frac{1}{d}+\epsilon} + n^{1+\epsilon})$ 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 [6] using HEaan [20] as the underlying cryptographic library implementing CKKS [15]. Our experiments show that our method is ×4.6 faster than the naïve implementation for a database of 32M records. We expect our algorithm to have impact in practice as well as in theory as databases often have significantly more records.

In Figure 1 we show a system that implements privacy preserving range searching. On the left is a data-owner (e.g. a hospital) wishing to provide analytics-as-a-service (AAAS) and charge customers per query. To save IT and maintenance costs they encrypt and upload their database to an untrusted (honest but curious) cloud. Such services are given, for example by IBM [3], CryptoLab [2] and Duality [1]. Paying customers can query the database to get various statistics by sending an encrypted query range to the cloud, receiving the encrypted result and decrypting it. Each party can use its own key and transcipher the database from the data-owner key to the client key [7].

1.1 Our Contribution

We list below our contributions in this paper.

- **The copy-and-recurse method.** Given a tree, T, and a recursive plaintext algorithm, A, that traverses T and computes a function f on nodes it visits, such that: (1) T is a full r-ary tree (i.e. each inner node has r children); (2) A continues in recursion on at most $\xi = r^c < r$ nodes, then we show how T can be traversed under FHE, such that f is applied only a sub-linear number of times $n^{c+\epsilon} < n$. We note that the FHE version of A has a linear overhead because it still needs to consider all the nodes in T. In the cases we focus on in this paper this overhead is less costly than applying f on all T.
  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 construct a selection matrix to generate a copy of the ξ children and their subtrees.
  - Recurse into the copies of ξ children.
- **An algorithm to answer privacy preserving range queries.** We show how to build a FHE-friendly partition tree that efficiently answers range searching queries (see [4, 5, 33]). 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 then uses

[1]In the special case where Γ is the set of axis-parallel hyper-boxes we show better times: $O(t \cdot n^{\epsilon} + n \log^{d-1} n)$

This work (FHE)	$d = 1$ (segments)	$O(t \cdot n^{\epsilon} + n)$
	$d > 1$ axis-parallel boxes	$O(t \cdot n^{\epsilon} + n \log^{d-1} n)$
	$d > 1$ general ranges	$O(t \cdot n^{1-1/d+\epsilon} + n^{1+\epsilon})$
Naive (FHE)		$O(t \cdot n)$

Table 1: A summary of our work to solve privacy preserving range searching under FHE. When $d = 1$ (and the ranges are segments), when the ranges are axis-parallel boxes and when the ranges are general semi-algebraic ranges, where n is the number of points in the database and $\epsilon > 0$ is arbitrarily small. Also, t is the complexity to compare a range to a point or a simplex. The last line shows the naïve FHE solution.

the copy-and-recurse technique and is implemented by an arithmetic circuit of size $O(t \cdot n^{1-\frac{1}{d}+\epsilon} + n^{1+\epsilon})$, where t is the size of an arithmetic circuit that compares to γ 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 6, 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 3) securely evaluates $f(\gamma \cap P)$ in a circuit whose size is $O(t \cdot n^{1-\frac{1}{d}+\epsilon} + n^{1+\epsilon})$ and depth is $O(\ell \cdot \log n)$.*

  Here and throughout the paper we use $2^{\mathbb{R}^d}$ to denote the power set of $\mathbb{R}^d$, i.e., the set of all subsets of $\mathbb{R}^d$. In the special case where Γ is the set of all axis-parallel hyperboxes in $\mathbb{R}^d$, the circuit size is only $O(t \cdot n^{\epsilon} + n \log^{d-1} n)$.
- **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 [6] with HEAAN [20] 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 (we defer the ones that relate to computational geometry 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

2.1 Range Searching

In the 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 [33] showed how to build a data structure called *partition tree* of $O(n)$ size where for any $\gamma \in \Gamma$, where Γ is the set of all halfspaces in $\mathbb{R}^d$ bounded by a hyperplane, $P \cap \gamma$ can be computed in $O(n^{1-1/d+\epsilon})$ time. Later, Agarwal and Matoušek [4] 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 $d \geq 4$ (for the case of $d \geq 4$ a later work by Vldalen [29] is needed). Using tools from algebraic geometry Agarwal, Matoušek and Sharir [5] 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 $i = 1, \ldots, n$ and report the (single) point in the range $\gamma_x = \{(a, b) \in \mathbb{R}^2 \mid x - 0.5 < a < x + 0.5\}$. In [12] a lower bound of $\Omega(n)$ was shown for the PIR problem, which holds for the privacy preserving range searching query as well.

In the privacy setting, the problem can be solved with different techniques as we now mention. See also Table 2.

2.2 Symmetric Searchable Encryption (SSE)

SSE schemes offer a trade off between efficiency and revealing some well defined information about queries and stored data. See for example [21–25]. SSE schemes are less secure than other schemes. Even a semi-honest adversary that follows the protocol learns significant knowledge, for example the access pattern. Indeed, several attacks have been proposed that use this leakage. See for example [26, 27, 30, 32]. Structured encryption (STE) is closely related to SSE. Using STE also leaks some well-defined leakage profile.

Unlike these solutions our solution is based on FHE which does not leak the access pattern (or the content of inputs and intermediate values). Specifically, our protocol "visits" every element in the database which hides the access pattern.

2.3 Oblivious-RAM (ORAM) and Private Information Retrieval (PIR)

When the data-owner and the querier collude they may use an ORAM-based scheme [34]. Such a scheme uses the cloud as an oblivious RAM, i.e., a memory bank that can be accessed for reading and writing in an oblivious manner. Oblivious here means the cloud does not learn the content of the database and also it does not learn

	ORAM PIR [34]	MCORAM [18, 31]	SSE [21–25]	FHE [17]	FHE This work
2-party		✓	✓	✓	✓
Passive querier			✓	✓	✓
Secure	✓	✓		✓	✓
Sublinear time	✓	✓	✓		
Sublinear cmp	✓	✓	✓		✓

Table 2: Comparing different cryptographic techniques for range searching. *2-party* means the data owner and the querier can be 2 different non-colluding parties. *Passive querier* means the querier does not need to stay online during the execution. *Secure* means there are no known attacks against this technique. *Sublinear time* means there are protocols that achieve sublinear running time. *Sublinear cmp* means the protocols call the compare function a sublinear number of times.

which element in the database was accessed (thus hiding the access pattern). The basic recipe is to keep the database in the cloud using ORAM. The querier then searches the database by locally executing the plaintext algorithm where a local access to the database is replaced with a remote ORAM access. For example, to traverse a binary search tree, the querier obliviously reads the root, locally decides on the child it should continue with (left or right) and obliviously reads that child node and so on. When the data owner and the queriers do not collude multi-client ORAM [18, 31] can be used.

Since the protocol makes only read operations it can also be implemented using a series of PIR queries, each obliviously reading an element stored on the cloud. In both cases (ORAM or PIR) the querier needs to be active throughout the protocol. As the querier follows the protocol they read and learn the content of (at least part of) the database.

Unlike these solutions, with FHE the processing is done by the cloud and the querier is passive, i.e., they only send the input and receive the output. Also, with FHE the data-owner can encrypt the database once (e.g. using AES) and use transciphering to re-crypt the database with the querier's key (see [7] for example).

2.4 Fully Homomorphic Encryption

As mentioned in the introduction, we are interested in a solution based on FHE. FHE has better privacy than SSE. Specifically, nothing on the content of the inputs, intermediate values, access pattern or output leaks to the cloud. In addition, unlike ORAM, FHE has efficient communication complexity. This comes at a cost of being less efficient in CPU. The main difference from SSE and ORAM/PIR solution is FHE cannot make decisions based on the content of intermediate values. For example, when traversing a search tree the protocol cannot follow a single path. Instead it visits all the nodes in the tree which makes the search tree inefficient. Indeed, previous solutions [17] compared all data records to the query range to compute the output.

	[10]	[9]	[19]	copy-and-recurse
Encoding agnostic	✓	✓		✓
Sublinear	✓		$\cancel{\checkmark}$	✓
Encrypted	✓	✓		✓
Range searching				✓

Table 3: Comparing our copy-and-recurse method to other works under FHE. *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.

Although we also go over all data records (which is unavoidable under FHE) the function used for comparison (which dominates the running time) is called only a sublinear number of times.

2.5 Traversing a Tree Under FHE

One of our contributions is the copy-and-recurse method which efficiently traverses a tree under FHE. A recent work by Azogagh et al. [10] 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 retrieve the variable and the parameter from an array of variables and parameters. Here, the index of the node that the traversing reaches is encrypted and used to fetch the right variable and parameter. However, it is not clear how to extend their ideas to answering range searching queries. Another recent work by Cong et al. [19] discussed *homomorphic traversing* where they show how to efficiently traverse a decision tree. To use their techniques the values at each decision node need to be bit-wise encrypted i.e., each bit in its own ciphertext. The thresholds 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 still 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 [9] 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 [35] 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 [36] the authors considered a solution that uses garbled circuits and ORAM to achieve sub-linear prediction time with a decision tree, however their solution requires the querier to be active and collude with the data owner. We summarize these works in Table 3.

2.6 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 $h : \mathbb{Z}^m \to \mathbb{Z}$, Iliashenko et al. showed in [28] how to preprocess P s.t. given an encrypted value x, 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 $\gamma_x = \{p \in \mathbb{Z}^m \mid h(p) = x\}$. 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. [17] considered encrypted queries over an encrypted database. They propose a search-and-compute method, where they 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 matching a query they apply a *IsMatch* function on all 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.

Power set. For a set A, we use 2^A to denote the power set of A, i.e., the set of all subsets of A.

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 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. Intuitively, a selection matrix is a generalization of the *multiplexer* (Mux) operation that selects one of 2 inputs. More formally: $M \cdot x^T = (x_{i_1}, \ldots, x_{i_\xi})$, where $x = (x_1, \ldots, x_d)$. 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 $M_{r,c} = 0$. Algorithm 1 shows how to construct a selection matrix. An overview and analysis of this algorithm is given in Section A.

Trees. We use v to denote a node in a tree. We use dot (".") to denote members of v, so for example, $v.child[1], \ldots, v.child[r]$ 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.

Algorithm 1: $BuildSelectionMatrix_{r,\xi}(\llbracket c \rrbracket)$

Parameters: $\xi < r$.
Input: A vector $\llbracket c \rrbracket$, where $c \in \{0,1\}^r$, s.t. $\sum c_i \le \xi$.
Output: A selection matrix $\llbracket M \rrbracket$ that selects the non-zero elements in c.

```
for ( row = 1, ..., r )
    [[M[1, row]]] := [[c[row]]] · ∏_{i=1}^{row-1} (1 − [[c[i]]])
for ( col = 2, ..., ξ )
    for ( row = 1, ..., r )
        [[M[col, row]]] := [[c[row]]] ∑_{k=1}^{row-1} ( [[M[col − i, k]]] ∏_{h=k+1}^{row-1} (1 − [[c[h]]]) )
Output: [[M]]
```

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. [14, 15]) is an asymmetric encryption scheme that also supports + and × 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 $\llbracket d \rrbracket$.

Correctness is the requirement that $m = m'$, $c = a + b \mod p$ and $d = a \cdot b \mod p$. In an approximated FHE (e.g. CKKS [15]) we require that $m \approx m'$, $c \approx a + b$ and $d \approx a \cdot b$.

Semantic security is the requirement that given $pk, \llbracket m_1 \rrbracket, \ldots, \llbracket m_{poly(\lambda)} \rrbracket, m_1, \ldots, m_{poly(\lambda)}$, 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 $\mathbb{P}(x_1, \ldots)$ on the ciphertexts $\llbracket x_1 \rrbracket, \ldots$. 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, denoted size(C), which is the number of operators in C and with the depth of C, denoted depth(C), which is the maximal number of multiplication gates on a path of C. The time to evaluate a circuit is then $Time = overhead \cdot \text{size}(C)$, 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 $[\![\cdot]\!]_{pk}$ to denote a ciphertext. When pk is clear from the context we omit it. We use an abbreviated syntax:

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

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 contradicts 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).

There are works (e.g. [16]) for implementing a $IsSmaller([\![c_1]\!], [\![c_2]\!])$ function that returns a **ciphertext** whose message is the indicating bit (whether or not $c_1 < c_2$). In a nutshell, such works implement a bi-variate polynomial $IsSmaller(x, y) = \begin{cases} 1, & \text{if } x < y \\ 0, & \text{otherwise.} \end{cases}$ (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.*

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):

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

The d-dimensions version of these functions is similar except that it gets a simplex $\sigma \subset \mathbb{R}^d$ and a range $\gamma \subset \mathbb{R}^d$. See more details in Section 7. See also Figure 2.

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 sub-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. In addition, in approximate schemes the implementation of *IsContaining* and *IsCrossing* depends on $n = |P|$. These functions need to be more accurate when n grows since any error they incur is aggregated over more points. We therefore have $\ell = \ell(n)$ and $t = t(n)$. The exact details depend on the the implementation and the parameters of the cryptographic scheme. 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 which capture the hardness of *IsContaining* and *IsCrossing*.

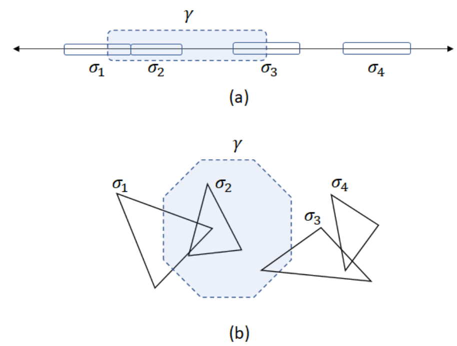

Figure 2: A range γ and simplices $\sigma_1, \sigma_2, \sigma_3$ and σ_4. Above (a) is a setting in 1-dim, where the range is a segment and simplices are non-intersecting segments. Below (b) is a setting in 2-dim, where the range is an octagon and the simplices are (possibly intersecting) triangles.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

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 $\mathbb{D}$ 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 $\mathbb{D}$ is constructed. See for example Figure 2 (bottom) for an example in 2-dim, where Γ is the set of all octangles.

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 concur manner.

In all cases our privacy guaranty is that $\mathbb{D}$ 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 parties: (1) data owner that owns P, (2) querier that performs range search queries and (3) the cloud to which the database and queries are uploaded to and performs the computation. This security model is motivated by the growing trend of providing analytics-as-a-service and outsourcing that service to the cloud (to save maintenance and other IT costs). See for example solutions by IBM [3], CryptoLab [2] and Duality [1]. In the example of Figure 1 the hospital is the data owner and doctors are the queriers.

Protocol 2 shows an overall view of a system implementing our secure range searching protocol.

Protocol 2: *RangeSearchingProtocol*

Parties: Data owner, Querier, 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.
Querier input: A pair (sk, pk) of secret and public keys.
Ranges $\gamma_1, \ldots, \gamma_c \in \Gamma$, where $c \in \mathbb{N}$ is polynomial in the security parameter of (sk, pk).
The cloud has no input.
Querier output: $|P \cap \gamma_i|$, for $i = 1, \ldots, c$.
The cloud and the data owner have no output.

1 **Querier 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 Γ with parameter r. `// See Section 6`
6 $T := FillTree(n, r, h, T')$. `// See Algorithm 6`
7 $[\![T]\!] :=$ encrypt $v.val, v.\sigma$ for every $v \in T$.
8 Send $[\![T]\!], n, \Gamma, \xi, r, h$ to Cloud.
9 **foreach** $i = 1, \ldots, c$ **do**
10 **Querier performs:**
11 $[\![\gamma_i]\!] := \mathrm{Enc}_{pk}(\gamma_i)$
12 Send $[\![\gamma_i]\!]$ to Cloud.
13 **Cloud performs:**
14 $[\![x_i]\!] := PPRangeSearch_{n,d,\Gamma,\xi,r,h}([\![T]\!], [\![\gamma_i]\!])$ `// See Algorithm 3`
15 Send $[\![x_i]\!]$ to querier.
16 **Querier performs:**
17 $x_i := \mathrm{Dec}_{sk}([\![x_i]\!])$
18 Output x_i

We consider computationally-bounded, semi-honest adversaries. We assume the querier doesn't collude with the cloud or the data owner. The semantic security of FHE guarantees that the cloud learns nothing on the content of γ or P.

4.2 Multiple Queriers

Protocol 2 considers a single querier. The protocol can be extended to multiple queriers without keeping an encrypted version of the database for each querier by using transciphering as we now explain. Suppose there are q queriers $Q_1, \ldots, Q_q$ where the key pair of Q_i is (sk_i, pk_i). In line 2 each querier sends their public key to the data owner. Prior to running Line 7 the data owner generates an AES key ak. In Line 7 the data owner encrypts T using ak. Denote this as $[\![T]\!]_{ak}$. In Line 8 the data owner additionally sends $[\![ak]\!]_{pk_i}$, i.e., ak encrypted using pk_i, for $i = 1, \ldots, q$. When answering a query sent by Q_i we make the following changes: before executing Line 14 the cloud uses $[\![ak]\!]_{pk_i}$ to decrypt $[\![T]\!]_{ak}$ under FHE. This transciphers $[\![T]\!]_{ak}$ into $[\![T]\!]_{pk_i}$. See for example [7] for an example for transciphering AES into CKKS. Then the cloud continues as usual with $[\![T]\!]_{pk_i}$ and returns the answer to Q_i.

The full algorithm is given in Appendix B.

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.

For example, full binary trees meet these prerequisites with $r = 2$ and $\xi = 1$ because we recurse into exactly one child. 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 a 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 $\chi = (\chi_1, \ldots, \chi_r)$, 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) Build a selection matrix to generate (under FHE) a copy of the children (and their subtrees) that we need to recurse into. From the prerequisites mentioned in Section 5.1 only copies of $\xi < r$ are generated since at most ξ children need to be recursed into. We stress that under FHE this is done my multiplying the vector of children by a $\xi \times r$ selection matrix and that takes time at least linear in the tree size.
(3) Recurse. After making copies of $\xi < r$ 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 ($r = 2, \xi = 1$) with n leaves. Applying copy-and-recurse at each node we determine which subtree to recurse into by computing the value of a comparison, cmp_i. Then multiplying by a selection matrix to copy that subtree. The figure depicts (the more familiar) Mux operation since a 1×2 selection matrix is identical to a Mux. We continue in recursion until we reach a leaf. Eventually, the decision function was called only $\log n$ times (computing cmp_1, cmp_2 and cmp_3).

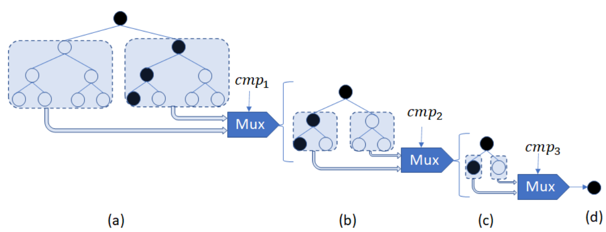

Figure 3: An example of copy-and-recurse method used with a full binary tree with $n = 8$ leaves. The black nodes are the nodes the plaintext algorithm visits as it traverses the tree. First (a) the FHE algorithm visits the root of the tree. Based on a comparison whose output is $cmp_1 \in \{0, 1\}$ a copy of the right child and its subtree is generated by multiplexing both children ($Mux(l, r, cmp_1) = cmp_1 \cdot r + (1 - cmp_1) \cdot l$). Multiplexing is equivalent to a 1×2 selection matrix. Next, (b) the algorithm continues in the copied subtree. Again it generates a copy of the left child and its subtree using Mux. Similarly that happens at (c). In (d) the FHE algorithm reaches a leaf and reports it. Using copy-and-recurse, the expensive decision function was called $\log n$ times, as oppose to the naïve FHE algorithm that calls the decision function $O(n)$ 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 diminish exponentially. Consider an algorithm that in plaintext traverses a tree and performs some additional work at each node it visits (e.g. a comparison). It can be migrated to run under FHE using copy-and-recurse, such that the additional work is 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 multiplying by a selection matrix.

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, $p_1 \leq \ldots \leq p_n \in \mathbb{R}$, 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 [33]) 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) $child[1], \ldots, child[r]$ - the children nodes.
- $[\![|S_v|]\!]$ - 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 $[\![|S_v|]\!]$ redundant, but we still mention it here because this is changed in Sections 7 and 9.
- $[\![\sigma]\!] = [\![[\sigma_{min}, \sigma_{max}]]\!]$ - i.e., the encryption of the segment defined by $\sigma_{min} = \min S_v$ and $\sigma_{max} = \max S_v$. We call the segment $[\sigma_{min}, \sigma_{max}]$ the *bounding segment* of S_v because $S_v \subset [\sigma_{min}, \sigma_{max}]$.

6.2 Building a Partition Tree

We now describe how to build a partition tree. Recall that we are given $p_1 \leq \ldots \leq p_n \in \mathbb{R}$ 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 $v.val = |S_v|$.
- Set $[\![v.\sigma]\!] = [\![[\min S_v, \max S_v]]\!]$.
- Partition S_v into r subsets $P_1 = (p_1, \ldots, p_{n/r})$, $P_2 = (p_{n/r+1}, \ldots, p_{2n/r}), \ldots, P_r = (p_{n-r+1}, \ldots, p_n)$.
- 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 to compute $|P \cap \gamma|$.

As we traverse the tree we keep a counter $count$ (encrypted) that will eventually hold $|P \cap \gamma|$. At the beginning we set $count = 0$. When at a node v we consider each child $u_i = v.child[i]$, compare its bounding segment, $[\![u_i.\sigma]\!]$, 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 $u_i.val = |S_{u_i}|$ to $count$ without recursing into the subtree of u_i.
(2) **disjoint.** If $u.\sigma$ 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 $u.\sigma$ 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 $\sigma_{min_1} \le \sigma_{max_1} \le \dots \le \sigma_{min_r} \le \sigma_{max_r}$. 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 $\gamma = [4, 7]$.

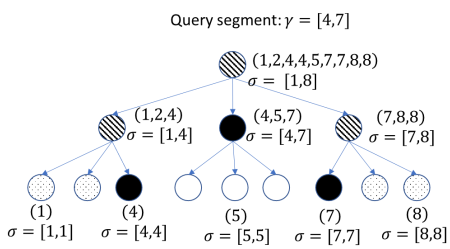

Figure 4: An example of a 1-dim partition tree that was built for the numbers 1,2,4,4,5,7,7,8,8 with $r = 3$. At each node v we show S_v and the bounding segment of S_v (for readability we omit this for some leaves). The figure also shows how the tree is traversed for counting all values in a query segment $\gamma = [4, 7]$. Nodes with $v.\sigma \subset \gamma$ are marked with solid black. All values in these nodes can be counted without recursing. Nodes with $v.\sigma \cap \gamma = \phi$ are marked with black dots. All values in these nodes can be ignored. Nodes the traversal doesn't reach (because an ancestor is one of the cases above) are marked with white. 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 Algorithm 3 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 3 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.

The **input** of the algorithm is $[\![T]\!]$ and $[\![\gamma]\!]$. 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 ($v.\sigma$ and $v.val$) are encrypted.

Algorithm 3: $PPRangeSearch_{n,r}([\![T]\!], [\![\gamma]\!])$

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 $[\![\gamma]\!]$
Output: $[\![x]\!]$, where $x = |P \cap \gamma|$.

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

We note that the structure of T (which node is the child of which) is not encrypted.

The **output** of the algorithm is $[\![x]\!]$ where $x = |P \cap \gamma|$.

Algorithm 3 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 input being $child'[i]$, a copy of a subtree of a child as generated by multiplying the selection matrix. The recursion stops when it reaches a leaf. While traversing the partition tree the algorithm sums $[\![v.val]\!]$ from various nodes (as we explain below). The improved efficiency of the algorithm comes from the property (proved in Lemma 1) that at most $\xi = 2$ children need to be recursed into. This is done under *FHE* by multiplying the vector of r children by a $\xi \times r$ selection matrix to get a copy of ξ children and their subtrees. Lemma 1 proves that the children we actually need to recurse into are among the 2 children we copy.

Algorithm 3 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 $[\![c]\!]$, where $c = 1$ if $v.\sigma \subseteq \gamma$ and 0 otherwise. Then we use $[\![c]\!]$ 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, is contained in γ (Line 6). If $S_{v.child[i]} \subset v.child[i].\sigma \subset \gamma$ we can count $v.val = |S_{v.child[i]}|$ 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 1). Then the algorithm computes $M \cdot v.child$ (Line 11) to generate a copy of ξ children (here we regard $v.child$ as a vector $(v.child[1], \ldots, v.child[r])$ with r elements 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 d dimensional points. 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**[2] $v.\sigma$ that contains all the points in S_v.

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 3 to support multiple dimensions.

7.1 Computational Geometry Preliminaries

In this section we explain some terminology in computational geometry.

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

Partition Theorem. In a seminal work [33] Matoušek showed a non-trivial partition with small crossing number when Γ is the set of halfspaces bounded by hyperplanes. In [4] it was extended to general semi-algebraic ranges with constant description complexity. The most recent partition theorem is due to [5] who improved the bound on the crossing number. Their result is summarized in the following theorem.

Theorem 2 (From [5]). *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 $\Pi = \{(P_1, \sigma_1), \ldots, (P_m, \sigma_m)\}$ 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 3 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 $\lfloor \log_r n \rfloor \leq height \leq \lfloor \log_{r/h} n \rfloor$. This follows from the subset sizes at node v being $\lceil |S_v|/r \rceil \leq m \leq h\lceil |S_v|/r \rceil$.

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 PPRangeSearch. The changes to *PPRangeSearch* include:

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

- The parameters include d, Γ, *IsContaining* and *IsCrossing* (more precisely, only *IsContaining* and *IsCrossing* are needed but their implementation depends on d and Γ).
- Call *BuildSelectionMatrix* 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 has 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 leaves are 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 D.

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

Lemma 6. *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, $\Pi = \{(P'_1, \sigma_1), \ldots, (P'_m, \sigma_m)\}$ 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 $n^{\frac{1}{1-\lceil \log_r h \rceil}} = O(n^{1+\epsilon})$ nodes.*

Lemma 7. *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 $T_i = FillTree(n, r, h, T'_i)$, for $i = 1, 2$.*

8 SIZE AND DEPTH ANALYSIS

In this section we analyze the size and depth of a circuit that implements *PPConut* (Algorithm 3) to compute $|P \cap \gamma|$. We start by analyzing the size and depth of *BuildSelectionMatrix* which is used by *PPRangeSearch*.

8.1 Analyzing *BuildSelectionMatrix*

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

Lemma 2. *Computing $M[col, row]$ for $1 \le col \le \xi$ and $1 \le row \le r$ 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 $1 \le row \le r$ 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 $1 \le row \le r$ 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. □

8.2 Analyzing *PPRangeSearch*

We now turn to analyze *PPRangeSearch* (Algorithm 3). 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 6, 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 6 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. □

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 6, 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 3) securely evaluates $f(\gamma \cap P)$ in a circuit whose size is $O(t \cdot n^{1-\frac{1}{d}+\epsilon} + n^{1+\epsilon})$ and depth is $O(\ell \cdot \log n)$.*

Informally, the correctness follows from the plaintext algorithm that Algorithm 3 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 C.

8.3 Axis-Parallel Hyperboxes

The case where $\Gamma \subset \mathbb{R}^d$ is the set of axis-parallel hyper-boxes can be solved more efficiently. Then $\gamma \in \Gamma$ is a conjunction of d 1-dimensional ranges (the projections of γ on each axis). In this case, we follow a standard method (see for example Section 16.2 in [13]) to answer range queries. In a nutshell, we build a 1-dim partition tree T^1 for the projection of P on the first axis. At each node v of T^1 (that represents S_v) we build a secondary 1-dim partition tree T^2_v for the point of S_v projected onto the second axis, and so on. To analyze the storage complexity we start at the last level T^d. Each tree there has space that is linear in the number of points stored in it. A tree T^i, for $d = 1, \ldots, (d-1)$, keeps a tree $T^{i+1}_{v_i}$, at every node $v_i \in T^i$. It is easy to prove by induction that the total size of a multi-level data structure with d levels is $O(n \log^{d-1} n)$ which leads to a circuit size of $O(t \cdot n^{\epsilon} + n \log^{d-1})$. We give a more detailed explanation in Section C.1.

8.4 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 querier who also holds the secret key and gets the output. We assume the querier does not collude with the cloud and claim the cloud does not learn anything on the context of the encrypted input it receives. The view of the querier includes only its input and output. The view of the data owner includes only its input. 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 E.

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 *PPRangeSearch* 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 input[3] 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 *PPRangeSearch*. In Appendix F we give a few useful applications that use different functions f and g.

Changes needed to be made on PPRangeSearch:

- Set $v.val = f(S_v)$, i.e. each node in the partition tree keeps the value of f applied on S_v.
- The output when v is a leaf (Line 3) is $[\![Cont]\!] \cdot [\![v.val]\!] + (1 - [\![Cont]\!]) \cdot f(\emptyset)$, i.e. return $v.val$ if $Cont = 1$ and $f(\emptyset)$ otherwise.
- Adding the contribution of children that are contained in γ should use g and be:

  foreach $i = 1, \ldots, r$ **do**
  $$[\![x]\!] := [\![Cont[i]]\!] \cdot g([\![x]\!], [\![v.child[i].val]\!], [\![\gamma]\!]) + (1 - [\![Cont[i]]\!]) \cdot [\![x]\!]$$
- Adding the contribution of children that cross γ should use g and be:

  foreach $i = 1, \ldots, \xi$ **do**
  $$[\![x]\!] := [\![Cross[i]]\!] \cdot g([\![x]\!], PPRangeSearch(child'[i], \gamma)) + (1 - [\![Cross'[i]]\!]) \cdot [\![x]\!]$$

In Appendix B we give the complete algorithm. In Appendix F we give a few useful applications for privacy preserving range searching.

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 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 [15], which is a popular scheme. To implement the *IsSmaller* function we used the work of [16].

We used CKKS when implementing both the naïve and our algorithms. We used the HELayers framework [6] which can work with any cryptographic library. In our experiments we used the HEaaN library [20] as the implementation for CKKS. We set the parameters of the keys to have 128 bits of security with $slots = 2^{15}$ 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. [17] 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 $O(n/slots)$ checks are actually needed. In our experiments, checking whether $p \in \gamma$ was made by comparing $p \in P$ to the 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 4 and in Figure 5. Table 4 shows the running time (reported in minutes) of *PPRangeSearch* 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 *PPRangeSearch* 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 different size. For example, the 4th row shows the case where $|P| = 2^{21}$. These results also

[3] we avoid the more acceptable terms "domain" and "range" to avoid confusion.

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 $|P| \geq 2^{23}$. 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 $O(t \cdot n)$. 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 *PPRangeSearch* using 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 $O(r \cdot t \cdot n^{\log_r 2} + r \cdot n) = O(t \cdot n^{\epsilon} + n)$. Indeed, when $n < t^{1+\frac{\epsilon}{1-\epsilon}}$ the term tn^{ϵ} is dominant. It becomes less dominant as n grows. That is why we see sub-linear growth when $|P| > 2^{21}$. For a fixed n, the choice of r affects the running time and our algorithm is more efficient than the naïve when $r \cdot t \cdot n^{\log_r 2} + r \cdot n < t \cdot n$. For large n that is when $r < \frac{t}{\text{time to copy a node of } T}$. The choice of r in higher dimension is more complicated and is referred to in Section C. The results for are also summarized in Figure 5.

$\|P\|$	$r=3$	$r=5$	$r=7$	$r=9$	Naïve
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 4: 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 is the size of the database, $|P|$. The next 4 columns are the running time of *PPRangeSearch* using copy-and-recurse with $r = 3, 5, 7, 9$. The last column is the running time of the naïve method.

10.2 Results in Higher Dimensions

We also repeated our experiment in 2 dimensions where Γ was the set of all axis-parallel rectangles. In this case we used the multi-level data structure described in Section 8.3. The results are summarized in Table 5. 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.

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.

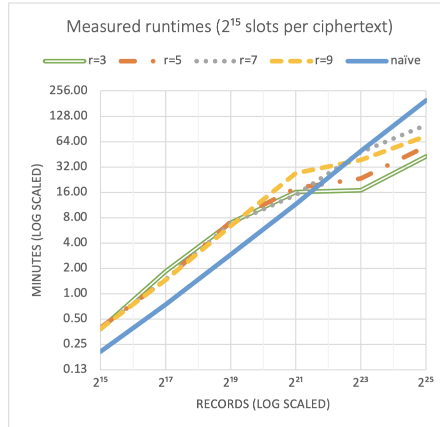

Figure 5: The running time of range counting, i.e., finding $|P \cap \gamma|$ where $P \subset \mathbb{R}$. The x-axis is the size of P. The y-axis is the running time of the algorithm in minutes. The blue (solid) line shows the running time of the naive method. The other lines show the running time of *PPRangeSearch* 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.

$\|P\|$	2^{17}	2^{19}	2^{21}	2^{23}	2^{25}
Naïve	3	14	54	213	910
$r = 4$	13	64	161	335	700

Table 5: 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 row is the size of P. The second row is the running time of *PPRangeSearch* using copy-and-recurse with $r = 4$. The last row is the running time of the naïve method.

We showed how to efficiently implement partition trees under FHE with this method. Partition trees are useful for solving range query problems (note the proliferation of plaintext problems 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.

We showed how to solve the general case of semi-algebraic ranges where more complex ranges yield a higher running time. We also showed the special case of axis-parallel hyper-boxes that can be solved more efficiently.

The efficiency of our results 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 adds to the $O(n^{1+\epsilon})$ overhead of copy-and-recurse). 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] Duality query engine. https://dualitytech.com/platform/duality-query/.

[2] Heaan homomorphic analytics. https://www.ncloud.com/product/analytics/heaanHomomorphic.

[3] Ibm z solutions. https://www.ibm.com/support/z-content-solutions/fully-homomorphic-encryption/.

[4] Pankaj K. Agarwal and Jiří Matoušek. On range searching with semialgebraic sets. *DISCRETE COMPUT. GEOM*, 11:393–418, 1994.

[5] Pankaj K. Agarwal, Jiří Matoušek, and Micha Sharir. On range searching with semialgebraic sets. ii. *SIAM Journal on Computing*, 42(6):2039–2062, 2013.

[6] Ehud Aharoni, Allon Adir, Moran Baruch, Nir Drucker, Gilad Ezov, Ariel Farkash, Lev Greenberg, Ramy Masalha, Guy Moshkowich, Dov Murik, Hayim Shaul, and Omri Soceanu. HeLayers: A Tile Tensors Framework for Large Neural Networks on Encrypted Data. *CoRR*, abs/2011.0, 2020.

[7] Ehud Aharoni, Nir Drucker, Gilad Ezov, Eyal Kushnir, Hayim Shaul, and Omri Soceanu. E2E near-standard hybrid encryption, March 2023. Poster session at 6th HomomorphicEncryption.org Standards Meeting, Seoul, South korea.

[8] 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.

[9] Adi Akavia, Max Leibovich, Yehezkel S Resheff, Roey Ron, Moni Shahar, and Margarita Vald. Privacy-preserving decision trees training and prediction. *ACM Transactions on Privacy and Security*, 25(3):1–30, 2022.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] 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.

[18] Sherman S. M. Chow, Katharina Fech, Russell W. F. Lai, and Giulio Malavolta. Multi-client oblivious ram with poly-logarithmic communication. In Shiho Moriai and Huaxiong Wang, editors, *Advances in Cryptology – ASIACRYPT 2020*, pages 160–190, Cham, 2020. Springer International Publishing.

[19] 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.

[20] CryptoLab. HEaaN: Homomorphic Encryption for Arithmetic of Approximate Numbers, version 3.1.4, 2022.

[21] Reza Curtmola, Juan Garay, Seny Kamara, and Rafail Ostrovsky. Searchable symmetric encryption: Improved definitions and efficient constructions. In *Proceedings of the 13th ACM Conference on Computer and Communications Security*, CCS '06, page 79–88, New York, NY, USA, 2006. Association for Computing Machinery.

[22] 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.

[23] 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.

[24] Sky Faber, Stanislaw Jarecki, Hugo Krawczyk, Quan Nguyen, Marcel Rosu, and Michael Steiner. Rich queries on encrypted data: Beyond exact matches. In Günther Pernul, Peter Y A Ryan, and Edgar Weippl, editors, *Computer Security – ESORICS 2015*, pages 123–145, Cham, 2015. Springer International Publishing.

[25] Francesca Falzon, Evangelia Anna Markatou, Zachary Espiritu, and Roberto Tamassia. Range search over encrypted multi-attribute data. *Cryptology ePrint Archive*, 2022.

[26] Paul Grubbs, Marie-Sarah Lacharite, Brice Minaud, and Kenneth G. Paterson. Pump up the volume: Practical database reconstruction from volume leakage on range queries. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security*, CCS '18, page 315–331, New York, NY, USA, 2018. Association for Computing Machinery.

[27] Zichen Gui, Oliver Johnson, and Bogdan Warinschi. Encrypted databases: New volume attacks against range queries. In *Proceedings of the 2019 ACM SIGSAC Conference on Computer and Communications Security*, CCS '19, page 361–378, New York, NY, USA, 2019. Association for Computing Machinery.

[28] Ilia Iliashenko, Malika Izabachene, Axel Mertens, and Hilder V L Pereira. Homomorphically counting elements with the same property. page 20.

[29] V. Koltun. Almost tight upper bounds for vertical decompositions in four dimensions. In *Proceedings 42nd IEEE Symposium on Foundations of Computer Science*, pages 56–65, 2001.

[30] Marie-Sarah Lacharité, Brice Minaud, and Kenneth G. Paterson. Improved reconstruction attacks on encrypted data using range query leakage. In *2018 IEEE Symposium on Security and Privacy (SP)*, pages 297–314, 2018.

[31] Matteo Maffei, Giulio Malavolta, Manuel Reinert, and Dominique Schröder. Privacy and access control for outsourced personal records. In *2015 IEEE Symposium on Security and Privacy*, pages 341–358, 2015.

[32] Evangelia Anna Markatou, Francesca Falzon, Zachary Espiritu, and Roberto Tamassia. Attacks on Encrypted Response-Hiding Range Search Schemes in Multiple Dimensions. *Proceedings on Privacy Enhancing Technologies*, 2023(4):204–223, 2023.

[33] 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.

[34] Gamze Tillem, Ömer Mert Candan, Erkay Savaş, and Kamer Kaya. Hiding access patterns in range queries using private information retrieval and oram. In Jeremy Clark, Sarah Meiklejohn, Peter Y.A. Ryan, Dan Wallach, Michael Brenner, and Kurt Rohloff, editors, *Financial Cryptography and Data Security*, pages 253–270, Berlin, Heidelberg, 2016. Springer Berlin Heidelberg.

[35] 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.

[36] 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 and analyze Algorithm 1 implementing $BuildSelectionMatrix_{r,\xi}(\llbracket c \rrbracket)$, a function that generates the selection matrix, M, used in Algorithm 3. 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 $BuildSelectionMatrix_{r,\xi}(\llbracket c \rrbracket)$ is a matrix $M \in \{0,1\}^{\xi \times r}$, such that for any vector $x \in \mathbb{R}^r$ we have

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

To understand how Algorithm 1 works we note that $M[i,j] = 1$ iff $c[i]$ is the j-th cell with a value of 1. Algorithm 1 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 $1 \le k < i$. Then, Algorithm 1 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 $(j-1)$-st element with a value of 1. $\prod_{h=k+1}^{i-1}(1-c[h]) = 1$ iff $c[k+1] = \ldots = c[i-1] = 0$. 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 $j-1$ values of 1 in $c[1], \ldots, c[i-1]$. Multiplying this by $c[i]$ we get that $M[j,i] = 1$ iff $c[i]$ is the j-th value of 1.

B PROTOCOLS FOR THE GENERIC CASE

In this section we give in full two protocols that were mentioned above and where described only by listing the differences from a previous protocol. Specifically, *MultiQuerierRangeSearching* (Protocol 4) generalizes *RangeSearchingProtocol* (Protocol 2) to the multi-querier case and *PPRangeSearch* (Algorithm 5) generalizes *PPRangeSearch* (Algorithm 3) to the case of a general function f.

MultiQuerierRangeSearching includes the changes to efficiently support multiple queriers by encrypting T using AES (we use AES as an example for an efficient encryption scheme) and then transciphering the AES-encrypted T into FHE-encrypted T.

PPRangeSearch is more general than just computing a count of the points, $|P \cap \gamma|$. Specifically, instead of using $|A \cup B| = |A| + |B|$ it uses $f(A \cup B) = g(f(A), f(B))$. This change is reflected in how the contribution of various nodes is added into the output.

C 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 6, 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 3) securely evaluates $f(\gamma \cap P)$ in a circuit whose size is $O(t \cdot n^{1-\frac{1}{d}+\epsilon} + n^{1+\epsilon})$ and depth is $O(\ell \cdot \log n)$.*

Proof. **Correctness.** The correctness of the plaintext algorithm for range searching was proven in [4, 5, 33]. 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 3: (1) computes *IsContaining* and *IsCrossing* r times; (2) builds a selection matrix M; (3) multiply the vector of children by M to get a vector of 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 $O(r^2 \cdot \xi)$. Denote the size of each child (including its subtree) by $S(n)$ then copying ξ children (out of r) takes $O(r \cdot \xi \cdot S(n))$. When $d = 1$ we have $S(n) = O(n)$ (since the partition procedure generates a complete tree) and when $d > 1$ we have $S(n) = O((n/r)^{\frac{1}{1-\log_r h}}) = O((n/r)^{1+\epsilon})$, where $1 + \epsilon = \frac{1}{1-\log_r h}$ (this follows from Lemma 6).

Protocol 4: *MultiQuerierRangeSearching*

Parties: Data owner, q Queriers $Q_1, \ldots, Q_q$, 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, AES key ak.
Querier Q_j Input: A pair (sk_j, pk_j) of secret and public keys.
Ranges $\gamma_1^j, \ldots, \gamma_c^j \in \Gamma$, where $c \in \mathbb{N}$ is polynomial in the security parameter of (sk_j, pk_j).
The Cloud has no input.
Querier Q_j Output: $|P \cap \gamma_i^j|$, for $i = 1, \ldots, c$.
The cloud and the data owner have no output.

Querier Q_j Performs:
  Send pk_j to the Cloud and to the Data owner.
Data owner Performs:
  Choose a parameter $0 < r < n$.
  (T', ξ, h) := Build a partition tree for P and Γ with parameter r. `// See Section 6`
  $T := FillTree(n, r, h, T')$. `// See Algorithm 6`
  $[\![T]\!]_{ak}$:= encrypt $v.val, v.\sigma$ for every $v \in T$ using ak.
  $[\![ak]\!]_{pk_j} = \mathrm{Enc}_{pk_j}(ak)$, for $j = 1, \ldots, q$.
  Send $[\![T]\!]_{ak}, [\![ak]\!]_{pk_1}, \ldots, [\![ak]\!]_{pk_q}, n, \Gamma, \xi, r, h$ to Cloud.
foreach $j = 1, \ldots, q$ **do**
  Cloud performs:
    $[\![T]\!]_{pk_j} = transcipher([\![T]\!]_{ak}, [\![ak]\!]_{pk_j})$.
  foreach $i = 1, \ldots, c$ **do**
    Querier Q_j performs:
      $[\![\gamma_i]\!]_{pk_j} := \mathrm{Enc}_{pk_j}(\gamma_i)$
      Send $[\![\gamma_i]\!]_{pk_j}$ to Cloud.
    Cloud performs:
      $[\![x_i]\!]_{pk_j} := PPRangeSearch_{n,d,\Gamma,\xi,r,h}([\![T]\!]_{pk_j}, [\![\gamma_i]\!]_{pk_j})$
      `// See Algorithm 3`
      Send $[\![x_i]\!]_{pk_j}$ to querier.
    Querier performs:
      $x_i := \mathrm{Dec}_{sk_j}([\![x_i]\!]_{pk_j})$
      Output x_i

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

$$T(n) \le O(r \cdot t) + O(r^2 \cdot \xi) + O(r \cdot \xi \cdot S(n/r)) + \xi \cdot T(h \cdot n/r) \quad (1)$$

When $d > 1$ we have $S(n) = (n/r)^{1+\epsilon}$ and $\xi = O(r^{1-1/d})$. Then this solves to

$$T_{d>1}(n) = \sum_{i=0}^{\log_{r/h} n-1} O((r \cdot t + r^2 \cdot \xi) \cdot \xi^i) + \sum_{i=0}^{\log_{r/h} n-1} O(r \cdot \xi (\frac{n}{r^{i+1}})^{1+\epsilon} \cdot \xi^i)$$

$$= (r \cdot t + r^2 \cdot \xi) \frac{\xi^{\log_{r/h} n} - 1}{\xi - 1} + O(r \cdot n^{1+\epsilon} \cdot \frac{1 - (\frac{\xi}{r^{1+\epsilon}})^{\log_{r/n}+1}}{1 - \frac{\xi}{r^{1+\epsilon}}})$$

$$= O((r \cdot t + r^2 \cdot \xi)\xi^{\log_{r/h} n} + r \cdot n^{1+\epsilon}) = O(t \cdot n^{1-1/d+\epsilon_2} + n^{1+\epsilon_3}),$$

Algorithm 5: *PPRangeSearch*$_{n,r}(\llbracket T \rrbracket, \llbracket \gamma \rrbracket)$

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 $x = |P \cap \gamma|$.

if *v is a leaf* **then**
  // check whether $v.\sigma \subset \gamma$
  $\llbracket Cont \rrbracket := IsContaining(\llbracket v.\sigma \rrbracket, \llbracket \gamma \rrbracket)$
  Output $\llbracket Cont \rrbracket \cdot \llbracket v.val \rrbracket$
else
  foreach $i = 1, \dots, r$ **do**
    // check whether $v.child[i].\sigma \subset \gamma$
    $\llbracket Cont[i] \rrbracket := IsContaining(\llbracket v.child[i].\sigma \rrbracket, \llbracket \gamma \rrbracket)$
  $\llbracket x \rrbracket = \llbracket f(\emptyset) \rrbracket$
  foreach $i = 1, \dots, r$ **do**
    $\llbracket x \rrbracket := \llbracket Cont[i] \rrbracket \cdot g(\llbracket x \rrbracket, \llbracket v.child[i].val \rrbracket) + (1 - \llbracket Cont[i] \rrbracket) \cdot \llbracket x \rrbracket$
  foreach $i = 1, \dots, r$ **do**
    // check whether $c.child[i].\sigma$ crosses γ
    $\llbracket Cross[i] \rrbracket := IsCrossing(\llbracket v.child[i].\sigma \rrbracket, \llbracket \gamma \rrbracket)$
  $\llbracket M \rrbracket := BuildSelectionMatrix_{r,\xi}(\llbracket Cross \rrbracket)$// See Algorithm 1
  $\llbracket child' \rrbracket := M \cdot \llbracket v.child \rrbracket$
  $\llbracket Cross' \rrbracket := M \cdot \llbracket Cross \rrbracket$
  foreach $i = 1, \dots, \xi$ **do**
    $\llbracket x \rrbracket := \llbracket Cross'[i] \rrbracket \cdot g(\llbracket x \rrbracket, \llbracket child'[i].val \rrbracket) + (1 - \llbracket Cross'[i] \rrbracket) \cdot \llbracket x \rrbracket$
  Output $\llbracket x \rrbracket$

where $\epsilon_2 = (1 - \frac{1}{d})\frac{\log_r h}{1 - \log_r h}$ and $\epsilon_3 = \frac{\epsilon_2}{1 - \log_r h}$.

For the case $d = 1$ we have from Lemma 1 $\xi = 2$, $h = 1$ and $S(n) = O(n)$. Putting these into Equation 1 we get

$$\begin{aligned} T_{d=1}(n) &= \sum_{i=0}^{\log_r n - 1} O((r \cdot t + r^2 \cdot \xi) \cdot \xi^i) + \sum_{i=0}^{\log_r n - 1} O(r \cdot \xi(\frac{n}{r^{i+1}}) \cdot \xi^i) \\ &= (r \cdot t + r^2 \cdot \xi)\frac{\xi^{\log_r n} - 1}{\xi - 1} + O(r \cdot n \cdot \frac{1 - (\frac{\xi}{r})^{\log_{r/n} + 1}}{1 - \frac{\xi}{r}}) \\ &= O((r \cdot t + r^2 \cdot 2)2^{\log_2 n \cdot \log_r 2} + r \cdot n) \\ &= O(r \cdot t \cdot n^{\log_r 2} + r^2 \cdot n^{\log_r 2} + r \cdot n) = O(t \cdot n^{\epsilon} + n), \end{aligned}$$

where $\epsilon = \log_r 2$.

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 *BuildSelectionMatrix* 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 *BuildSelectionMatrix* subcircuit followed by ξ subcircuits that compute range search queries on subtrees of height $(d - 1)$. This yields a circuit depth of $d \cdot \ell + d \cdot O(\xi \log r) = O(\ell \cdot \log n)$. Since the tree height is $\log_{r/h} n$ and ξ is a parameter that depends on r. ◻

Corollary 1. *When $d > 1$ and for fixed values of t and n, our algorithm improves over the naïve algorithm for values of r for which:*

$$\log^2 r - (d + 1)\log n \log r + \log n \log r < 0$$

and

$$((1 - d)\log n \log h - \log t)\log^2 r - 2\log t \log h \log r - \log^2 h < 0.$$

We remind that in approximate schemes (e.g., CKKS) $t = t(n)$ is a function of n because it needs to be more accurate to keep the aggregated noise small.

Proof. The running time of our algorithm is

$$O(r \cdot t \cdot n^{1 - 1/d + \epsilon_2} + r \cdot n^{1 + \epsilon_3})$$

where $\epsilon_2 = (1 - \frac{1}{d})\frac{\log_r h}{1 - \log_r h}$ and $\epsilon_3 = \frac{\epsilon_2}{1 - \log_r h}$, while the naïve algorithm takes $O(t \cdot n)$. We get that

$$r \cdot t \cdot n^{1 - 1/d + \epsilon_2} < t \cdot n$$

when

$$\log^2 r - (d + 1)\log n \log r + \log n \log r < 0.$$

In addition, we get that

$$r \cdot n^{1 + \epsilon_3} < t \cdot n$$

when

$$((1 - d)\log n \log h - \log t)\log^2 r - 2\log t \log h \log r - \log^2 h < 0.$$

◻

C.1 Axis-Parallel Hyper-Boxes

In this subsection we consider the special case of axis-parallel hyper-boxes. An axis-parallel hyper-box $\gamma \subset \mathbb{R}^d$, defined by the parameters $a_1, \dots, a_d$ and $b_1, \dots, b_d$, is the range

$$\gamma = \{(x_1, \dots, x_d) \in \mathbb{R}^d \mid a_i \le x_i \le b_i, \text{ for } i = 1, \dots, d\}.$$

In this special case we can construct a multi-level data-structure that is more efficient than the generic data-structure when $d > 1$.

A multi-level data structure is a common practice (see for example [13]), but we iterate it here in a nutshell. In what follows we use $D^{(d)}(n)$ to denote a d-level data structure built on n points, and we use $Proj(P, \{x_1\})$ and $Proj(P, \{x_2, \dots, x_d\})$ to denote the projection of P onto the x_1 axis and onto the subspace spanned by $x_2, \dots, x_d$, respectively.

Given $P = \{p_1, \dots, p_n\} \subset \mathbb{R}^d$ we construct the data structure in the following recursive way. We construct a primary 1-dim partition tree $T^{(1)}$ for the points $Proj(P, x_1)$. At every vertex v we construct a secondary data structure for the points $Proj(S_v, \{x_2, \dots, x_d\})$. The values for $T^{(2)}$ are $Proj(S_v, \{x_2\})$. We then continue in recursion to build $T^{(3)}$ from every node u of $T^{(2)}$ and so on. See Figure 6 for an example.

Lemma 4. *The size of the d-level data structure described above of a set of points $P \subset \mathbb{R}^d$, where $|P| = n$ is $S(n) = n\log^{d-1} n$.*

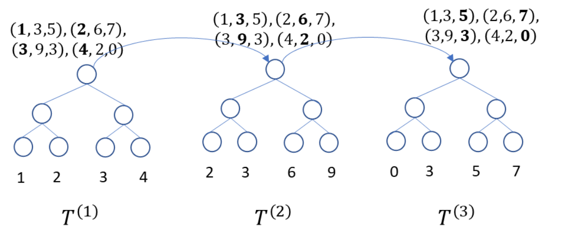

Figure 6: A multi-level data structure for the points $P = \{(1,3,5),(2,6,7),(3,9,3),(4,2,0)\}$. The primary tree, $T^{(1)}$, (left)) built for the values $1,2,3,4$ (the projection of P onto the x_1-axis, shown in bold). Each node v keeps a secondary tree for the points in S_v built for the projections of S_v onto the x_2-axis. The figure shows only the secondary tree $T^{(2)}$ built for the root of $T^{(1)}$. In this example it is built for the values $2,3,6,9$ (shown in bold). Each node u of each of the secondary trees keeps a ternary tree for the points in S_u built for the projections of S_u onto the x_3-axis. The figure shows (on the right) only the ternary tree $T^{(3)}$ built for the root of $T^{(2)}$ that is depicted. In this example the values are $0,3,5,7$ (shown in bold).

Proof. We prove this claim by induction. The case $d = 1$ follows immediately from Theorem 1. Assume that for a $(d-1)$-level data structure for $(d-1)$ dimensions the storage complexity is $O(n \log^{d-1} n)$. We now prove it for a data structure built for d levels (and dimensions). Denote by $T^{(1)}$ the primary tree. Each node v of $T^{(1)}$ keeps a $(d-1)$-level data structure of S_v. We remind that the depth of $T^{(1)}$ is $\log n$. We denote by *floor* the set of nodes with the same distance from the root. There are r^i nodes in the i-th floor, each associated with $\frac{n}{r^i}$ points. Each node is associated with a $(d-1)$-level data structure. By the induction assumption the storage complexity of a $(d-1)$-level data structure for m points is $O(m \log^{d-2} m)$. It follows that the total storage complexity is

$$S(n) = \sum_{0}^{\log_r n} r^i O\left(\frac{n}{r^i} \log^{d-2} \frac{n}{r^i}\right) = O(n \log^{d-1} n).$$

□

Lemma 5. *Answering a range searching query for axis-parallel hyper-boxes with the multi-level data structure described above requires a circuit of size $O(t \cdot n^{\epsilon} + n \log^{d-1} n)$, where $\epsilon > 0$ is arbitrarily small and t is as in Theorem 1.*

Proof. We prove this claim by induction on the dimension d, which is also the number of levels in the data structure. For $d = 1$, it follows from Theorem 1 that $T(n) = O(t \cdot n^{\epsilon} + n)$. Denote the time to query a multi-level data structure for n points with d levels as $T^{(d)}(n)$, and assume $T^{(d-2)}(n) = O(t \cdot n^{\epsilon} + n \cdot \log^{(d-2)} n)$.

When answering a query with a d-level data structure at each inner node we: (1) compute *IsContaining* and *IsCrossing* r times; (2) for each child v, compute *PPRangeSearch* on the next level data structure built at v, multiply it by the output of *IsContaining* and add it to *count*; (3) build selection matrix M; (4) multiply the vector of children by M to get a vector of $\xi = 2$ children and (5) recurse into 2 children of v.

Computing all *IsContaining* and *IsCrossing* takes $O(t \cdot r)$ time. From Lemma 2 (and remembering that $\xi = 2$), computing M takes $O(r^2)$. Copying $\xi = 2$ children (out of r) takes $O(r \cdot n \cdot \log^{d-1} n)$ since by Lemma 4 the size of each sub tree is $O(n \cdot \log^{d-1} n)$. Computing r queries on the next level data structure takes

$$rT^{(d-1)}(n/r) = O(r \cdot t \cdot (n/r)^{\epsilon} + r \cdot (n/r) \cdot \log^{d-2}(n/r))$$

by the induction assumption.

Denote by $T^{(d)}(n)$ the time to answer a query with a d-level data structure of n points for axis-parallel hyper-box ranges. It follows that $T^{(d)}(n)$, when $d > 1$, is given by the following recursion rule:

$$\begin{aligned} T^{(d)}(n) &\leq O(r \cdot t) + O(r^2) + O(r \cdot (n/r) \log^{d-1}(n/r))) \\ &\quad + O(r \cdot T^{(d-1)}(n/r)) + \xi \cdot T^{(d)}(n/r). \end{aligned}$$

Putting in the induction assumption we get:

$$\begin{aligned} T^{(d)}(n) &\leq O(r \cdot t) + O(r^2) + O(n \log^{d-1} n) + \\ &\qquad O(r \cdot (t \cdot n^{\epsilon_{d-1}} + n \log^{d-2} n)) + 2 \cdot T^{(d)}(n/r). \end{aligned}$$

This solves to

$$\begin{aligned} T^{(d)}(n) &\leq O(r \cdot t \cdot n^{\log_r 2}) + O(r^2 \cdot n^{\log_r 2}) + O(r \cdot n \log^{d-1} n) + \\ &\quad O(t \cdot n^{\epsilon_{d-1} + \log_r 2}) + O(n \log^{d-2} n) \leq O(t \cdot n^{\epsilon} + n \log^{d-1} n) \\ &= O(t \cdot n^{\epsilon} + n \log^{d-1} n), \end{aligned}$$

where $\epsilon = d \cdot \log_r 2$. □

D HIDING TREE STRUCTURE

In this section we describe Algorithm 6 which we call *FillTree* that adds empty nodes to an input tree T as mentioned in Section 7.3.

Algorithm 6: *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.

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**
- Add an empty child node to an inner node that has less than r children.
- Add r empty children nodes to leaves whose distance from the root is less than $\lceil \log_{r/h} n \rceil$.

In Figure 7 we show an example of a partition tree and its full version after empty nodes have been added to it.

Lemma 6. *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, $\Pi = \{(P'_1, \sigma_1), \ldots, (P'_m, \sigma_m)\}$ 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 $n^{\frac{1}{1 - \lceil \log_r h \rceil}} = O(n^{1+\epsilon})$ nodes.*

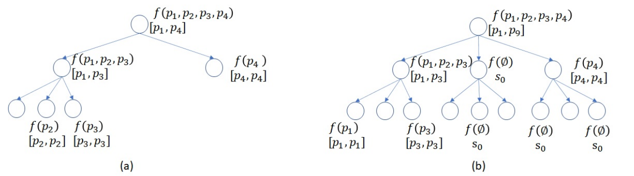

Figure 7: An example of a partition tree before (left) and after (right) applying $FillTree$. On the left is a partition tree for 4 points $p_1, \ldots, p_4$. On the right is a full version of the same tree. All the inner nodes need to have the same number of children, so empty nodes are added to the 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 $v.val = f(\emptyset)$ and empty bounding simplex s_0.

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 $r^{\lceil \log_{r/h} n \rceil} = r^{\lceil \log_r n^{\frac{1}{1-\log_r h}} \rceil} = O(n^{\frac{1}{1-\log_r h}}) = O(n^{1+\epsilon})$. □

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

Lemma 7. *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 $T_i = FillTree(n, r, h, T_i')$, for $i = 1, 2$.*

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. □

E SECURITY ANALYSIS

In this section we prove the privacy against the cloud. The data owner has no output. The querier output is only $f(P \cap \gamma_1), \ldots, f(P \cap \gamma_c)$ which is required by the problem statement. Both don't have anything else in their view and therefore learn nothing else.

Theorem 3. *Algorithm 3 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

$$\text{view}_C = (pk, d, n, \Gamma, h, \xi, r, c, [\![\gamma_1]\!], \ldots, [\![\gamma_c]\!], [\![T]\!], [\![f(P \cap \gamma_1)]\!], \ldots, [\![f(P \cap \gamma_c)]\!], [\![I_1]\!], [\![I_2]\!], \ldots),$$

where pk is the public key as was received from the querier, d, n, Γ, h, r and ξ are the parameters the partition tree was built with and were received from the data owner, $[\![\gamma_1]\!], \ldots, [\![\gamma_c]\!]$ and $[\![T]\!]$ which is the partition tree whose structure is given but the content in its nodes: $[\![v.val]\!]$ and $[\![v.\sigma]\!]$, for every node $v \in T$, is encrypted. In addition, the view includes $[\![f(P \cap \gamma_1)]\!], \ldots, [\![f(P \cap \gamma_c)]\!]$ and all the intermediate values $([\![I_1]\!], [\![I_2]\!], \ldots)$ that are generated during the execution of Algorithm 3. For simplicity, we reorder

$$\text{view}_C = (pk, d, n, \Gamma, h, \xi, r, c, \text{structure of } T, [\![m_1]\!], [\![m_2]\!], \ldots),$$

where $[\![m_1]\!], \ldots$ 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)

$$\text{view}_C^{restr} = (pk, d, n, \Gamma, h, \xi, r, c, [\![m_1]\!], [\![m_2]\!], \ldots).$$

is computationally indistinguishable from this view:

$$\text{view}_Z = (pk, d, n, \Gamma, h, \xi, r, c, [\![0]\!], [\![0]\!], \ldots).$$

Lemma 8. *view_C^{restr} is computationally indistinguishable from view_Z.*

Proof. Consider this set of views:

$$\begin{aligned}
\text{view}_C^{restr} = \text{view}_C^0 &= (pk, d, n, \Gamma, h, \xi, r, c, [\![m_1]\!], [\![m_2]\!], \ldots) \\
\text{view}_C^1 &= (pk, d, n, \Gamma, h, \xi, r, c, [\![0]\!], [\![m_2]\!], \ldots) \\
\text{view}_C^2 &= (pk, d, n, \Gamma, h, \xi, r, c, [\![0]\!], [\![0]\!], \ldots) \\
&\vdots \\
\text{view}_Z &= (pk, d, n, \Gamma, h, \xi, r, c, [\![0]\!], [\![0]\!], \ldots),
\end{aligned}$$

where $\text{view}_C^{(i)}$ is different from $\text{view}_C^{(i-1)}$ by replacing $[\![m_i]\!]$ with $[\![0]\!]$, for $i = 1, 2, \ldots$. 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 $[\![m_i]\!]$ is distinguishable from $[\![0]\!]$ without having sk. Since the number of ciphertexts is polynomial in the security parameter of sk, the view this contradicts the semantic security of FHE. □

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 $(pk, d, n, \Gamma, h, \xi, r, c)$ 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 d, n, ξ, r, h and applies $\tau := FillTree(n, r, h, \tau')$; (3) encrypts γ and $v.val$ and $v.\sigma$ for every node $v \in \tau$; (3) executes Algorithm 3 *PPRangeSearch*; (4) outputs the view of the simulator

$$\text{view}_S = (pk, d, n, \Gamma, h, \xi, r, c, \text{ structure of } \tau, [\![m_1]\!], [\![m_2]\!], \ldots)$$

By Lemma 7 T and τ have the same structure. By Lemma 8 the restricted simulator view (without the structure of τ) view_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. □

F 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 $f : 2^P \to \mathbb{N}$ be defined as $f(A) = |A|$ and $g : \mathbb{N} \times \mathbb{N} \to \mathbb{N}$ 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 $S_{v_1}, \ldots, S_{v_m}$ 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 [8] can be used to efficiently output $P \cap \gamma$ if $|P \cap \gamma|$ is small.

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