Klee's Measure Problem Made Oblivious

Thore Thießen(✉) and Jan Vahrenhold

Department of Computer Science, University of Münster, Münster, Germany
{t.thiessen,jan.vahrenhold}@uni-muenster.de

Abstract. We study Klee's measure problem — computing the volume of the union of n axis-parallel hyperrectangles in $\mathbb{R}^d$ — in the oblivious RAM (ORAM) setting. For this, we modify Chan's algorithm [12] to guarantee memory access patterns and control flow independent of the input; this makes the resulting algorithm applicable to privacy-preserving computation over outsourced data and (secure) multi-party computation.

For $d = 2$, we develop an oblivious version of Chan's algorithm that runs in expected $\mathcal{O}(n \log^{5/3} n)$ time for perfect security or $\mathcal{O}(n \log^{3/2} n)$ time for computational security, thus improving over optimal general transformations. For $d \geq 3$, we obtain an oblivious version with perfect security while maintaining the $\mathcal{O}(n^{d/2})$ runtime, i. e., without any overhead.

Generalizing our approach, we derive a technique to transform divide-and-conquer algorithms that rely on linear-scan processing into oblivious counterparts. As such, our results are of independent interest for geometric divide-and-conquer algorithms that maintain an order over the input. We apply our technique to two such algorithms and obtain efficient oblivious counterparts of algorithms for inversion counting and computing a closest pair in two dimensions.

Keywords: Klee's measure problem · Oblivious RAM · Data-oblivious algorithms · Data-oblivious divide-and-conquer algorithms

1 Introduction

First introduced by Klee [20] in 1977, *Klee's measure* problem is a well-known problem in computational geometry:

> Given a set B of n axis-parallel hyperrectangles (for short: *boxes*) in $\mathbb{R}^d$, compute $\|\bigcup(B)\|$. Here, $\|\cdot\|$ is the d-dimensional volume of a (measurable) subset of $\mathbb{R}^d$.

On the theoretical side, the problem is related to other geometric problems, e. g., the *depth* and *coverage* problems [12]. There has been a series of works improving the upper bounds for Klee's measure problem [11,12,16,23]. The currently best runtime of $\mathcal{O}(n \log n + n^{d/2})$ for any (fixed) number d of dimensions is obtained by Chan's algorithm [12].

A special case of Klee's measure problem, computing the so-called *hypervolume indicator*, is used in, e. g., evaluating multi-objective optimizations [19].

For the hypervolume indicator, the boxes are restricted to have a common lower bound in all dimensions [12,19]. For this and other special cases, faster algorithms than for the general problem are known [12].

Oblivious Algorithms. We design a *data-oblivious* (for short: *oblivious*) algorithm for Klee's measure problem. In the random access machine (RAM) context, the notion of (data-)obliviousness was introduced by Goldreich and Ostrovsky [17]. Informally, the requirement is that the *probe sequence*, i. e., the sequence of memory operations and memory access locations, must be independent of the input. This guarantees that no information about the input can be derived from observing the memory access patterns. Oblivious algorithms — in combination with encryption — can be used to perform privacy-preserving computation over outsourced data. Additionally, oblivious algorithms can be transformed into efficient protocols for multi-party computation [15,25].

We distinguish two notions of obliviousness: Let x and y be two inputs of equal length n. Then the probe sequences for x and y must be identically distributed (*perfect security*) or indistinguishable by any polynomial-time algorithm except for negligible probability in n (*computational security*).[1] In line with standard assumptions for oblivious algorithms [6,15], we include the control flow in the probe sequence. Access to a constant number of *private memory* cells (registers) as well as the memory contents are not considered to be part of the probe sequence (since we may assume that the memory is encrypted [17]).

As the underlying model of computation, we assume the *word RAM* model (in line with standard assumptions for oblivious algorithms). However, we note that our results with perfect security also hold in the *real RAM* model usually assumed in computational geometry. We use oblivious sorting as a building block. There are oblivious sorting algorithms with $\mathcal{O}(n \log n)$ runtime and perfect security, e. g., due to asymptotically optimal sorting networks [1].

In recent years, there has been a lot of progress regarding oblivious algorithms. It is known that — in general — transforming a RAM program into an oblivious program incurs an $\Omega(\log N)$ (multiplicative) overhead [17,21] (where N is the space used by the program). On the constructive side, there is an ORAM construction matching this lower bound [4]. This provides — for programs with at least linear runtime — a black-box transformation to achieve obliviousness with only a logarithmic overhead in runtime. Even with such general transformations, there are still some limitations: Optimal general ORAM transformations currently entail high constant runtime factors that make them unsuitable for practical application [4]. Additionally, all known optimal transformations only satisfy the weaker requirement of computational security [7]. The state-of-the-art general transformation with perfect security is due to Chan et al. [9] and achieves a runtime overhead of $\mathcal{O}(\log^3 N / \log \log N)$.

To address these issues and overcome the $\Omega(\log n)$ lower bound associated with the general transformation, oblivious algorithms for specific problems have

[1] See the definitions by Asharov et al. [3, Section 3] for a more formal introduction of oblivious security applicable to oblivious algorithms.

been proposed: examples include sorting algorithms [2,22], graph algorithms [6], and some algorithms for fundamental geometric problems [15]. To the best of our knowledge, oblivious algorithms for Klee's measure problem and its variants have not been considered in the literature so far.

Algorithm 1. Chan's RAM algorithm [12] to compute Klee's measure for $d \geq 2$. The coordinates of the boxes B are sorted in a pre-processing step.

```
1: function MEASURE($B, \Gamma$)
2:     if $|B| \leq$ some suitably chosen constant then
3:         return $\|\Gamma \setminus \bigcup(B)\|$ (computed using a brute-force approach)    ▷ $\mathcal{O}(d)$
4:     SIMPLIFY($B, \Gamma$)                                                              ▷ $\mathcal{O}(d \cdot n)$
5:     $\langle \Gamma_\mathrm{L}, \Gamma_\mathrm{R} \rangle \leftarrow$ CUT($B, \Gamma$)          ▷ $\mathcal{O}(d \cdot n)$
6:     $B_\mathrm{L} \leftarrow$ all $b \in B$ intersecting $\Gamma_\mathrm{L}$; $B_\mathrm{R} \leftarrow$ all $b \in B$ intersecting $\Gamma_\mathrm{R}$    ▷ $\mathcal{O}(d \cdot n)$
7:     return MEASURE($B_\mathrm{L}, \Gamma_\mathrm{L}$) + MEASURE($B_\mathrm{R}, \Gamma_\mathrm{R}$)
```

2 Warm-Up: Shaving Off a $\log \log n$ Factor

As a warm-up, we first show how to improve over general and naive transformations of Chan's algorithm for $d = 2$ by a $\mathcal{O}(\log \log n)$ factor in runtime. We also introduce the representation of the boxes used throughout the paper. In Sect. 3, we build on this approach when showing how to improve the runtime for $d = 2$ to our main result. For completeness, we first describe Chan's algorithm as presented in the original paper [12]. We will also briefly sketch how to obtain an oblivious modification for $d \geq 3$.

2.1 Original Algorithm

The input for Klee's measure problem is a set B of n axis-parallel hyperrectangles. Chan's algorithm, shown in Algorithm 1, first computes the measure $\overline{m}$ of the complement of $\bigcup(B)$ relative to some box Γ (*domain*) containing all boxes B. For the final result, $\overline{m}$ is subtracted from the measure of Γ. The algorithm to compute $\overline{m}$ is — at its core — a simple divide-and-conquer algorithm with two helper functions: SIMPLIFY and CUT. In each call, a set of boxes intersecting the current domain Γ is processed. Throughout the algorithm, the coordinates of the boxes B are maintained in sorted order for each dimension.

The main idea of SIMPLIFY is to remove the area covered by the boxes $D \subseteq B$ that cover the domain Γ in all but one dimension: For a given dimension $i \in \{1, \ldots, d\}$, let D_i denote the boxes covering Γ in all but dimension i. The simplification can be performed by first computing the union of the boxes D_i and then adjusting the x_i-coordinates of all other boxes in B so that the volume of all connected components of $\bigcup(D_i)$ is reduced to 0. Since the coordinates are sorted in each dimension, this simplification can be realized with a linear scan in each dimension. To maintain the measure of the complement, the extent of Γ in dimension i (*height*) is reduced accordingly, i. e., by the height of $\bigcup(D_i)$.

Intuitively, the simplification reduces the problem complexity by reducing the number of sub-domains a box intersects with. Consider the problem for $d = 2$: When cutting the domain Γ for the recursion, the simplification guarantees that — on every level of the recursion — each box only has to be considered in a constant number of sub-domains; this is since the area covered by the box is removed from the covered sub-domains in-between.

For the Cut-step, a weighted median is computed that splits Γ into two sub-domains Γ_{L} and Γ_{R}. Like in k-d-tree constructions, the algorithm cycles through the dimensions in a round-robin manner. In the analysis, Chan shows that cutting reduces the weight of each sub-domain by a factor of $2^{2/d}$. Since the weight is related to the number of boxes by a constant factor (depending on d), this also provides an upper bound for the number of boxes in each sub-domain. The runtime for Algorithm 1 is thus bounded by the recurrence $T(n) = 2 \cdot T(n \,/\, 2^{2/d}) + \mathcal{O}(n)$. With the time $\mathcal{O}(n \log n)$ for pre-sorting the coordinates this implies a runtime of $\mathcal{O}(n^{d/2})$ for $d \geq 3$ and a runtime of $\mathcal{O}(n \log n)$ for $d = 2$ overall.

2.2 High Dimensions

Before presenting an oblivious modification of Chan's algorithm for $d = 2$, we will briefly consider higher dimensions $d \geq 3$. Here, we note that Chan's algorithm can easily be transformed into an oblivious algorithm while maintaining the runtime complexity. This leaves the planar case $d = 2$ as the "hard case".

To see why the case $d \geq 3$ is actually easier, consider again the recurrence bounding the runtime: As we will discuss in Sect. 2.5, the main difficulty for the oblivious algorithm is to maintain the sorted coordinates for an efficient simplification. If we solve this naively by sorting, we have costs of $\mathcal{O}(n \log n)$ instead of $\mathcal{O}(n)$ in each recursive call. This results in a runtime bounded by the recurrence $T(n) = 2 \cdot T(n \,/\, 2^{2/d}) + \mathcal{O}(n \log n)$. A naive transformations thus leads to a runtime of $T(n) \in \mathcal{O}(n \log^2 n)$ for $d = 2$, yet for $d \geq 3$ this immediately solves to $T(n) \in \mathcal{O}(n^{d/2})$, maintaining the runtime of the original algorithm.

There still remain challenges for an oblivious implementation of Chan's algorithm for $d \geq 3$. We will briefly sketch how to address them: For the representation, it suffices to store the individual boxes with their bounds in each dimension; the boxes can be duplicated and sorted as needed. An oblivious implementation also needs to ensure that the input size for the recursive calls does only depend on the problem size n, otherwise the runtimes for the recursive calls might leak information about the input. For this, we note that the recurrence given by Chan already bounds the number of boxes in the recursive calls. By carefully padding the input with additional (empty) boxes we can always recursively call with the worst-case size. Splitting the n boxes for the recursive calls can be done by duplication and oblivious routing in $\mathcal{O}(n \log n)$ time [18].

2.3 Oblivious Box Representation

For the planar case $d = 2$, Chan's algorithm can be simplified to only cutting the domain in the x-dimension [12]. Consequently, simplification is only performed

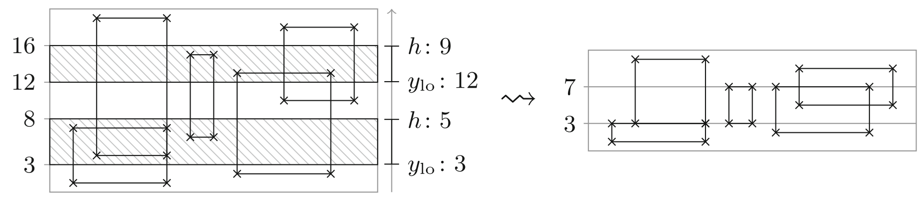

Fig. 1. Simplification of a sub-domain Γ with 5 boxes to remove the area covered by the hatched slabs. The result of the simplification is shown on the right.

for the y-dimension. We say a box b *covers* the domain Γ iff b spans over Γ in the x-dimension; in this case we call b a *slab*. For $d = 2$ the cutting can also be performed without weights, balancing the number of vertices in each sub-domain. Note that since the initial domain contains all vertices and cuts are only performed in the x-dimension, we only need to check the x-dimension to see if a vertex is contained in a given sub-domain Γ.

We represent each box by its four vertices. The sequence of vertices is not maintained in a fixed order; instead, we explicitly sort the sequence when needed. Since we need information about the horizontal extent of the boxes when processing the vertices, we keep the full x-interval in each vertex. We also store the relative location in both dimensions, i. e., whether the vertex is a lower (lo) or upper (hi) vertex. So overall, each vertex is represented by a tuple

$$b \in \underbrace{\mathbb{R}^2 \times \{\text{lo}, \text{hi}\}}_{x\text{-interval}} \times \underbrace{\mathbb{R} \times \{\text{lo}, \text{hi}\}}_{y\text{-coordinate}} .$$

For a vertex b, we write b_x, b_y to denote the coordinates and $b_{[x]}$ to denote the x-interval. We write $\text{box}(b)$ to refer to the box the vertex b (partially) represents.

Since our algorithms for the planar case are based on processing the vertices in y-order, we simplify terminology by referring to the lower y-vertices as *start*- and to the upper y-vertices as *endpoints*. Similarly, we will refer to the lower x-vertices as *left* and to the upper x-vertices as *right* vertices. Storing the x-interval in each vertex allows us to represent the box by either both left or both right vertices. We will leverage this when cutting the box at sub-domain boundaries.

2.4 Simplification in One Sub-domain

For an oblivious modification of Chan's algorithm we need a way to *obliviously* simplify a given domain Γ, i. e., to remove the area covered by the slabs D. To do so, we reverse the order of cutting and simplification: We remove the area covered by slabs in a sub-domain (while maintaining the measure of the complement) before recursing. This exchange of the two algorithmic steps means that in each recursive call, the number of boxes (and vertices) remains unchanged — an important requirement for not leaking any information about the input.

For the simplification, Chan's algorithm first computes the connected components of $\bigcup(D)$ and then adjusts the coordinates of the remaining boxes in a

Algorithm 2. Algorithm to remove the area covered by slabs D in Γ. The return value is the combined height of all connected components of $\bigcup(D)$.

```
1: procedure OblSimplify($B, \Gamma$)
2:     $y_{\mathrm{lo}} \leftarrow$ undef; $c \leftarrow 0$; $h \leftarrow 0$          ▷ $y$-anchor; overlap; height
3:     for each vertex $b \in B$ do                                     ▷ in $y$-order
4:         if box($b$) covers $\Gamma$ then Update($\langle y_{\mathrm{lo}}, c, h\rangle, b$)     ▷ update the state
5:         if $b$ is in $\Gamma$ then Adjust($\langle y_{\mathrm{lo}}, c, h\rangle, b$)          ▷ adjust the vertex
6:     return $h$
7: procedure Update($\langle y_{\mathrm{lo}}, c, h\rangle, b$)
8:     if $b$ is a startpoint $\wedge\ c = 0$ then $y_{\mathrm{lo}} \leftarrow b_y$            ▷ start of a component
9:     if $b$ is a startpoint then $c \leftarrow c + 1$ else $c \leftarrow c - 1$
10:    if $b$ is an endpoint $\wedge\ c = 0$ then $h \leftarrow h + (b_y - y_{\mathrm{lo}})$   ▷ end of the component
11: procedure Adjust($\langle y_{\mathrm{lo}}, c, h\rangle, b$)
12:    $\Delta \leftarrow h$
13:    if $c > 0$ then $\Delta \leftarrow \Delta + (b_y - y_{\mathrm{lo}})$                ▷ in the component
14:    $b_y \leftarrow b_y - \Delta$
```

synchronized traversal. Unfortunately, such traversal is infeasible in the oblivious context since doing so might leak information about the coordinates' distribution. To address this, we redesign the subroutine to perform a single linear scan: The vertices B are processed in y-order (start- before endpoints); each vertex $b \in B$ contained in the sub-domain Γ ($b_x \in \Gamma_{[x]}$) is adjusted to remove the area covered by the slabs D. Specifically, b_y is reduced by the height of $\bigcup(D)$ below b_y.

The resulting Algorithm 2 is the key ingredient to realize the subroutine Simplify; we can maintain the measure of the complement by subtracting the obtained value h from the height of Γ. An example of a simplification is shown in Fig. 1. It is essential that the algorithm performs exactly one linear scan over the entire input — processing (and potentially modifying) each vertex with a constant number of operations — and uses only a constant amount of space for the state $\langle y_{\mathrm{lo}}, c, h\rangle$. This immediately implies that this algorithm can be implemented as a linear time, oblivious program.

Regarding the correctness of OblSimplify for a set B of vertices and a domain Γ, we let $B_\Gamma := \{\mathrm{box}(b) \mid b \in B \wedge b_x \in \Gamma_{[x]}\}$ be the boxes represented by vertices in Γ and let $D_\Gamma := \{\mathrm{box}(b) \mid b \in B \wedge b_{[x]} \supseteq \Gamma_{[x]}\}$ be the boxes covering Γ. Let B'_Γ be the boxes B_Γ after running the algorithm. By considering the connected components of D_Γ when projected on the y-axis we can easily prove:

Lemma 1. *Given a sequence B of vertices sorted by their y-coordinate and a domain Γ,* OblSimplify *adjusts the y-coordinates and returns a value h so that*

$$\left\|\bigcup(B'_\Gamma)\right\| = \left\|\bigcup(B_\Gamma) \setminus \bigcup(D_\Gamma)\right\| \quad \textit{and} \quad \left\|\bigcup(D_\Gamma)\right\| = h \cdot \mathrm{width}(\Gamma)$$

where all measures are restricted to the domain Γ.

There are two properties of this simple algorithm that are worth pointing out as they are required for the correctness: Firstly note that, for a slab $b \in D$, it does not matter whether we process the left, the right, or both pairs of vertices. Processing both pairs temporarily increases the overlap, but — since both start- and both endpoints have the same y-coordinate — effects neither the anchor point y_{lo} nor the height h. This property ensures the correctness of the complete algorithm since we process both the left and the right vertices of a box before eventually separating them by cutting at a sub-domain boundary.

Secondly, it may be that we adjust a vertex of a slab $b \in D$ in Adjust, i. e., that we adjust a box that previously caused an update in Update. In this case the height of b is reduced to 0. This implies that changes to y_{lo} and c due to the vertices of b in any subsequent invocation of OblSimplify have no effect; for the analysis we can thus assume that the vertices b with $b_x \in \Gamma_{[x]} \subseteq b_{[x]}$ are removed. We stress that no vertices are ever actually removed.

2.5 Oblivious Algorithm

While OblSimplify realizes an oblivious simplification, it still requires the vertices to be sorted by their y-coordinate. This is necessary in each recursive call of the divide-and-conquer algorithm. Chan's RAM algorithm pre-sorts the coordinates in each dimension and maintains these orders, e. g., with a doubly-linked list over the boxes for each dimension. This is possible since both Simplify and Cut do not effect the relative order. For $d = 2$, another option would be to stably partition the vertices for the recursive calls.

Neither can be done efficiently in the ORAM model: The transformation lower bound mentioned in the introduction implies a $\Omega(\log n)$ lower bound on random access to n elements, barring us from efficiently maintaining links. This makes a direct implementation of Chan's approach inefficient. Unfortunately, there also is an $\Omega(n \log n)$ lower bound on stable partitioning n elements (assuming indivisibility) [22]; this implies that — unlike in the RAM model — stable partitioning is no faster than sorting.

As we will show in Sect. 4, this challenge is not unique to this algorithm and in fact arises for many geometric divide-and-conquer algorithms. Our approach — instead of maintaining the order — is to re-sort the vertices by y-coordinate in each recursive call. To make up for at least some part of this runtime overhead, we increase the number of recursive calls from 2 to m (a value to be determined below). Algorithm 3 shows our general outline for the combined cutting and simplification with $m \geq 2$ recursive calls. Remember that in contrast to Chan's algorithm, we simplify immediately after cutting.

To determine the boundaries of the m sub-domains $\Gamma_1, \ldots, \Gamma_m$, we first sort the vertices by their x-coordinate. We ensure that both left vertices of a box always end up in the same sub-domain, same for the right vertices. To avoid leaking information about the input via the runtime of the recursive calls, the number of vertices in each sub-domain must only depend on n: Since there may be several vertices with identical x- and y-coordinates, we assign an additional unique identifier to each box. By establishing a total order this allows us to ensure an even distribution of the $n := |B|$ vertices to the m sub-domains, independent

Algorithm 3. Procedure for cutting the given vertices B into m sub-domains and simplifying each sub-domain.

```
1: procedure CutAndSimplify($B, \Gamma$)
2:     sort $B$ by $x$-coordinate                                          $\triangleright\ \mathcal{O}(n \log n)$
3:     det. sub-domains $\Gamma_1, \dots, \Gamma_m$ (boundaries: $B[\lceil \frac{i \cdot |B|}{m} \rceil]_x$ for $1 \leq i < m$)   $\triangleright\ \mathcal{O}(m)$
4:     sort $B$ by $y$-coordinate                                          $\triangleright\ \mathcal{O}(n \log n)$
5:     for each sub-domain $\Gamma_i$ (simultaneously) do                  $\triangleright\ \mathcal{O}(n \cdot m)$
6:         $h_i \leftarrow$ OblSimplify($B, \Gamma_i$)
7:         height($\Gamma_i$) $\leftarrow$ height($\Gamma_i$) $- h_i$
8:     sort $B$ by $x$-coordinate                                          $\triangleright\ \mathcal{O}(n \log n)$
9:     return $\langle \Gamma_1, \dots, \Gamma_m \rangle$
```

of the input configuration. The value for m only depends on n and we only access the sub-domain representations in a non–input-dependent manner; thus we can store the sub-domain representations in $\Theta(m)$ successive memory cells and access them directly without violating the oblivious security requirement.

For the planar case $d = 2$ we consider, it suffices to partition the vertices, there is no need to duplicate or explicitly remove vertices. This follows from the properties of Algorithm 2: The initial domain Γ is recursively divided into sub-domains. Any box b can — on any level of the recursion — partially cover no more than two sub-domains, one for each of its vertical sides. The sub-domains in-between are fully covered, and through the simplification the area covered by b is removed from these sub-domains. If a left or right pair of vertices coincides with the bounds of a sub-domain (and thus no longer forms a left or right box), that box is implicitly removed by reducing its height to 0.

After determining the sub-domains, we can sort the vertices by their y-coordinates to apply OblSimplify for simplification. Afterwards, we reduce the height of Γ_i by the height h_i removed in that sub-domain. Sequentially processing the sub-domains might reduce the height of a slab covering the next sub-domain, thus resulting in an incorrect simplification. To avoid this, we can perform all simplifications simultaneously: We process all vertices $b \in B$, keeping a separate state for each of the m sub-domains. Again, we can store the states in $\Theta(m)$ successive memory cells. In each iteration, we process the current vertex b for each sub-domain and keep track of the single adjustment to the vertex (which we can apply before processing the next vertex). The final sorting step partitions the vertices according to their sub-domain for the recursive calls.

To balance the costs of sorting and simplifying, we set $m := \max(\lceil \log_2 n \rceil, 2)$. The recursion tree then has $\mathcal{O}(\log_m n) = \mathcal{O}(\log n / \log\log n)$ height; this yields $\mathcal{O}(n \log^2 n / \log\log n)$ total runtime. Not only does this improve over an optimal general transformation by a $\mathcal{O}(\log\log n)$ factor, our construction also guarantees (deterministic) identical memory access patterns, implying perfect security.

3 Improved Processing Using Oblivious Data Structures

When processing the m sub-domains individually, the algorithm presented in the previous section spends considerable time processing boxes not intersecting the current sub-domain and repeatedly adjusting vertices in different sub-domains to remove the area covered by the same slab. Here, we show how to avoid both of these, resulting in further improvement over the general transformation.

For our improved algorithm, we apply Algorithm 2 to m slabs simultaneously in a hierarchical manner. As a prerequisite, we first note on how to construct the necessary data structures with perfect and computational security. We then describe a novel, tree-based data structure; using this data structure we construct the improved algorithm and state its runtime and security properties.

3.1 Oblivious Data Structures

The conceptual idea of our oblivious data structure is to model the recursion tree of a divide-and-conquer algorithm up to a certain height. For this, we first sketch how to construct oblivious static binary trees from known oblivious primitives. We then describe how to use the static binary trees to model the divide-and-conquer recursion tree; for this, we will explicitly state some necessary conditions.

Oblivious Static Binary Tree. As observed by Wang et al. [27], it is possible to build efficient oblivious data structures from *position-based* ORAM constructions. Position-based ORAMs are often used as a building block to construct a "full" ORAM [7,24,26] since they are efficient in practice [24,26] and suitable to build ORAM constructions with perfect security [7,9].

To access the elements efficiently, position-based ORAMs assign a (temporary) label to each element. Accessing an element reveals its label, so to hide the access pattern it is necessary to assign a new label every time an element is accessed. The labels for all elements are maintained in a *position map* which — in general — does not fit in the private memory. *Recursive* position-based ORAM schemes address this by recursively storing the position map in a (smaller) position-based ORAM until it fits in private memory [7,9,24,26]. Depending on the construction, this recursion usually leads to an additional logarithmic runtime overhead [7,24,26]. For data structures where the access pattern graph exhibits some degree of predictability the recursion (with the associated overhead) is not necessary [27]. An example are rooted trees with a bounded degree; here, the labels of the $\mathcal{O}(1)$ direct successors can be encoded in the current element.

We build on this insight to construct oblivious static binary trees with perfect security. Wang et al. use *Path ORAM* [24] to represent oblivious binary trees [27]: Their representation allows traversing a rooted path in a (static) binary tree with N nodes and height h in $\mathcal{O}(h \log N)$ time. However, due to the constraints of the employed ORAM construction, they only achieve the (weaker) statistical security and require a super-constant number of private memory cells [24].[2] To

[2] Most statistically-secure ORAM constructions incur a $\mathcal{O}(\log n)$ time overhead to achieve a security failure probability negligible in n [9]. This excludes many ORAM construction with good performance in practice [24,26] for our range of parameters.

achieve perfect security, we apply their technique to a position-based ORAM construction with perfect security, e. g., the construction of Chan et al. [7].[3]

In conclusion, by leveraging the technique of Wang et al. [27], we can obtain perfectly-secure oblivious static binary tree data structures. The runtime properties follow from the ORAM construction of Chan et al. [7].

Lemma 2. *For a static binary tree $\mathcal{T}$ with N nodes and height h, both public parameters, we can construct an oblivious representation with perfect security. Then $\mathcal{T}$ requires $\mathcal{O}(N)$ space, $\mathcal{O}(1)$ private memory cells, and can be constructed in $\mathcal{O}(N \log N)$ time. A rooted path in $\mathcal{T}$ can be traversed in expected (over the random choices) and amortized (over the path traversals) $\mathcal{O}(h \log^2 N)$ time.*

We will refer to $\mathcal{T}$ as an *oblivious binary tree*. Note that for complete binary trees $h = \lceil \log_2(N+1) \rceil$, thus N is effectively the only public parameter.

By relaxing the security requirement to computational security (and assuming the existence of a family of pseudorandom functions with negligible security failure probability), the query complexity stated in Lemma 2 can be reduced by a $\mathcal{O}(\log N)$ factor. For this, the tree nodes are stored as elements in an asymptotically optimal ORAM construction, e. g., *OptORAMa* [4]. This results in an amortized $\mathcal{O}(h \log N)$ time for traversing a rooted path. As will become clear later, we have (for a constant $c > 1$) some small number $N \in \Theta(2^{\log^{1/c} n})$ of nodes and a large number $T \in \mathcal{O}(n^{\mathcal{O}(1)})$ of accesses. For these parameters, we explicitly consider the parameter λ guarding the security failure probability for OptORAMa: Choosing $\lambda \in \Theta(n)$, we can apply a theorem due to Asharov et al. [4, Theorem 7.2] and simultaneously achieve an (amortized) $\mathcal{O}(h \log N)$ runtime and a negligible security failure probability.

Oblivious Processing Tree. We use the oblivious binary tree as a tool to model the recursion tree of what we will refer to as a *scan-based* divide-and-conquer algorithm: an algorithm that — in each invocation — performs a linear processing scan over the input using $\mathcal{O}(1)$ additional space and then stably partitions the input in preparation for some (constant) number a of recursive calls. One example for such an algorithm is the closest-pair algorithm of Bentley and Shamos [5] (see Sect. 4). Overall, these restrictions imply that the algorithm is order-preserving, i. e., the processing order remains the same in all recursive calls.

The challenge when obliviously implementing a scan-based divide-and-conquer algorithm is the $\Omega(n \log n)$ lower bound for stably partitioning n elements [22]. Thus, although conceptually simple, a naive oblivious computation of such a divide-and-conquer algorithm up to some depth h would incur a runtime of $\mathcal{O}(h \cdot n \log n)$ for a given input of size n. To avoid this, we explicitly construct an oblivious tree $\mathcal{T}$ where the nodes correspond to the recursive calls

[3] The state-of-the-art perfectly-secure ORAM construction [8] further improves the runtime by a $\mathcal{O}(\log \log n)$ factor; this improvement is due to a reduction in the recursion depth and therefore does not benefit our application.

of the algorithm, i. e., the nodes of the recursion tree. Each node $v \in \mathcal{T}$ stores the state of the processing step (state(v)) in the corresponding recursive call as well as routing information, i. e., information on how the elements are partitioned for the recursive calls. We then individually trace each element e through the tree, guided by the routing information. For each node v on the path we update state(v) by simulating the processing step with e as the current element.

This approach correctly simulates the recursion since we require the processing step to be linear and sequence of elements processed in each node corresponds to the sequence of elements processed in the corresponding recursive call. Since we traverse paths of length $h \in \mathcal{O}(\log m)$ in a tree with $N \in \mathcal{O}(m)$ nodes, the oblivious binary trees described above yield:

Lemma 3. *Using an oblivious binary tree we can obliviously simulate a scan-based divide-and-conquer algorithm up to depth h, resulting in $m := a^h$ leaves. The processing steps can be applied to $n \in \Omega(m) \cap \mathcal{O}(2^{\log^c m})$ elements, for $c > 1$, in expected $\mathcal{O}(n \log^3 m)$ time for perfect security or $\mathcal{O}(n \log^2 m)$ time for computational security, $\mathcal{O}(m)$ additional space, and $\mathcal{O}(1)$ private memory cells.*

3.2 Simplification in m Sub-domains

We now show how to simplify the m given sub-domains by processing all vertices and sub-domains simultaneously in a single scan. Rather than concentrating on the geometric details, we will derive this from Algorithm 2 in a general way by applying the above processing technique. We use an oblivious binary tree with m leaves as described above: This recreates the recursion structure of the original algorithm (except we simplify before recursing). As in Sect. 2 the sub-domains $\Gamma_1, \ldots, \Gamma_m$ are determined in advance by sorting the vertices.

Each of the m sub-domains Γ_i corresponds to the i-th leftmost leaf ℓ_i in the oblivious binary tree $\mathcal{T}$. Then $\Gamma_{\ell_i} = \Gamma_i$ is the domain of the leaf ℓ_i; the domain Γ_v of an inner node v is exactly the union of sub-domains below v. To efficiently traverse $\mathcal{T}$, we annotate each inner node v with their left and right sub-domains $\Gamma_{\mathrm{L}} = \Gamma_{\mathrm{left}(v)}$ and $\Gamma_{\mathrm{R}} = \Gamma_{\mathrm{right}(v)}$. This allows us to easily identify whether a vertex is contained in or a box covers the left or right sub-domain.

The algorithm is shown in Algorithm 4; see Fig. 2 for an example. To simulate the recursion of the original algorithm, we trace each vertex b through the explicit recursion tree $\mathcal{T}$, adjusting b to remove the area covered by slabs. To simplify the left and right sub-domains before recursing, we keep two separate states per inner node v of the binary recursion tree: s_{L} for simplification in the left and s_{R} for simplification in the right sub-domain of v. After processing all vertices it remains to determine the combined height h_i of the areas removed in each sub-domain Γ_i. For this, we can traverse the paths to each leaf ℓ_i and sum the heights h in the respective states (in s_{L} for $\Gamma_{\ell_i} \subseteq \Gamma_{\mathrm{L}}$ and in s_{R} for $\Gamma_{\ell_i} \subseteq \Gamma_{\mathrm{R}}$).

We now state the correctness of OblCombinedSimplify. For each $i \in \{1, \ldots, m\}$, let $B_i := \{\mathrm{box}(b) \mid b \in B \wedge b_x \in (\Gamma_i)_{[x]}\}$ be the set of boxes represented by the vertices in Γ_i and let $D_i := \{\mathrm{box}(b) \mid b \in B \wedge b_{[x]} \supseteq (\Gamma_i)_{[x]}\}$ be the slabs over Γ_i. Let each B'_i be the boxes B_i after running the algorithm:

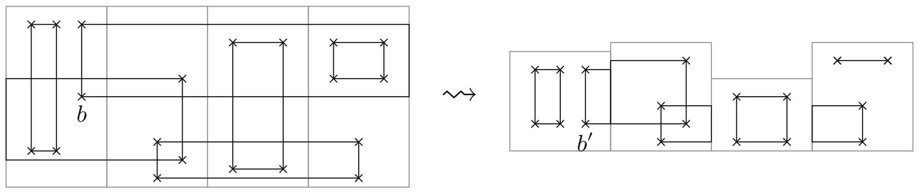

Fig. 2. Simplification of $\Gamma_1, \Gamma_2, \Gamma_3, \Gamma_4$ with 20 vertices. The boundaries are drawn in gray and the vertices are indicated by the marks. Consider processing the vertex b: In the root, b updates the state s_R for the domain $\Gamma_3 \cup \Gamma_4$ and is adjusted according to the initial state s_L. Then b is processed in the left successor where b updates the state s_R for Γ_2 and is adjusted according to s_L for Γ_1; this results in the processed b'.

Algorithm 4. Algorithm to simplify in all sub-domains Γ_i simultaneously. The procedures UPDATE and ADJUST correspond to the sub-routines in Algorithm 2.

```
1:  procedure OBLCOMBINEDSIMPLIFY($B, \Gamma_1, \dots, \Gamma_m$)
2:      $\mathcal{T} \leftarrow$ (complete) oblivious binary tree with $m$ leaves        ▷ recursion tree
3:      for each vertex $b \in B$ do                                                  ▷ in $y$-order
4:          $v \leftarrow$ root($\mathcal{T}$)
5:          do
6:              $\Gamma_\mathrm{L} \leftarrow$ domain for left($v$); $\Gamma_\mathrm{R} \leftarrow$ domain for right($v$)
7:              $\langle s_\mathrm{L}, s_\mathrm{R} \rangle \leftarrow$ state($v$)               ▷ read the simplification states
8:              if box($b$) covers $\Gamma_\mathrm{L}$ then UPDATE($s_\mathrm{L}, b$)      ▷ update the states for
9:              if box($b$) covers $\Gamma_\mathrm{R}$ then UPDATE($s_\mathrm{R}, b$)        covered sub-domains
10:             if $b$ is in $\Gamma_\mathrm{L}$ then ADJUST($b, s_\mathrm{L}$); $v \leftarrow$ left($v$)     ▷ adjust according to
11:             if $b$ is in $\Gamma_\mathrm{R}$ then ADJUST($b, s_\mathrm{R}$); $v \leftarrow$ right($v$)      the respective state
12:             state($v$) $\leftarrow \langle s_\mathrm{L}, s_\mathrm{R} \rangle$              ▷ write back the simplification states
13:         while $v$ is not a leaf
14:     return heights $\langle h_1, \dots, h_m \rangle$ of the boxes removed from each sub-domain
```

Lemma 4. *Given a sequence B of vertices sorted by their y-coordinate and m disjoint sub-domains $\Gamma_1, \dots, \Gamma_m$,* OBLCOMBINEDSIMPLIFY *adjusts the y-coordinates and returns values $h_1, \dots, h_m$ so that*

$$\left\|\bigcup(B'_i)\right\| = \left\|\bigcup(B_i) \setminus \bigcup(D_i)\right\| \quad \text{and} \quad \left\|\bigcup(D_i)\right\| = h_i \cdot \mathrm{width}(\Gamma_i)$$

for each $i \leq m$ where all measures are restricted to the respective domain Γ_i.

Proof (sketch). Let $\widehat{B}$ be a sorted sequence of vertices and $\widehat{B}'$ those vertices adjusted to remove the area covered by some slabs $\widehat{D}$. For the sub-procedures UPDATE and ADJUST we first show that orderly processing UPDATE(s, b) for the vertices $b \in \widehat{B}'$ maintains the height of $\bigcup(\{\mathrm{box}(b) \mid b \in \widehat{B}\}) \setminus \bigcup(\widehat{D})$ in the state s.

We then proceed by induction over the levels of $\mathcal{T}$, starting with the root: We maintain the invariant that, for any node $v \in \mathcal{T}$ and any vertex $b \in \Gamma_v$, $b^v_y = b_y - \mathrm{h}_{\leq b_y}(\bigcup(\bigcap_{\Gamma_i \subseteq \Gamma_v} D_i))$ $(*)$. Here, b^v_y is defined to be the y-coordinate of b before the iteration of the `do-while` loop for v and $\mathrm{h}_{\leq y}(S)$ is the height of $S_{\leq y} \coloneqq S \cap \mathbb{R}\times(-\infty, y]$, i. e., the one-dimensional measure of $S_{\leq y}$ when projected onto the y-axis. This invariant together with the above properties of UPDATE and ADJUST implies that, for any $v \in \mathcal{T}$ with direct successor w, $\textsc{Update}(s, \cdot)$ maintains the height of $\bigcup(D_w) \setminus \bigcup(D_v)$ in the state s for the sub-domain Γ_w $(\star)$.

The first equality of Lemma 4 then follows from $(*)$ for the respective leaf $v = \ell_i$ by considering the bounds of the connected components of $\bigcup(B'_i)$. For the second equality, consider the state s corresponding to each non-root node w (with parent v) after processing the last vertex b: Clearly $s.c = 0$ and $s.h = \mathrm{height}(\bigcup(D_w)) - \mathrm{height}(\bigcup(D_v))$ according to $(\star)$. By summing the heights $s.h$ on the path to each leaf ℓ_i we obtain $h_i = \mathrm{height}(\bigcup(D_i))$. $\square$

With Lemma 3 and since the inner loop in Lines 6 to 12 only accesses a constant number memory cells in the current node, the following properties hold:

Lemma 5. OBLCOMBINEDSIMPLIFY *obliviously simplifies* $m \in \Omega(2^{\log^{1/c} n}) \cap \mathcal{O}(n)$ *sub-domains, for* $c > 1$, *with a total of* n *vertices in expected* $\mathcal{O}(n \log^3 m)$ *time for perfect security or* $\mathcal{O}(n \log^2 m)$ *time for computational security,* $\mathcal{O}(m)$ *space, and* $\mathcal{O}(1)$ *private memory cells.*

3.3 Putting Everything Together

To obtain a faster algorithm for Klee's measure problem, we replace the one-slab-at-a-time simplification (Algorithm 2) with the multi-slab simplification (Algorithm 4) in Algorithm 3. As before, the returned values h_i can be used to update the heights of the sub-domains Γ_i.

Since adjusting the y-coordinates for m sub-domains can be done in expected $\mathcal{O}(n \log^c m)$ time (with $c = 2$ for computational security and $c = 3$ for perfect security), we can balance the cost of sorting and updating the coordinates by choosing $m \coloneqq \max(2^{\lceil \log_2^{1/c} n \rceil}, 2)$. This leads to a recursion tree height of $\mathcal{O}(\log_m n) = \mathcal{O}(\log^{1-1/c} n)$ for the complete algorithm. With the time required for sorting in each recursive call this yields a $\mathcal{O}(n \log^{2-1/c} n)$ runtime overall.

The security of Algorithm 4 immediately follows from the security of the oblivious binary tree $\mathcal{T}$: The algorithm repeatedly traverses rooted paths in $\mathcal{T}$, performing a constant number of operations for each node. Algorithm 3 is secure due to the security of oblivious sorting and the fact that the sub-domain access is independent of the input. With a total order over the boxes, the input sizes for the recursive calls solely depend on the problem size n. The base case for $\mathcal{O}(1)$ boxes with runtime $\mathcal{O}(1)$ can trivially be transformed into an oblivious algorithm. Since each recursive call processes the boxes in an oblivious manner, the obliviousness of the full divide-and-conquer algorithm follows.

Theorem 1. *There is an oblivious algorithm solving Klee's measure problem for* $d = 2$ *in expected* $\mathcal{O}(n \log^{5/3} n)$ *time for perfect security or* $\mathcal{O}(n \log^{3/2} n)$ *time for computational security,* $\mathcal{O}(n)$ *additional space, and* $\mathcal{O}(1)$ *private memory cells.*

4 General Technique

Above we showed how to use the oblivious binary tree to construct an efficient algorithm for Klee's measure problem. The technique is not specific to Chan's algorithm and can be applied to other geometric divide-and-conquer algorithms as well. In this section, we will outline the necessary conditions for the application of our technique. As an illustration, we will also sketch the application to two other problems, namely inversion counting and the closest-pair problem for $d = 2$. For both problems we will focus on the linear processing step; the runtime — both for perfect and computational security — then immediately follows as for Klee's measure problem above.

Our technique for transforming divide-and-conquer algorithms into oblivious counterparts requires that the following conditions are met:

(a) The input sizes for the recursive calls must only depend on the problem size.
(b) Each element must be contained in the input to at most one recursive call.
(c) Within each recursive call, the elements are processed with a linear scan using $\mathcal{O}(1)$ additional memory cells.

The first condition (a) is necessary for security, otherwise the algorithm might leak information via the runtime of the recursive calls. It is often possible to accommodate small size variations without affecting the runtime by padding the inputs with dummy elements. The conditions (b) and (c) are necessary to individually trace the elements through the recursion tree. This individual tracing also limits the information that can be passed to the recursive calls.

Inversion Counting. The *inversion counting* problem — for a sequence A of length n — is to determine the number of indices i, j so that $i < j$ and $A[i] > A[j]$. In the RAM model, this problem can be solved in $\mathcal{O}(n \log n)$ time by an augmented merge-sort algorithm [14, Exercise 2-4 d].[4] Since merging is a linear-time operation in the RAM model, but has an $\Omega(n \log n)$ lower bound in the oblivious RAM model (assuming indivisibility) [22], a direct implementation of this approach results in an $\mathcal{O}(n \log^2 n)$ time oblivious algorithm.

To improve over this, we interpret this algorithm as scan-based divide-and-conquer algorithm: Since the RAM algorithm is a divide-and-conquer algorithm, it remains to describe how to obliviously count inversion pairs using linear scans. For this, note that it is possible to separate the merging and counting steps by marking each element. Counting inversions can then be done in a linear scan by counting the elements from the second half and adding the current count to the number of inversions for every encountered element from the first half.

[4] In the word RAM model of computation there is an $\mathcal{O}(n \log^{1/2} n)$ time algorithm due to Chan and Pătraşcu [13]. For computational security, it is thus also possible to obtain an $\mathcal{O}(n \log^{3/2} n)$ time oblivious algorithm through optimal oblivious transformation.

For more efficient processing we split the input into m parts and recurse for each part. We then annotate each element with its part, sort the elements, and count inversions between the m parts using an oblivious binary tree: The annotated parts identify the leafs and each node has a counter for the elements belonging to a leaf in the right subtree. For the final number of inversions we sum the inversions counted in all nodes and in the recursive calls.

Closest Pair. For the planar *closest-pair* problem, we are given a set of n points $P \subset \mathbb{R}^2$ and want to determine a pair $p, q \in P$ with minimal distance according to some metric d. For an oblivious algorithm, we apply our technique to the divide-and-conquer algorithm of Bentley and Shamos [5].[5]

As for the inversion counting above, we begin by describing the necessary modifications to obtain a scan-based divide-and-conquer algorithm: After partitioning the points into P_{L} and P_{R} according to the median x-coordinate and recursing, we have minimal distances δ_{L} and δ_{R}; let $\delta := \min\{\delta_{\mathrm{L}}, \delta_{\mathrm{R}}\}$. It remains to check whether there is a pair in $P_{\mathrm{L}} \times P_{\mathrm{R}}$ with a smaller distance: While the original algorithm performs a synchronized traversal over P_{L} and P_{R}, we need to slightly modify this to adhere to the requirement that we may only perform a linear scan over the complete input. We thus sort the points by increasing y-coordinate and perform a linear scan over the sorted sequence. Using a standard packing argument [5], we maintain a queue Q of constant size that stores all points within distance at most δ from the median x-coordinate and within distance of at most δ below the current point. Whenever we encounter a point p with $|s_x - p_x| \geq \delta$ where s_x is the median x-coordinate, we simply ignore this point; otherwise, we check p against all points currently in Q to see whether $\mathrm{d}(p, q) < \delta$ for any point q on the other side of s_x. If we find such a point, we update δ to $\mathrm{d}(p, q)$. We then put p into Q while pruning all points too far below p or away from s_x.

Again, we can utilize an oblivious binary tree by sorting the input according to the x-coordinate and splitting evenly into m parts before recursing. We additionally store the separating x-coordinate s_x in each node of the binary tree; then we sort the elements by y-coordinate and process them so that for each node v the state consists of the queue Q (of constant size) as well as the current closest pair. Finally, we iterate over all nodes, updating the closest pair as needed.

5 Conclusion and Future Work

We gave an efficient oblivious modification of Chan's algorithm [12] for Klee's measure problem for $d = 2$, both for perfect and computational security.

[5] Eppstein et al. [15] discuss how to obliviously compute a closest pair in the plane in $\mathcal{O}(n \log n)$ time through an efficient construction of a well-separated pair decomposition [10]. For this, the input points need to have integer coordinates or a bounded spread. In contrast, our algorithm works without any such assumptions.

For $d \geq 3$, we sketched how to maintain the runtime of the original algorithm. Our oblivious algorithms only require $\mathcal{O}(1)$ private memory cells and can be used to construct a protocol for multi-party computation. We constructed our results with a general technique for oblivious divide-and-conquer algorithms and demonstrated its generality by applying it to the inversion counting and closest-pair problems.

Some open problems remain: Most notably, is it possible to obliviously solve Klee's measure problem for $d = 2$ in $\Theta(n \log n)$ time? From a more general perspective, faster oblivious binary tree implementations would not only improve our algorithm, but also a more general class of divide-and-conquer algorithms using our technique. Designing efficient oblivious tree data structures thus remains an interesting open problem.

Acknowledgments. We thank the reviewers for their constructive comments that helped to improve the presentation.

References

1. Ajtai, M., Komlós, J., Szemerédi, E.: An $\mathcal{O}(n \log n)$ sorting network. In: Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing, pp. 1–9 (1983). https://doi.org/10.1145/800061.808726
2. Asharov, G., Chan, T.H.H., Nayak, K., Pass, R., Ren, L., Shi, E.: Bucket oblivious sort: an extremely simple oblivious sort. In: Symposium on Simplicity in Algorithms (SOSA), pp. 8–14 (2020). https://doi.org/10.1137/1.9781611976014.2
3. Asharov, G., Komargodski, I., Lin, W.K., Nayak, K., Peserico, E., Shi, E.: OptORAMa: optimal oblivious RAM. Cryptology ePrint Archive, Report 2018/892 (2018). https://ia.cr/2018/892
4. Asharov, G., Komargodski, I., Lin, W.K., Nayak, K., Peserico, E., Shi, E.: OptORAMa: optimal oblivious RAM. In: Advances in Cryptology - EUROCRYPT 2020, pp. 403–432 (2020). https://doi.org/10.1007/978-3-030-45724-2_14
5. Bentley, J.L., Shamos, M.I.: Divide-and-conquer in multidimensional space. In: Proceedings of the Eighth Annual ACM Symposium on Theory of Computing, pp. 220–230 (1976). https://doi.org/10.1145/800113.803652
6. Blanton, M., Steele, A., Alisagari, M.: Data-oblivious graph algorithms for secure computation and outsourcing. In: Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security, pp. 207–218 (2013). https://doi.org/10.1145/2484313.2484341
7. Chan, T.H.H., Nayak, K., Shi, E.: Perfectly secure oblivious parallel RAM. In: Theory of Cryptography, pp. 636–668 (2018). https://doi.org/10.1007/978-3-030-03810-6_23
8. Chan, T.H.H., Shi, E., Lin, W.K., Nayak, K.: Perfectly oblivious (parallel) RAM revisited, and improved constructions. Cryptology ePrint Archive, Report 2020/604 (2020). https://ia.cr/2020/604
9. Chan, T.H.H., Shi, E., Lin, W.K., Nayak, K.: Perfectly oblivious (parallel) RAM revisited, and improved constructions. In: 2nd Conference on Information-Theoretic Cryptography (ITC 2021), pp. 8:1–8:23 (2021). https://doi.org/10.4230/LIPIcs.ITC.2021.8

10. Chan, T.M.: Well-separated pair decomposition in linear time? Inf. Process. Lett. **107**(5), 138–141 (2008). https://doi.org/10.1016/j.ipl.2008.02.008
11. Chan, T.M.: A (slightly) faster algorithm for Klee's measure problem. Comput. Geom. **43**(3), 243–250 (2010). https://doi.org/10.1016/j.comgeo.2009.01.007
12. Chan, T.M.: Klee's measure problem made easy. In: 2013 IEEE 54th Annual Symposium on Foundations of Computer Science, pp. 410–419 (2013). https://doi.org/10.1109/FOCS.2013.51
13. Chan, T.M., Pătraşcu, M.: Counting inversions, offline orthogonal range counting, and related problems. In: Proceedings of the 2010 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 161–173 (2010). https://doi.org/10.1137/1.9781611973075.15
14. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, 3rd edn. MIT Press, Cambridge (2009)
15. Eppstein, D., Goodrich, M.T., Tamassia, R.: Privacy-preserving data-oblivious geometric algorithms for geographic data. In: Proceedings of the 18th SIGSPATIAL International Conference on Advances in Geographic Information Systems, pp. 13–22 (2010). https://doi.org/10.1145/1869790.1869796
16. Fredman, M.L., Weide, B.: On the complexity of computing the measure of $\bigcup[a_i, b_i]$. Commun. ACM **21**(7), 540–544 (1978). https://doi.org/10.1145/359545.359553
17. Goldreich, O., Ostrovsky, R.: Software protection and simulation on oblivious RAMs. J. ACM **43**(3), 431–473 (1996). https://doi.org/10.1145/233551.233553
18. Goodrich, M.T.: Data-oblivious external-memory algorithms for the compaction, selection, and sorting of outsourced data. In: Proceedings of the Twenty-Third Annual ACM Symposium on Parallelism in Algorithms and Architectures, pp. 379–388 (2011). https://doi.org/10.1145/1989493.1989555
19. Guerreiro, A.P., Fonseca, C.M., Paquete, L.: The hypervolume indicator: computational problems and algorithms. ACM Comput. Surv. **54**(6), 1–42 (2021). https://doi.org/10.1145/3453474
20. Klee, V.: Can the measure of $\bigcup_1^n[a_i, b_i]$ be computed in less than $\mathcal{O}(n \log n)$ steps? Am. Math. Mon. **84**(4), 284–285 (1977). https://doi.org/10.1080/00029890.1977.11994336
21. Larsen, K.G., Nielsen, J.B.: Yes, there is an oblivious RAM lower bound! In: Advances in Cryptology - CRYPTO 2018, pp. 523–542 (2018). https://doi.org/10.1007/978-3-319-96881-0_18
22. Lin, W.K., Shi, E., Xie, T.: Can we overcome the $n \log n$ barrier for oblivious sorting? In: Proceedings of the 2019 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 2419–2438 (2019). https://doi.org/10.1137/1.9781611975482.148
23. Overmars, M.H., Yap, C.K.: New upper bounds in Klee's measure problem. SIAM J. Comput. **20**(6), 1034–1045 (1991). https://doi.org/10.1137/0220065
24. Stefanov, E., et al.: Path ORAM: an extremely simple oblivious RAM protocol. In: Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security, pp. 299–310 (2013). https://doi.org/10.1145/2508859.2516660
25. Wang, G., Luo, T., Goodrich, M.T., Du, W., Zhu, Z.: Bureaucratic protocols for secure two-party sorting, selection, and permuting. In: Proceedings of the 5th ACM Symposium on Information, Computer and Communications Security, pp. 226–237 (2010). https://doi.org/10.1145/1755688.1755716

26. Wang, X., Chan, H., Shi, E.: Circuit ORAM: on tightness of the Goldreich-Ostrovsky lower bound. In: Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security, pp. 850–861 (2015). https://doi.org/10.1145/2810103.2813634
27. Wang, X.S., et al.: Oblivious data structures. In: Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security, pp. 215–226 (2014). https://doi.org/10.1145/2660267.2660314