Oblivious Median Slope Selection

Thore Thießen∗ Jan Vahrenhold∗

Abstract

We study the median slope selection problem in the oblivious RAM model. In this model memory accesses have to be independent of the data processed, i.e., an adversary cannot use observed access patterns to derive additional information about the input. We show how to modify the randomized algorithm of Matoušek [27] to obtain an oblivious version with \( O(n \log^2 n) \) expected time for \( n \) points in \( \mathbb{R}^2 \). This complexity matches a theoretical upper bound that can be obtained through general oblivious transformation. In addition, results from a proof-of-concept implementation show that our algorithm is also practically efficient.

1 Introduction

Data collected for statistical analysis is often sensitive in nature. Given the increasing reliance on cloud-based solutions for data processing, there is a demand for data-processing techniques that provide privacy guarantees. One such guarantee is obliviousness, i.e., an algorithm’s property to have externally observable runtime behavior that is independent of the data being processed. Depending on the runtime behavior observed, oblivious algorithms can be used to perform privacy-preserving computations on externally stored data or mitigate side channel attacks on shared resources [33, 26].

In the oblivious RAM model of computation [14, 15] algorithms need to be oblivious with respect to the memory access patterns; we refer to memory-access obliviousness as obliviousness. In general this leads to an \( \Omega(\log m) \) overhead compared to RAM algorithms when operating on \( m \) memory cells [15, 22, 18]. A transformation approach matches this lower bound asymptotically [5], but is known to result in prohibitively large constant runtime overhead.

The median slope, know as the Theil–Sen estimator, is a linear point estimator that is robust against outliers [31]. The randomized algorithm of Matoušek [27] computes the median slope of \( n \) points in \( \mathbb{R}^2 \) with expected runtime \( O(n \log n) \) and is fast in practice. We derive an oblivious version of Matoušek’s algorithm that is slower by a logarithmic factor — matching the complexity obtainable through general transformation — but still fast in practice.

1.1 Median slope selection problem

Median slope selection is a special case of the general slope selection problem: Given a set of points \( P \) in the plane, the slope selection problem for an integer \( k \) is to select a line with \( k \)-th smallest slope among all lines through points in \( P \) [10]. Formally, given a set of \( n \) points \( P \subset \mathbb{R}^2 \) let \( L := \{ \{p, q\} \subseteq P \mid p_x \neq q_x \} \) be the set of all pairs of points from \( P \) with distinct \( x \)-coordinates. We use \( \ell_{pq} \in L \) to denote the line through points \( \{p, q\} \in L \). No line in \( L \) is vertical by definition, so the slope \( m(\ell_{pq}) \) is well-defined for all \( \ell_{pq} \in L \). Let \( k \) be an integer with \( k \in [|L|] := \{0, \ldots, |L| - 1\} \). The slope selection problem for \( k \) then is to select points \( \{p, q\} \in L \) such that \( \ell_{pq} \) has a \( k \)-th smallest slope in \( L \).

Unless noted otherwise and in line with Matoušek [27] our exposition assumes that the points \( P \) are in general position: All \( x \)-coordinates of points \( \{p, q\} \subseteq P \) are distinct and all lines through different pairs of points have different slopes. For simplicity we also assume that \( |L| \) is odd, so that the median slope can be determined by solving the slope selection problem for \( k = \lfloor |L| \rfloor - \frac{1}{2} \). In Section 3, we discuss how to lift these restrictions.

Matoušek’s algorithm approaches the slope selection problem by considering the dual intersection selection problem [27]: Each point \( p = (p_x, p_y) \in P \) can be mapped to dual non-vertical line \( \overline{p} : x \mapsto (p_x, x - p_y) \) and vice versa. Since we have \( \overline{p(x)} = \overline{p(y)} = y \iff \ell_{pq} = \ell_{pq}, \) a point in the set of (dual) intersection points with \( k \)-th smallest \( x \)-coordinate is dual to a line in \( L \) with \( k \)-th smallest slope [27, 11].

We thus restrict ourselves to finding an intersection of dual lines \( \overline{P} \) with \( k \)-th smallest \( x \)-coordinate. By the above assumption regarding the general position of the points in \( P \), the lines in \( \overline{P} \) have distinct slopes and all intersection points have distinct \( x \)-coordinates.

1.2 Oblivious RAM model

We work in the oblivious RAM (ORAM) model [14, 15]. This model is concerned with what can be derived by an adversary observing the memory access patterns during the execution of a program. The general requirement is that memory accesses are (data-) oblivious, i.e., that the

∗Westfälische Wilhelms-Universität Münster, Dept. of Computer Science {t.thießen, jan.vahrenhold}@uni-muenster.de
adversary can learn nothing about the input (or output) from the memory access pattern.

In line with standard assumptions, we assume a probabilistic word RAM with word length $w$, a constant number of registers in the processing unit and access to $m \leq 2^w$ memory cells with $w$ bits each in the memory unit [18]. The constant number of registers in the processing unit are called private memory and do not have to be accessed in an oblivious manner.

Whether a given probabilistic RAM program $R$ operating on inputs $X$ is oblivious depends on the way memory is accessed. Let $A := \{\text{read}, \text{write}\} \times [m]$ be the set of memory probes observable by the adversary. Each probe is identified by the memory operation and the access location $i \in [m]$. The random variable $A_R(x)$: $\Omega \rightarrow A^*$ denotes the probe sequence performed by $R$ for an input $x \in X$ where $\Omega := \{0,1\}^{\leq w}$ is the set of possible random tape contents. The program $R$ is secure if no adversary, given inputs $x, x' \in X$ of equal length and a probe sequence $A \in A^*$, can reliably decide whether $A$ was induced by $x$ or $x'$. For a program with an output determined by the input this implies that no adversary can decide between given outputs [3].

We operationalize obliviousness by restricting the definition of Chan et al. [9] to perfect security, determined programs, and perfect correctness. The definition also generalizes the allowed dependence of the probe sequence on the length of the input to a general leakage; the leakage determines what information the adversary may be able to derive from the memory access patterns.

**Definition 1: Oblivious simulation.** Let $f: X \rightarrow Y$ be a computable function and let $R$ be a probabilistic RAM program. $R$ obliviously simulates $f$ with regard to leakage $\text{leak}: X \rightarrow \{0,1\}^*$ if $R$ is correct, i.e., for all inputs $x \in X$ the equality $\Pr[R(x) = f(x)] = 1$ holds, and if $R$ is secure, i.e., for all inputs $x, x' \in X$ with $\text{leak}(x) = \text{leak}(x')$ the equality $\sum_{A \in A^*} |\Pr[A_R(x) = A] - \Pr[A_R(x') = A]| = 0$ holds.

The composition of oblivious programs is also oblivious if the sub-procedures invoke each other in an oblivious manner; see Appendix A for a more detailed discussion and proof of compositability. Here relaxing the leakage allows us to place fewer restrictions on sub-procedures while maintaining obliviousness of the complete program.

For the specific problem in this paper, the algorithm is only allowed to leak the number of given lines, or, for subroutines, the length of each given input array. We will prove the obliviousness of our algorithm by compositability, so we will consider the obliviousness of sub-procedures individually. In line with Definition 1 we will show the obliviousness of each procedure in relation to the input. Since we only consider sub-procedures with determined result this implies the obliviousness in relation to the output.

## 1.3 Related work

There exists a breadth of research on the slope selection problem. Cole et al. [10] prove a lower bound of $\Omega(n \log n)$ for the general slope selection problem in the algebraic decision tree model that also holds in our setting (see Appendix B). Both deterministic algorithms [10, 19, 8] and randomized [27, 11] algorithms have been proposed that achieve an $O(n \log n)$ (expected) runtime. The problem has also been considered in other models, see, e.g., in-place algorithms [7].

Asharov et al. [5] recently proposed an asymptotically optimal ORAM construction that matches the overhead factor of $\Omega(\log m)$ per memory access. This construction provides a general way to transform RAM programs into oblivious variants with no more than logarithmic overhead per memory operation. Due to large constants this optimal oblivious transformation is not viable in practice, though practically efficient (yet asymptotically suboptimal) constructions are available, see, e.g., Path ORAM [34]. Our algorithm matches the asymptotic runtime of an optimal transformation while maintaining practical efficiency and perfect security.

A different approach is the design of problem-specific algorithms without providing general program transformations. Oblivious algorithms for fundamental problems have been considered, e.g., for sorting [17, 3], sampling [29, 32], database joins [1, 23, 21], and some geometric problems [13]. To the best of our knowledge neither the slope selection problem nor the related intersection counting problem have been considered in the oblivious setting before.

## 2 A simple algorithm

As mentioned above, our approach is to modify the randomized algorithm proposed by Matoušek [27]. For this, we replace all non-trivial building blocks of the original algorithm — most notably intersection counting and intersection sampling — by oblivious counterparts.

### 2.1 The original algorithm

Algorithm 1 shows the original algorithm as described by Matoušek [27]. In a nutshell the algorithm works by maintaining intersections $a$ and $b$ as lower and upper bounds for the intersection $p_k$ with $k$-th smallest $x$-coordinate to be identified.\(^2\)

In the main loop a randomized interpolating search is performed, tightening the bounds $a$ and $b$ until only $N \in O(n)$ intersections remain in between. For this, a multiset $R$ of $n$ intersections is sampled from the remaining intersections (with replacement) in each iteration. Then new bounds $a'$ and $b'$ are selected from $R$.

\(^2\)Generalizing the description of the algorithm [27] we maintain the intersections $a, b$ instead of only their $x$-coordinates.
Algorithm 1 Randomized intersection selection algorithm [27].

1: function IntSelection(\(P, k\))
2: \(n \leftarrow |P|; N \leftarrow \text{IntCount}(P, a, b)\) ⟹ Number of input lines and of remaining intersections
3: \(a \leftarrow -\infty; b \leftarrow +\infty\)
4: \(\text{do}
5: \quad j \leftarrow n \cdot (k - \text{IntCount}(P, -\infty, a) + 1) / N - 1
6: \quad j_a \leftarrow \max\{0, \lfloor j - 3\sqrt{n} \rfloor \}; j_b \leftarrow \min\{n - 1, \lceil j + 3\sqrt{n} \rceil \}
7: \quad R \leftarrow \text{IntSample}(P, a, b, n)\) ⟹ Sample intersection points
8: \quad a' \leftarrow \text{Select}_{x}(R, j_a); b' \leftarrow \text{Select}_{x}(R, j_b)\) ⟹ Select candidate boundaries for next iteration
9: \quad m_a' \leftarrow \text{IntCount}(P, -\infty, a'); m_b' \leftarrow \text{IntCount}(P, -\infty, b')\) ⟹ Count intersections left of \(a'\) and left of \(b'\)
10: \quad \text{if } m_a' \leq k < m_b' \land m_b' - m_a' \leq 11N / \sqrt{n}\) then
11: \quad \quad a, b \leftarrow a', b'\)
12: \quad \quad N \leftarrow m_b' - m_a'\)
13: \quad \text{while } N > n\)
14: \quad R \leftarrow \text{IntEnumeration}(P, a, b)\) ⟹ Enumerate all remaining intersections
15: \quad \text{return } \text{Select}_x(R, k - \text{IntCount}(P, -\infty, a))\) ⟹ Select correct intersection

2.2 Known oblivious building blocks

Sorting In the ORAM model, \(n\) elements can be sorted by a comparison-based algorithm in optimal \(\Theta(n \log n)\) time, e.g., using optimal sorting networks [2]. We refer to this building block as \textbf{Sort}.

For the application in this paper we require a sorting algorithm which is fast in practice. To this end we can use \textit{bucket oblivious sort} [3]: The algorithm works by performing an oblivious random permutation step, followed by a comparison-based sorting step. The random permutation ensures that the complete algorithm is oblivious, even if the sorting step is not.

Choosing suitable parameters (\(Z \in \Theta(\log n)\)) we achieve a failure probability bounded by a constant [3, Lemma 3.1]. Since failure of the random permutation leaks nothing about the input, we can repeat this step until it succeeds. Together with an optimal comparison-based sorting algorithm this results in an implementation for \textbf{Sort} that has an expected runtime of \(O(n \log n \log^2 \log n)\) and is fast in practice.

Merging The building block \textbf{Merge}(\(A, B\)) takes two individually sorted arrays \(A\) and \(B\) and sorts the concatenation \(A \parallel B\). There is a lower bound of \(\Omega(n \log n)\) for merging in the indivisible oblivious RAM model.\(^3\) \textit{Odd-even merge} [6, 20] is an optimal merge algorithm (in the indivisible oblivious RAM model) with a good performance in practice.

Selection \textbf{Select}(\(A, k\)) denotes the selection of an element with rank \(k\), i.e., a \(k\)-th smallest element, from an unordered array \(A\). An optimal algorithm in the RAM model is Blum’s linear-time selection algorithm. This problem can be solved by a near-linear oblivious algorithm [25], but current implementations suffer from high constant runtime factors due to the use of oblivious partitioning. For practical efficiency, we realize selection by sorting the given array \(A\). Since for our application we may leak the index \(k\), only one additional probe is required. We thus have leakage \textbf{leak}: \(\langle A, k \rangle \mapsto \langle |A|, k \rangle\).

Filtering Filtering a field \(A\) with a predicate \textbf{Pred}(\(\text{Filter}\)) extracts a sorted sub-list \(A'\) with all elements for which the predicate is true. The elements \(a \in A'\) are stable swapped to the front of \(A\) and the number \(|A'\|\) of such elements is returned. Since filtering can be used to realize stable partitioning, the lower runtime bound of \(\Omega(n \log n)\) for inputs of length \(n\) in the indivisible ORAM model [25] applies. This operation can be implemented with runtime \(O(n \log n)\) using oblivious routing networks [16].

\(^3\)Lin, Shi, and Xie [24] prove a lower bound of \(\Omega(n \log n)\) for stable partition in the indivisible oblivious RAM model that also applies to merging. This bound applies even when restricting the input to arrays of (nearly) equal size [28].
3.2 New oblivious building blocks

3.2.1 Inversion and intersection counting

The number of inversions in an array \( A \) is defined as the number of pairs of indices \( i, j \in |A| \) with \( A[i] > A[j] \) and \( i < j \). In the RAM model, an optimal comparison-based approach to determine the number of inversions is a modified merge sort. Our oblivious merge-based inversion counting \( \text{Inversions} \) generalizes this to an arbitrary merge algorithm (with indivisible keys).

As noted by Cole et al. [10], inversion counting can be used to calculate the number of intersections of a set of lines in a given range \([a_x, b_x]\). This is by ordering the lines according to the \( x \)-coordinates and counting inversions relative to the order at \( x = a_x \) \((\leq a)\) and \( x = b_x \) \((\leq b)\). We use this to implement \( \text{IntCount}(P, a, b) \) for determining the number of intersections of lines \( P \).

**Algorithm 2 Intersection counting.**

1: function \( \text{IntCount}(P, a, b) \)
2: \( P_a \leftarrow \text{Sort}_a(P) \) \( \triangleright \) Sort according to \( \leq a \)
3: return \( \text{Inversions}_a(P_a) \) \( \triangleright \) Count inversions

The building block \( \text{Append}(A, B, i, k) \) is given two fields \( A \) and \( B \) as well as two indices \( i \) and \( k \) and appends the first \( k \) elements of \( B \) to the first \( i \) elements of \( A \). This ensures that \( A' \) after the operation contains \( A[0 : i] \| B[0 : k] \) in the first \( i + k \) positions. All other positions may contain arbitrary elements. This operation can also be implemented with runtime \( O(n \log n) \) by using oblivious routing networks.

To do this obliviously, \( \text{BiInversions} \) labels the elements according to which half they come from, then merges the labeled elements, and finally uses these labels to simulate the standard RAM merging algorithm. For this algorithm to work correctly, in general a stable merge algorithm is required, which sorts elements from the first half before elements from the second half if they are equal with regard to the order. We can drop this requirement since we only work on totally ordered inputs of unique elements.

The correctness of inversion counting follows from the correctness of \( \text{BiInversions} \). Independent of the particular merge algorithm used, \( \text{BiInversions} \) is functionally equivalent to the merging step of the RAM algorithm. The runtime of \( \text{BiInversions} \) is dominated by merging, thus \( \text{Inversions} \) runs in time \( O(n \log^2 n) \). As merging has a lower bound of \( \Omega(n \log n) \) in the indivisible ORAM model and, even without assuming indivisibility, no ORAM algorithm with runtime \( o(n \log n) \) is known, any divide-and-conquer approach based on 2-way merges currently incurs a runtime of \( \Omega(n \log^2 n) \).

Except for the invocation of \( \text{Merge} \), all operations in \( \text{BiInversions} \) can be realized obliviously by a constant number of linear scans over the elements \( A \) and their labels. Since \( \text{Merge} \) is oblivious, the obliviousness of \( \text{BiInversions} \) follows from the composability of oblivious programs. The obliviousness of \( \text{Inversions} \) again follows from composability. Finally, since the input is divided depending only on the size of the input, \( \text{Inversions} \) and \( \text{IntCount} \) only leak the input size.

**Defining a suitable order** Intuitively, the algorithm sorts the input (lines sorted according to \( \leq a \)) while recording intersection points. At each such point, two lines adjacent in the underlying order exchange their position. In addition to handling boundary cases correctly, it is not immediately obvious how this approach can be modified to handle non-general positions, since there may be an arbitrary number of lines intersecting in a single point.

To be able to handle non-general positions obliviously, we do not explicitly use the \( y \)-coordinates to define \( \leq a \) and \( \leq b \). Instead, we — more generally — order the lines by their intersection points in relation to a given intersection \( p \). For this, we use \( p_{x, y} := \ell_x \cap \ell_y \) to denote the intersection point of two lines \( \ell_x \neq \ell_y \).

**Definition 2.** Let \( P_x := T \cup \{-\infty, +\infty\} \) be the set of all intersections formed by \( P \) with additional elements \(-\infty\) and \(+\infty\). Let also \( \preceq \) be an order over \( P_x \) (with the corresponding strict order \(<\)). For each \( p \in P_x \), we define the binary relation \( \preceq_p \) over \( P \) as

\[
\ell_1 \preceq_p \ell_2 := \begin{cases} 
T & \text{if } \ell_1 = \ell_2 \\
\ell_1 < p & \text{if } m(\ell_1) > m(\ell_2) \\
\ell_1 > p & \text{if } m(\ell_1) < m(\ell_2)
\end{cases}
\]
For lines in general position, this definition essentially captures the ordering by $y$-coordinate: If the slope of $\ell_1$ is larger than the slope of $\ell_2$, $\ell_1$ lies below $\ell_2$ if their intersection point lies to the right of $p$; if the slope of $\ell_1$ is smaller than the slope of $\ell_2$, $\ell_1$ lies above $\ell_2$ if and only if their intersection point lies to the right.

**Lemma 1: Correctness of IntCount.** Let $P_x$, $\preceq$, $\prec$, and $\leq_p$ be as defined above. If

(a) $\preceq$ is a total order over $P_x$ with minimum $-\infty$ and maximum $+\infty$ and

(b) $\leq_p$ is a total order over $\mathcal{P}$ for all $p \in P_x$,

then, given $a, b \in P_x$ with $a \preceq b$, IntCount determines the number of intersections $p \in \mathcal{P}$ with $a \leq_p b$.

**Proof.** IntCount sorts according to the order $\leq_a$ and then counts inversions according to the order $\leq_b$. The algorithm thus exactly counts the number of unique pairs $\{\ell_1, \ell_2\} \subseteq \mathcal{P}$ (assuming w.l.o.g. $m(\ell_1) > m(\ell_2)$) for which $(\ell_1 \preceq_a \ell_2) \neq (\ell_1 \preceq_b \ell_2)$. Since $\preceq$ is a total order and $a \preceq b$ this can only occur if $\ell_1 \preceq_a \ell_2 \land \ell_1 \preceq_b \ell_2$. Then $a \preceq_{p_{x2}} b$ follows directly from the definition of $\leq_p$, thus IntCount counts exactly the number of intersections in the range $[a, b)$.

Since we want to identify the intersection with median $x$-coordinate, the intersections need to be ordered primarily by their $x$-coordinate. If all intersection points have distinct $x$-coordinates — which is the case for lines $\mathcal{P}$ in general position — we have:

**Remark 1.** Let $\mathcal{P}$ be in general position and $\preceq$ be defined as $p \preceq q :\iff p_x \leq q_x$ for $p, q \in P_x$ with special cases $-\infty \preceq p$ and $p \preceq +\infty$ for all $p \in P_x$. Then both conditions in Theorem 1 are satisfied.

We will prove this more generally in Section 3.

The intersection point of two given lines can be determined in constant time, so $\leq_p$ can be evaluated in constant time as well. As such the runtime of IntCount is dominated by Inversions and thus $\mathcal{O}(n \log^2 n)$ for $n$ given lines. The method is oblivious by compositability.

### 2.3.2 Intersection sampling and enumeration

The last building blocks to consider are the independent sampling as well as the enumeration of intersection points from a given range $[a, b]$. We need to avoid calculating all intersections explicitly, as this would result in a runtime of $\mathcal{O}(n^2)$. Recall that sampling can be done efficiently in the RAM model by modifying the standard intersection counting algorithm: First, a set $K$ of $k$ indices from the range $[\text{IntCount}(\mathcal{P}, a, b)]$ are sampled and then the intersection count is computed while iterating over the generated indices, reporting the corresponding intersections on the fly [27].

Unfortunately, this approach is not oblivious: First, synchronized iterations (such as over $K$ and the set of intersections generated) are not oblivious in general as step widths depend on the data values encountered. Second, reporting an intersection on the fly leaks information about the lines inducing it.

We address these challenges in the following way. Just as we have done in Binversions, we simulate a synchronized traversal over arrays $A$ and $B$ by first sorting the (labeled) elements and then iterating over their concatenation $A\|B$. For each element, we decide in private memory how to process the element based on its label.

To avoid leaking information about the two lines inducing a single intersection, we operate on batches producing partial results padded to their maximum possible length where needed. This way we do not leak the number of samples from a specific sub-range of the input.

We combine sampling and enumerating into a single building block IntCollect($\mathcal{P}, a, b, K$); $K$ contains the indices of the intersections to sample in ascending order.

**Algorithm 4** Enumerating specified intersections.

1: function IntCollect($\mathcal{P}, a, b, K$) $\triangleright$ $a < b$, $|K| > 0$
2: $\ell \leftarrow 0$, $K' \leftarrow \text{array}[|K|]$ $\triangleright$ Intersection storage
3: $\mathcal{P}_a \leftarrow \text{Sort}_{a}(\mathcal{P})$ $\triangleright$ Sort according to $\leq_a$
4: $I \leftarrow 0$ $\triangleright$ Intersection counter
5: for $l \leftarrow 0, \ldots, \lfloor \log_2|K| \rfloor$ do $\triangleright$ All layers
6: DetermineLineIndices$_{K}(\mathcal{P}_a, I, l)$ $\triangleright$ Upd. $I$
7: $X \leftarrow \text{MatchAgainstLines}(\mathcal{P}_a, K, l)$
8: StoreIntersections($X$, $K'$, $l'$) $\triangleright$ Upd. $K'$
9: return $K'$

From a high-level perspective, the algorithm first sorts the input according to $\leq_a$ and then iteratively implements a bottom-up divide-and-conquer strategy: As in the RAM algorithm sketched before, unique consecutive indices are (implicitly) assigned to all encountered intersection points. Note that, as we randomly sample/enumerate intersections, we may assign indices to the intersections arbitrarily. All lines are explicitly labeled with indices so that — given the index for an intersection — the lines inducing that intersection can easily be identified.

The intersection indices $K$ are then matched against the lines, determining the inducing lines of each intersection. Finally, we store the pair of inducing lines as intersection in $K'$. These three steps are repeated for each layer $l$ so that after processing all layers the inducing lines of all specified intersections are known.

We now discuss the routines called for each layer $l$.

**Assigning indices to lines** The first sub-routine called for each layer $l$ is DetermineLineIndices. Building on the general ideas used in Algorithm 3, it iterates over pairs of subarrays of $2^l$ lines each, updates the intersection counter $I$, and assigns to each line in $\mathcal{P}$ four indices defined below that guide the oblivious sampling.
The index $i$ of a line $\ell$ (or: $1(\ell, i)$) denotes the pair of blocks (on the current layer) containing $\ell$. On each layer, we process only intersections of lines with the same index $i$.

- The index half of a line $\ell$ indicates whether $\ell$ was stored in the first subarray $P_{l_0}$ ($1(\ell, \text{half}) = 0$, “0-line”) or in the second subarray $P_{l_1}$ ($1(\ell, \text{half}) = 1$, “1-line”). For each pair of subarrays, we process only intersections of lines with different indices half.

- For a 0-line $\ell_0$, the index 0-index is the offset of the first intersection induced by $\ell_0$. By construction all intersections induced by $\ell_0$ in this layer have consecutive indices. For a 1-line, 0-index stores the number of intersections counted thus far, i.e., all lines are sorted by their values of 0-index after merging.

- For a 1-line $\ell_1$, 1-index is the offset among all 1-lines in this layer. For a 0-line $\ell_0$, 1-index stores the number of intersection points induced by $\ell_0$.

The resulting algorithm is given as Algorithm 5. Table 1 shows the labels assigned by Algorithm 5 in layer $l = 1$ for an input of 8 lines $\ell_0, \ldots, \ell_7$, numbered according to their $\leq_{\text{by}}$-order. In layer 0, $l = 3$ inversions have been counted. Layer 1 contains 6 inversions.

### Algorithm 5 Assigning indices to lines.

1. procedure DetermineLineIndices($P_a, I, l$)
2. $c^* \leftarrow 0$
3. for $i \leftarrow 0, \ldots, \lfloor \frac{|P_a|}{2} \rfloor$ do $\triangleright$ Pairs of subarrays
4. $P_{l_0} \leftarrow P_a[2 \cdot i \cdot 2^l : 2 \cdot (i + 1) \cdot 2^l - 1]$
5. $P_{l_1} \leftarrow P_a[2 \cdot (i + 1) \cdot 2^l : 2 \cdot (i + 2) \cdot 2^l - 1]$
6. $1(\ell, i) \leftarrow i$ for all $\ell \in P_{l_0} \parallel P_{l_1}$
7. BiInversions($P_{l_0}, P_{l_1}, I, c^*$)

8. procedure BiInversions($P_{l_0}, P_{l_1}, I, c^*$)
9. $1(\ell, \text{half}) \leftarrow 0, \ell \in P_{l_0}; 1(\ell, \text{half}) \leftarrow 1, \ell \in P_{l_1}$
10. $A \leftarrow \text{Merge}(P_{l_0}, P_{l_1}) \triangleright$ Permute labels as well
11. $c \leftarrow 0 \triangleright$ Local counter for 1-lines
12. for $\ell \leftarrow A[0], \ldots, A[|A| - 1]$ do $\triangleright$ 1-lines
13. $1(\ell, 0\text{-index}) \leftarrow I$
14. if $1(\ell, \text{half}) = 0$ then $\triangleright$ Number of int. for $\ell$
15. $1(\ell, 1\text{-index}) \leftarrow c$
16. $I \leftarrow I + c$ $\triangleright$ Update intersection count
17. else $\triangleright 1(\ell, \text{half}) = 1$
18. $1(\ell, 1\text{-index}) \leftarrow c^*$ $\triangleright$ Record offset
19. $c \leftarrow c + 1$ $\triangleright$ Update local counter
20. $c^* \leftarrow c^* + 1$ $\triangleright$ Update level counter

DetermineLineIndices only depends on $n$ and $l$, as does the size of the input to BiInversions, the procedure is oblivious by composability with regard to leakage leak: $\langle P_a, I, l \rangle \mapsto \langle |P_a|, l \rangle$.

### Matching lines and indices

The second sub-routine, MatchAgainstLines (Algorithm 6), pairs the lines inducing intersection points encountered in this layer that correspond to indices in $K$.

First, the indices are matched against the 0-lines. This is done by assigning each index $i \in K$ the 0-index $i + 0.5$ and then merging them with the lines (by
Algorithm 6 Method for matching intersection indices against lines.

1: function MatchAgainstLines($P_a, K, l$)  
\hspace{1cm} \triangleright Lines $P_a$ already have the appropriate labels  
2: \hspace{1cm} $1(i, K) \leftarrow \top$ for all $i \in K$  
3: \hspace{1cm} $1(i, 0\text{-index}) \leftarrow i + 0.5$ for all $i \in K$  
4: \hspace{1cm} $X \leftarrow \text{Merge}_{0\text{-index}}(P_a, K)$  
5: \hspace{1cm} $\ell_0 \leftarrow \bot; \ell_1 \leftarrow \bot$  
6: \hspace{2cm} for $e \leftarrow X[0], \ldots, X[|X| - 1]$ do  
7: \hspace{3cm} if \hspace{0.5cm} $-1(e, K) \land 1(e, \text{half}) = 0 \land 1(e, 1\text{-index}) > 0$ then  
8: \hspace{4cm} \hspace{1cm} $\ell_0 \leftarrow e$  
9: \hspace{3cm} else if \hspace{0.5cm} $1(e, K) \land 1(e, 0\text{-index}) < 1(\ell_0, 0\text{-index}) + 1(\ell_0, 1\text{-index})$ then  
10: \hspace{4cm} \hspace{1cm} $1(e, 0\text{-line}) \leftarrow \ell_0$  
11: \hspace{4cm} \hspace{1cm} $1(e, 1\text{-index}) \leftarrow 1(\ell_0, 1) \cdot 2^l + 1(e, 0\text{-index}) - 1(\ell_0, 0\text{-index})$  
12: \hspace{3cm} $\triangleright$ Find 0-line inducing intersections  
13: \hspace{2cm} Sort$_{1\text{-index}}(X)$  
14: \hspace{2cm} for $e \leftarrow X[0], \ldots, X[|X| - 1]$ do  
15: \hspace{3cm} if \hspace{0.5cm} $-1(e, K) \land 1(e, \text{half}) = 1$ then  
16: \hspace{4cm} \hspace{1cm} $\ell_1 \leftarrow e$  
17: \hspace{4cm} \hspace{1cm} $1(e, 1\text{-line}) \leftarrow \ell_1$  
18: return $X$  

0-line). When iterating over the merged sequence $X$, the 0-line inducing an intersection from this layer is exactly the last 0-line encountered before the index (that induces at least one intersection). Each index $i \in K$ is labeled with the inducing 0-line $\ell_0$ as 0-line and with the index of the corresponding 1-line as 1-index; the 1-index can be determined from the indices assigned to $\ell_0$.

Similarly, the indices are matched against the 1-lines by sorting the array $X$ of lines and indices according to the 1-index. When iterating over the sorted sequence, the previous 1-line before each intersection index is the second line inducing the intersection. Each $i \in K$ already assigned a 0-line can thus be labeled with the inducing 1-line $\ell_1$ as 1-line.

The runtime is dominated by the runtime for merging and sorting and thus is in $O((n + k) \log(n + k))$ for $k := |K|$ and $n := |P_a|$. The algorithm is oblivious since, in addition to merging and sorting, it only consists of linear scans over the array $X$. The input size for merging and sorting is at most $n + k$. Although not explicitly shown it is trivial to implement the loop bodies obliviously with respect to both memory and memory trace-obliviousness. By composability, MatchAgainstLines is oblivious with regard to leakage $\text{leak}: \langle P_a, K, l \rangle \rightarrow (|P_a|, |K|)$.

Storing intersection The third subroutine called for each layer, StoreIntersections (Algorithm 7), stores the intersections (consisting of the pairs of lines matched in the previous step) in $K'$. Exactly the indices with an assigned 0-line (and thus also 1-line) have been found in this layer. For storing, the building block Append is used where $k'$ is the number of indices already stored in $K'$. Append is oblivious and thus does not leak the number of intersections $k_\Delta$ from this layer. The runtime of this last step is dominated by the filtering and appending steps and thus realizable with runtime $O(n \log n)$ where $n := |X|$. The obliviousness follows from composability with regard to leakage $\text{leak}: \langle X, K', k' \rangle \rightarrow (|X|, |K'|)$.

Algorithm 7 Storing the sampled indices

1: procedure StoreSampledIntersections($X, K', k'$)  
2: \hspace{2cm} $k_\Delta \leftarrow \text{Filter}_{K', 0\text{-line}}(X)$  
3: \hspace{2cm} $\text{Append}(K', X, k', k_\Delta) \triangleright$ Append (pairs of) lines  
4: \hspace{2cm} $k' \leftarrow k' + k_\Delta$

Runtime and obliviousness Let $n := |P|$ be the number of lines and $k := |K|$. The runtime of IntCollect is dominated by the main loop. This results in a total runtime of $O((n + k) \log(n + k)) k \in \Omega(n) O(n \log^2 n)$.

2.4 Analysis

Since our implementation of Matoušek’s algorithm replaces only the building blocks used internally, the correctness and runtime properties follow from the respective analyses of the building blocks. We thus have:

Lemma 2: Correctness and runtime. Let IntSelection be Algorithm 1 instantiated with the oblivious building blocks described above. Then, given a set $P$ of $n$ lines in general position and an integer $k \in \binom{n}{2}$, IntSelection($P, k$) determines the in-
intersection with k-th smallest x-coordinate in expected \(O(n \log^2 n)\) time.

We now turn our attention to the analysis of the proposed algorithm’s obliviousness. Since oblivious programs are composable, we can prove the security by considering the leakage of each oblivious building block.

**Lemma 3: Obliviousness.** Let \(\mathcal{P}\) be a set of \(n\) lines in general position such that \(\binom{\mathcal{P}}{2}\) is odd. If Algorithm 1 is instantiated with the oblivious building blocks described above, MedianSelection(\(\mathcal{P}\)) := \text{IntSelection}(\mathcal{P}, k)

with \(k := \left(\frac{\mathcal{P}}{2}\right) - 1\) obliviously realizes the median intersection selection with respect to leakage leak(\(\mathcal{P}\)) := |\mathcal{P}|.

**Proof.** For the proof, we need to show both the correctness and the security of the algorithm for the specified inputs. The requirements above imply that \(k\) is an integer, so correctness follows from Theorem 2. It remains to show the security.

The oblivious algorithm directly uses the building blocks \(\mathcal{S} := (\text{Sort, Select, IntCount, IntCollect})\) and is realized by first determining the number of inversions \(i := \text{IntCount}(\mathcal{P}, a, b)\) in range \([a, b]\), independently sampling \(k\) random indices \(K \in [I]^k\), sorting the indices \(K\) and calling \(\text{IntCollect}(\mathcal{P}, a, b, K)\). We can thus use \(\text{IntSample}(\mathcal{P}, a, b)\) by initializing an array \(K := \langle 0, \ldots, i - 1 \rangle\) and calling \(\text{IntCollect}\). All building blocks are oblivious, with \text{Select} additionally leaking the rank of the selected element, \text{IntSample} leaking the number of samples via the size of \(K\) and \text{IntEnumeration} leaking the number of intersections in the given range, also via the size of \(K\). The arithmetic expressions and assignments operate on a constant number of memory cells and are trivially oblivious.

We first examine the values of \(n, k, N, N' := \text{IntCount}(\mathcal{P}, -\infty, a)\) throughout the execution of the algorithm. The value of \(n\) remains constant and \(k\) is fixed relative to \(n\), so we consider the sequence \(B = \langle (N_0, N_0'), (N_1, N_1'), \ldots, (N_m, N_m') \rangle\) where \(N_i, N'_i\) are the values for \(N, N'\) after the \(i\)-th iteration of the main loop for a total of \(m\) loop iterations. In each iteration of the main loop, \(n\) intersections \(R\) are chosen uniformly at random from the range \([a, b]\). Since \(\mathcal{P}\) is in general position, the intersections of distinct pairs of lines are distinct and all intersections are totally ordered. This implies that the random distribution of \(\text{IntCount}(\mathcal{P}, -\infty, c)\) for an intersection \(c\) with fixed rank in \(R\) only depends on \(n, N, N'\). Both \(j_a\) and \(j_b\) are fixed relative to \(n, N, N'\), so the random distribution of the next values for \(N, N'\) is solely determined by \(n\) and the previous values. Since initially \(N_0 = \binom{n}{2}\) and \(N'_0 = 0\) and the sequence ends with \(N_m \leq n\), the random distribution of the complete sequence \(B\) is solely determined by \(n\).

It can easily be seen that each sequence \(B\) of values for \(N, N'\) determines the sequence \(A\) of memory probes and sub-procedure invocations. This implies that any sequence \(A\) is equally likely for inputs of the same size and thus that MedianSelection is secure by compositability.

### 3 Non-general positions

For simplicity of exposition, we assumed so far that the lines \(\mathcal{P}\) are in general position, i.e., that all intersection points of two lines in \(\mathcal{P}\) have distinct \(x\)-coordinates and that all lines in \(\mathcal{P}\) have distinct slopes. We also assumed that the number of intersection points is odd, so that the median intersection point selection problem can always be solved by one call to a general intersection point selection algorithm; this latter assumption can be removed by computing both the element with rank \(k_1 = \lceil \frac{N-1}{2} \rceil\) and with rank \(k_2 = \lfloor \frac{N-1}{2} \rfloor\) (for \(N = \binom{\mathcal{P}}{2}\)) and returning their mean if there is an even number of intersections \([31\)]. Since \(k_1\) and \(k_2\) differ by one at most by one, both intersections can be computed simultaneously with no significant impact on the runtime.

In RAM algorithms, degenerate configurations are a nuisance, but often can be handled by generic approaches [e.g. 12, 30, 35]. For our proposed algorithm, we must take care that these approaches do not affect the obliviousness. In particular, the runtime of the algorithm must not depend on the number of intersection points with identical \(x\)-coordinates; this rules out the problem-specific technique described by Dillencourt, Mount, and Netanyahu [11] to explicitly handle non-general position.

Regarding arithmetic precision, we note that the only arithmetic computation performed on the input values is the calculation of the \(x\)-coordinate of an intersection point. Thus, recall we are working in the word RAM model, for fixed-point input values with \(b\) bits of precision the use of \(2(b + 1)\) bits of precision suffices to perform all arithmetic computations exactly.

**Parallel lines** For technical reasons, we first discuss how to deal with inputs in which lines are parallel, i.e., for which we cannot assume distinctness of slopes.

Earlier on, we noted that our algorithm is allowed to leak the values of \(N\) and \(k\).\(^4\) This means that we cannot introduce data-dependency of these values and this, in turn, implies that (a) pairs of parallel lines cannot simply be excluded and that (b) \(k\) cannot be adjusted based on the number of pairs of parallel lines.

---

\(^4\)Assuming the leakage of \(k\) allows us to treat the original algorithm of Matoušek as a black box. The author proves an expected lower bound on the reduction of \(N\) per loop iteration which is independent of \(k\). This does not necessarily imply that the exact reduction of \(N\) is in fact independent of \(k\).
We address this using a problem-specific, controlled version of the symbolic perturbation of Edelsbrunner and Mücke [12] and Yap [35]. We perturb the lines \( \mathcal{P} \) in such a way that each pair \( \{ \ell_1, \ell_2 \} \) of lines intersects in a single intersection point \( p_{1 \times 2} \). Let \( V \) be the set of intersections induced by lines that were parallel previous to the perturbation. We ensure that \( V \) is partitioned into \( V = V_- \cup V_+ \) such that \( V_- \) and \( V_+ \) are (nearly) equally sized and each \( v \in V_- \) has a \( x \)-coordinate less and each \( v' \in V_+ \). By equally distributing these “virtual” intersections to the left and to the right of all “real” intersections we maintain data-independent values of \( N = \binom{n}{2} \) and \( k = \frac{N-1}{2} \).

To realize this (symbolic) perturbation, we follow Edelsbrunner and Mücke [12] and introduce an infinitesimally small value \( \varepsilon > 0 \). We then identify each line \( \ell : x \mapsto m(\ell) \cdot x + b(\ell) \) with the perturbed line \( \ell' : x \mapsto m'(\ell) \cdot x + b'(\ell) \) where \( m'(\ell) := m(\ell) + s_\ell \cdot \# \ell \cdot \varepsilon^2 \), \( b'(\ell) := b(\ell) + \# \ell \cdot \varepsilon \), \( s_\ell \in \{-1, +1\} \) is a factor to achieve the distribution into \( V_- \) and \( V_+ \), and \( \# \ell \in \mathbb{N}_0 \) is a unique index given to each line with respect to the order of the line offsets, i.e., \( \forall \ell_1, \ell_2 \in \mathcal{P} : b(\ell_1) < b(\ell_2) \implies \# \ell_1 < \# \ell_2 \). We obtain the set \( \mathcal{P} ' \) of perturbed lines.

Due to space constraints, we omit the details of how to compute \( \# \ell \) and \( s_\ell \) as well how to avoid leaking the number of “virtual” intersections.

**Intersections with identical \( x \)-coordinates**

To handle intersections \( p, q \in \mathcal{L} \) with identical \( x \)-coordinates without significantly affecting the runtime of the algorithm, we establish a total order \( \preceq \) over all intersections, so that the lines \( \mathcal{P} \) can be totally ordered relative to each intersection \( p \) as in Definition 2. For this, we characterize an intersection by its inducing pair of lines and define an order based on these lines’ properties:

**Definition 3.** Let \( P_x := \mathcal{L} \cup \{-\infty, +\infty\} \) be the set of all intersections with additional elements \(-\infty \) and \( +\infty \). Let each \( p \in \mathcal{L} \) be formed by lines \( p_1 \) and \( p_2 \) with \( m(p_1) > m(p_2) \). We define a total order \( \preceq \) over \( P_x \) via:

\[
p \preceq q \iff \begin{cases} px < qx & \text{if } p_x \neq q_x \\ m(p_1) < m(q_1) & \text{else if } p_1 \neq q_1 \\ m(p_2) \leq m(q_2) & \text{else} \end{cases}
\]

for \( p, q \in P_x \setminus \{-\infty, +\infty\} \) and with special cases \(-\infty \preceq p \) and \( p \preceq +\infty \) for all \( p \in P_x \). Let \( \prec \) denote the corresponding strict order over \( P_x \).

By construction, \( \preceq \) is a (lexicographic) total order. This is ensured by the fact that all slopes are distinct. This order suffices to construct the total order over the lines in \( \mathcal{P} \). To show this, we need the following lemma:

**Lemma 4.** Let \( \ell_1, \ell_2, \ell_3 \) be non-vertical lines with \( m(\ell_1) < m(\ell_2) < m(\ell_3) \). Of the three intersections induced by these lines, the intersection \( p_{1 \times 3} \) of the two lines with extremal slopes is the median with respect to the order \( \preceq \) defined above.

Assuming only the distinctness of slopes (which, as discussed above, may be assumed w.l.o.g.), we have:

**Lemma 5.** Let \( P_x, \preceq, \prec \) be as in Definition 3. For each \( p \in P_x \), we have a total order \( \preceq_p \) over \( \mathcal{P} \):

\[
\ell_1 \preceq_p \ell_2 \iff \begin{cases} \top & \text{if } \ell_1 = \ell_2 \\ p \preceq p_{1 \times 2} & \text{if } m(\ell_1) > m(\ell_2) \\ p_{1 \times 2} \prec p & \text{if } m(\ell_1) < m(\ell_2) \end{cases}
\]

With the above definition, we can impose a total order on the set of lines irrespective of whether or not their intersection points’ \( x \)-coordinates are distinct. Since the predicate \( p \preceq q \) for intersections \( p, q \in P_x \) can still be evaluated in constant time, the asymptotic runtime of the algorithm remains unchanged.

**Summary** In conclusion, the two techniques sketched in this section generalize the algorithm not only to inputs \( \mathcal{P} \) with parallel lines, but also to inputs with identical lines. The algorithm is thus applicable to arbitrary inputs. Since we can achieve the desired (symbolic) perturbation via pre-processing in \( O(n \log n) \) time for an input of \( n \) lines, our main theorem follows:

**Theorem 6:** Main result. There exists a RAM program that obliviously realizes the median selection in expected \( O(n \log^2 n) \) time for \( n \) non-vertical lines inducing at least one intersection.

4 Implementation and evaluation

We developed a prototype of our oblivious algorithm in C++.\(^5\) The goal of the implementation is to show that the algorithm is easily implementable and to provide an estimate of the algorithm’s performance. For this we also implemented the baseline algorithm [27].

**Limitations** The primary limitation is that our prototype only accesses arrays of non-constant size in an oblivious manner. Code fragments such as inner loops and methods accessing only a constant number of memory cells do not necessarily probe memory obliviously. Even though it is conceptually trivial to transform those code fragments to achieve “full” obliviousness, we note that — without publicly available libraries providing low-level primitives for implementations of oblivious algorithms — the obliviousness eventually might depend on the compiler and platform used.

We believe that our implementation still provides a good estimate of the performance of a “fully” oblivious implementation: The loops in our runtime-intensive

\(^5\)http://go.wwu.de/aa6fz
primitives are all linear scans over arrays. As such “fully” oblivious loop bodies will not introduce a large overhead since they will likely not introduce cache misses. Also our oblivious primitives can also be implemented largely without data-dependent branches, thus potentially eliminating branch mispredictions.

The second main limitation is that we do not implement the handling of parallel lines (as described in Section 3). This would require an additional pre-processing step as well as extending both the slope and the offset with a symbolic perturbation. As mentioned above this would result in a low constant factor overhead in both runtime and memory space usage. Since this applies to both the oblivious and non-oblivious algorithm this has no direct implication for the performance evaluation below, although there might be a more efficient way to handle identical slopes in the non-oblivious case.

Finally, our implementation resorts to a suboptimal, but easy-to-implement oblivious sorting primitive with $O(n \log^2 n)$ and thus has an expected $O(n \log^2 n)$ runtime in the oblivious setting. This leads to an additional $O(\log n)$ overhead in runtime as compared to our non-oblivious implementation and thus underestimates the performance of the proposed algorithm.

Performance  We used libbenchmark\(^6\) to measure the runtime for inputs ranging from 1,000 to 30,000 lines. The input consists of shuffled sets of lines with non-uniformly increasing slope and a random offset, both represented by 64-bit integers. For all our experiments and independent of $n$, we fixed an interval $[m_{\min}, m_{\max}]$ and an interval $[b_{\min}, b_{\max}]$. To generate a set of $n$ random lines, we then set $r := (m_{\max} - m_{\min}) / n$ and constructed each line $\ell_i = \langle m_i, b_i \rangle$ in turn by independently sampling a random slope $m_i$ from $m_{\min} + i \cdot r \leq m_i < m_{\min} + (i + 1) \cdot r$ (thus ensuring both spread and distinctness of slopes) and a random offset $b_i$ from $b_{\min} \leq b_i \leq b_{\max}$. We then permuted the resulting set of lines using std::ranges::shuffle.

\(^6\)https://github.com/google/benchmark

The performance evaluation results are shown in Fig. 1. For inputs of 10,000–30,000 random lines our algorithm is about 150–210 times slower than the baseline algorithm. While this is a significant slowdown, we remind the reader of both the logarithmic overhead incurred by choosing a suboptimal sorting algorithm and the fact that the baseline algorithm does not offer any obliviousness. The runtime was less than 10 seconds for all evaluated input sizes.

All experiments were performed on a Dell XPS 7390 with an Intel i7–10510U CPU and 16 GiB RAM running Ubuntu 20.04.

Obliviousness  We assessed the obliviousness of our implementation of the building blocks by tracing memory accesses as part of unit testing. For this, we abstracted the memory sections as arrays of fixed but dynamic size. We assigned a fingerprint to each sequence of reads and writes by hashing both the memory operation and the access location. Since all building blocks used by the main algorithm are deterministic, we asserted their obliviousness by comparing fingerprints for different inputs with identical leakage.

Additionally, we evaluated the runtime of both our oblivious algorithm and the baseline algorithm when applied to two inputs of different characteristics. For this we compared the random lines described above with a sorted set of lines $\langle i, -i \rangle$, intersecting in the single point $p = \langle 1, 0 \rangle$. The baseline algorithm showed significantly different runtimes for different inputs (Fig. 2), making it abundantly clear that even without statistical analyses an adversary can distinguish these different kinds of input from the runtime alone. In contrast, there was only slight variation in the runtime of our proposed algorithm which we attribute to the presence of code processing constant-sized subproblems in a (currently) non-oblivious manner.
5 Conclusion

We presented a modification of Matoušek’s randomized algorithm [27] for obliviously determining the median slope for a given set of $n$ points. We also showed how to generalize the algorithm to arbitrary inputs — allowing both collinear points and multiple points with identical $x$-coordinate — while maintaining obliviousness. Our modified algorithm has an expected $O(n \log^2 n)$ runtime, matching the general oblivious transformation bound of the original algorithm. We provide a proof-of-concept of the oblivious algorithm in C++, showing that the algorithm indeed can be implemented and has a runtime that makes its application viable in practice.

References

[1] Rakesh Agrawal et al. “Sovereign Joins”. In: Proceedings of the 22nd International Conference on Data Engineering. 2006. DOI: 10.1109/ICDE.2006.144.

[2] Miklós Ajtai, János Komlós, and Endre Szemerédi. “An $O(n \log n)$ Sorting Network”. In: Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing. 1983, pp. 1–9. DOI: 10.1145/800618.808726.

[3] Gilad Asharov et al. “Bucket Oblivious Sort: An Extremely Simple Oblivious Sort”. In: Proceedings of the 3rd SIAM Symposium on Simplicity in Algorithms. 2020, pp. 8–14. DOI: 10.1137/1.9781611976014.2.

[4] Gilad Asharov et al. OptORAMa: Optimal Oblivious RAM. 2018/892. 2018. URL: https://eprint.iacr.org/2018/892/20200916:051812.

[5] Gilad Asharov et al. “OptORAMa: Optimal Oblivious RAM”. In: Advances in Cryptology – EUROCRYPT 2020. Vol. 12106. Lecture Notes in Computer Science. 2020, pp. 403–432. DOI: 10.1007/978-3-030-45724-2_14.

[6] Ken E. Batcher. “Sorting Networks and Their Applications”. In: Proceedings of the April 30–May 2, 1968 Spring Joint Computer Conference. 1968, pp. 307–314. DOI: 10.1145/1468075.1468121.

[7] Henrik Blunck and Jan Vahrenhold. “In-Place Randomized Slope Selection”. In: Algorithms and Complexity. Vol. 3998. Lecture Notes in Computer Science. 2006, pp. 30–41. DOI: 10.1007/11758471_6.

[8] Hervé Brunnimann and Bernard Chazelle. “Optimal Slope Selection via Cuttings”. In: Computational Geometry 10.1 (1998), pp. 23–29. DOI: 10.1016/S0925-7721(97)00025-4.

[9] T.-H. Hubert Chan et al. “Cache-Oblivious and Data-Oblivious Sorting and Applications”. In: Proceedings of the Twenty-Ninth Annual ACM-SIAM Symposium on Discrete Algorithms. 2018, pp. 2201–2220. DOI: 10.1137/1.9781611975031.143.

[10] Richard Cole et al. “An Optimal-Time Algorithm for Slope Selection”. In: SIAM Journal on Computing 18.4 (1989), pp. 792–810. DOI: 10.1137/0218055.

[11] Michael B. Dillencourt, David M. Mount, and Nathan S. Netanyahu. “A Randomized Algorithm for Slope Selection”. In: International Journal of Computational Geometry & Applications 2.1 (1992), pp. 1–27. DOI: 10.1142/S0218195992000020.

[12] Herbert Edelsbrunner and Ernst Peter Mücke. “Simulation of Simplicity: A Technique to Cope with Degenerate Cases in Geometric Algorithms”. In: ACM Transactions on Graphics 9.1 (1990), pp. 66–104. DOI: 10.1145/77635.77639.

[13] David Eppstein, Michael T. Goodrich, and Roberto Tamassia. “Privacy-Preserving Data-Oblivious Geometric Algorithms for Geographic Data”. In: Proceedings of the 18th SIGSPATIAL International Conference on Advances in Geographic Information Systems. 2010, pp. 13–22. DOI: 10.1145/1869790.1869796.

[14] Oded Goldreich. “Towards a Theory of Software Protection and Simulation by Oblivious RAMs”. In: Proceedings of the Nineteenth Annual ACM Symposium on Theory of Computing. 1987, pp. 182–194. DOI: 10.1145/233551.233553.

[15] Oded Goldreich and Rafail Ostrovsky. “Software Protection and Simulation on Oblivious RAMs”. In: Journal of the ACM 43.3 (1996), pp. 431–473. DOI: 10.1145/233551.233553.

[16] Michael T. Goodrich. “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. 2011, pp. 379–388. DOI: 10.1145/1989493.1989555.

[17] Michael T. Goodrich. “Randomized Shellsort: A Simple Oblivious Sorting Algorithm”. In: Proceedings of the Twenty-First Annual ACM-SIAM Symposium on Discrete Algorithms. 2010, pp. 1262–1277. DOI: 10.1137/1.9781611973075.101.
33rd Canadian Conference on Computational Geometry, 2021

[18] Pavel Hubáček et al. “Stronger Lower Bounds for Online ORAM”. In: Theory of Cryptography, Vol. 11892. 2019, pp. 264–284. DOI: 10.1007/978-3-030-36033-7_10.

[19] Matthew J. Katz and Micha Sharir. “Optimal Slope Selection via Expanders”. In: Information Processing Letters 47.3 (1993), pp. 115–122. DOI: 10.1016/0020-0190(93)90234-Z.

[20] Donald Ervin Knuth. Sorting and Searching. Vol. 3. The Art of Computer Programming. 1973.

[21] Simeon Krastnikov, Florian Kerschbaum, and Douglas Stebila. “Efficient Oblivious Database Joins”. In: Proceedings of the VLDB Endowment 13.12 (2020), pp. 2132–2145. DOI: 10.14778/3407790.3407814.

[22] Kasper Green Larsen and Jesper Buus Nielsen. “Yes, There Is an Oblivious RAM Lower Bound!” In: Advances in Cryptology. Vol. 10992. Lecture Notes in Computer Science. 2018, pp. 523–542. DOI: 10.1007/978-3-319-96881-0_18.

[23] Yaping Li and Minghua Chen. “Privacy Preserving Joins”. In: Proceedings of the 2008 IEEE 24th International Conference on Data Engineering. 2008, pp. 1352–1354. DOI: 10.1109/ICDE.2008.4497553.

[24] Wei-Kai Lin, Elaine Shi, and Tiancheng Xie. Can We Overcome the n log n Barrier for Oblivious Sorting? 2018/227. 2018. URL: https://eprint.iacr.org/2018/227.

[25] Wei-Kai Lin, Elaine Shi, and Tiancheng Xie. “Can We Overcome the n log n Barrier for Oblivious Sorting?” In: Proceedings of the 2019 Annual ACM-SIAM Symposium on Discrete Algorithms. 2019, pp. 2419–2438. DOI: 10.1137/1.9781611975482.148.

[26] Chang Liu, Michael Hicks, and Elaine Shi. “Memory Trace Oblivious Program Execution”. In: 2013 IEEE 26th Computer Security Foundations Symposium. 2013, pp. 51–65. DOI: 10.1109/CSF.2013.11.

[27] Jiří Matoušek. “Randomized Optimal Algorithm for Slope Selection”. In: Information Processing Letters 39.4 (1991), pp. 183–187. DOI: 10.1016/0020-0190(91)90177-J.

[28] Peter Bro Miltersen, Mike Paterson, and Jun Tarui. “The Asymptotic Complexity of Merging Networks”. In: Journal of the ACM 43.1 (1996), pp. 147–165. DOI: 10.1145/227595.227693.

[29] Sajin Sasy and Olga Ohrimenko. “Oblivious Sampling Algorithms for Private Data Analysis”. In: Advances in Neural Information Processing Systems 32. 2019, pp. 6495–6506. URL: http://papers.nips.cc/paper/8877-oblivious-sampling-algorithms-for-private-data-analysis.

[30] Stefan Schirra. “Precision and Robustness in Geometric Computations”. In: Algorithmic Foundations of Geographic Information Systems. Vol. 1340. Lecture Notes in Computer Science. 1996, pp. 255–287. ISBN: 978-3-540-69653-7.

[31] Pranab Kumar Sen. “Estimates of the Regression Coefficient Based on Kendall’s Tau”. In: Journal of the American Statistical Association 63.324 (1968), pp. 1379–1389. DOI: 10.1080/01621459.1968.10480934.

[32] Elaine Shi. “Path Oblivious Heap: Optimal and Practical Oblivious Priority Queue”. In: Proceedings of the 2020 IEEE Symposium on Security and Privacy. 2020, pp. 842–858. DOI: 10.1109/SP40000.2020.00037.

[33] Emil Stefanov and Elaine Shi. “ObliviStore: High Performance Oblivious Distributed Cloud Data Store”. In: Proceedings of the 20th Annual Network & Distributed System Security Symposium. 2013. URL: https://www.ndss-symposium.org/ndss2013/ndss-2013-programme/oblivistore-high-performance-oblivious-distributed-cloud-data-store/.

[34] Emil Stefanov et al. “Path ORAM: An Extremely Simple Oblivious RAM Protocol”. In: Proceedings of the 2013 ACM SIGSAC Conference on Computer & Communications Security. 2013, pp. 299–310. DOI: 10.1145/2508859.2516660.

[35] Chee-Keng Yap. “A Geometric Consistency Theorem for a Symbolic Perturbation Scheme”. In: Journal of Computer and System Sciences 40.1 (1990), pp. 2–18. DOI: https://doi.org/10.1016/0022-0000(90)90016-E.
Appendices

A — Composability of oblivious programs

Here we prove the composability of oblivious programs, i.e., that an oblivious program additionally invoking other oblivious programs as sub-procedures remains oblivious. This is an adoption of the argument by Asharov et al. [4] to our definition of obliviousness.

Let $S := (R_0, \ldots, R_k)$ be probabilistic RAM programs with $k \geq 1$ such that $R_1, \ldots, R_k$ are oblivious. We will first analyze the obliviousness of $R_0$ in the $S$-hybrid RAM model: Program $R_0$ may invoke any program from $S$ as sub-procedure. For the invocation of $R_i$ we assume that $R_0$ copies the input $x_i$ for $R_i$ to a new location $p$ in memory and executes a special machine instruction \texttt{invoke}(R$_i$, $p$) (only available in the $S$-hybrid model). The instruction immediately changes the partial memory beginning at location $p$ as if program $R_i$ were executed on the memory with offset $p$. Since $R_i$ is executed as part of the instruction, any memory probes performed by $R_i$ are not part of the probe sequence of $R_0$ in the $S$-hybrid model. To ensure that the execution of $R_i$ does not interfere with the memory of $R_0$ a location $p$ after all used memory locations must be selected. After the invocation $R_0$ can read the result computed by $R_i$ from memory.

In the $S$-hybrid model we augment the probe sequence for $R_0$ with the sub-procedure invocations. Similarly to the access locations for memory probes we identify each invocation by the invoked program and the leakage for the respective input:

**Definition 4: Augmented probe sequence.** Let $A$ be as defined in Section 1.2. Let $S$ be as defined above and for each $R_i$ with $1 \leq i \leq k$ let $\text{leak}_i : X_i \rightarrow \{0, 1\}^*$ be the leakage. We define the set of probes visible to the adversary during the execution of $R_0$ in the $S$-hybrid model as $A_S := A \cup \{\text{invoke}\} \times S \times \{0, 1\}^*$.

The random variable $A_{R_0(x)} : \Omega \rightarrow A_S^*$ is defined to denote the hybrid sequence of probes and sub-procedure invocations by $R_0$ for input $x$. Specifically, for each memory probe $\text{probe} \in \{\text{read, write}\}$ at location $i \in [N]$ the sequence contains an entry $\langle \text{probe}, i \rangle$. The sequence does not include memory probes performed by sub-procedures. For each invocation of sub-procedure $R_i \in S$ with input $x_i$ the sequence contains an entry $\langle \text{invoke}, R_i, \text{leak}_i(x_i) \rangle$.

Note that according to the definition above the offset of the partial memory $p$ is not visible to the adversary. This is a simplification which is justified by the fact that $R_0$ can always choose $p$ to be directly after the largest memory location written to before. This guarantees that no memory contents are overwritten and implies that the adversary, given any probe sequence $A \in A_S^*$, can reconstruct $p$ for all sub-procedure invocations.

Invocations in the plain model can be realized by executing $R_i$ directly instead of the instruction $\text{invoke}(R_i, p)$. The offset of the partial memory $p$ can be held in a single special register and applied to every memory probe by a simple modification of $R_i$. The register contents can be temporarily stored in memory and recovered after the execution of the sub-procedure. In the plain model memory probes performed by $R_i$ are contained in the probe sequence. The goal now is to show that obliviousness of $R_0$ in the $S$-hybrid model implies obliviousness of the composed program $R_0$ in the plain RAM model:

**Lemma 7: Composability of oblivious programs.** Let $f_0, \ldots, f_k$ with $f_i : X_i \rightarrow Y_i$ be computable functions, $R_0, \ldots, R_k$ randomized RAM programs and $\text{leak}_0, \ldots, \text{leak}_k$ with $\text{leak}_i : X_i \rightarrow \{0, 1\}^*$ leakages. $R_0$ obliviously simulates $f_0$ with respect to leakage $\text{leak}_0$ in the plain model if

(a) each $R_i$ for $1 \leq i \leq k$ obliviously simulates $f_i$ with respect to leakage $\text{leak}_i$ in the plain model,

(b) $R_0$ is correct in the $S$-hybrid model, i.e., for all inputs $x \in X_0$ the equality $\Pr[R_0(x)^S = f_0(x)] = 1$ holds,

(c) and $R_0$ is secure in the $S$-hybrid model, i.e., for all inputs $x, x' \in X_0$ with $\text{leak}_0(x) = \text{leak}_0(x')$ the equality $\sum_{A \in A_S^*} \Pr[A_{R_0(x)}^S = A] - \Pr[A_{R_0(x')}^S = A] = 0$ holds.

Proof. To prove this we need to show that $R_0$ is both correct and secure in the plain model.

The correctness immediately follows from the correctness of $R_0$ in the $S$-hybrid model: Since the invocation in the plain and $S$-models are functionally equivalent, termination with correct result in the $S$-hybrid model implies termination with the correct result in the plain model. It thus only remains to show that $R_0$ is secure in the plain model, i.e., that any finite probe sequence $A$ is equally likely for any two inputs with identical leakage.

We first consider the simple case where $R_0$ does not invoke itself. For this we fix any two inputs $x, x' \in X_0$ with $\text{leak}_0(x) = \text{leak}_0(x')$ and any finite probe sequence $A \in A_S^*$ for $R_0$ in the plain model. Consider any separation $A = \langle A_0 || I_1 || A_1 || \ldots || I_n || A_n \rangle$ of $A$ where each $A_i$ consists of any number of memory probes performed by $R_0$ and each $I_i$ consists of the memory probes performed during the $i$-th invocation. Each separation of $A$ corresponds to any probe sequence $A' = \langle A_0 || \langle \text{invoke}, R_{j_1}, I_{i_1} \rangle || \ldots || A_n \rangle \in A_S^*$ for $R_0$ in the $S$-hybrid model. In $A'$ the memory probes $A_i$ by $R_0$ remain the same and sub-procedure $R_{j_1} \in S \setminus \{R_0\}$ with some leakage $I_{i_1}$ performs memory probes $I_{i_1}$.
We need to show the security
\[
\Pr[\mathcal{A}_{R_0}(x) = A] = \Pr[\mathcal{A}_{R_0}(x') = A]
\]
in the plain model. Since the distribution of memory probes for the sub-procedures is independent of the memory probes \(A_i\) by construction\(^7\), it follows that the probability for \(A\) with a specific \(A'\) under input \(y\) is exactly
\[
\Pr[\mathcal{A}^S_{R_0}(y) = A'] \cdot \prod_{1 \leq i \leq n} \Pr[\mathcal{A}_{B_i}(y_i) = I_i]
\]
for some inputs \(y_i\) to the sub-procedures with respective leakages \(l_i\). From the security of \(R_0\) in the \(S\)-hybrid model and the security of each \(R_i\) in the plain model it follows that this is the same for both \(y := x\) and \(y := x'\). Thus the probability for \(A\) — summing over all possible \(A'\) — is also the same and \(R_0\) is secure in the plain model.

For the second case we show that the lemma also holds when \(R_0\) invokes itself recursively. We do so by induction over the depth \(d\) of the recursion. The base case \(d = 0\) corresponds to the case above when \(R_0\) does not invoke itself. For \(d > 0\) we consider finite probe sequences \(A\) when each invocation may be either an invocation of sub-procedure \(R_i \in S \setminus \{R_0\}\) or an invocation of \(R_0\) with a recursion depth of at most \(d - 1\). By induction hypothesis for all invocations of \(R_0\) with recursion depth of at most \(d - 1\) the distribution of probe sequences is determined by the leakage of the input. Thus the same is true for any recursion depth \(d \in \mathbb{N}_0\). Since for any finite probe sequence \(A\) the recursion depth of \(R_0\) is also finite this proves the lemma.

\[\square\]

**B — Lower bound**

The lower bound of Cole et al. \cite{cole2007} for the general slope selection problem applies to problem definitions (a) allowing the selection of the smallest slope \((k = 0)\) and (b) regarding slopes through points with equal \(x\)-coordinates as having a non-finite negative slope.

For an arbitrary input \(X = \langle x_1, \ldots, x_n \rangle \in \mathbb{R}^n\) selecting the smallest slope through points \(P = \{\langle x_1, 1 \rangle, \ldots, \langle x_n, n \rangle \}\) yields a non-finite negative slope if and only if not all values in \(X\) are distinct. This proves, through reduction from the element uniqueness problem, a lower bound of \(O(n \log n)\) in the algebraic decision tree model. \cite{cole2007}

This argument can be modified to prove a lower bound for the median slope selection problem also excluding non-finite slopes.

\[\text{Lemma 8. Let } P \subset \mathbb{R}^2 \text{ be a set of } n \text{ points. Then, in general, determining the median slope through points in } P \text{ as defined in Section 1.1 requires } \Omega(n \log n) \text{ steps in the algebraic decision tree model.}\]

**Proof.** As done by Cole et al. \cite{cole2007} we reduce from the element uniqueness problem. Given an arbitrary input \(X = \langle x_1, \ldots, x_n \rangle \in \mathbb{R}^n\), we first transform \(X\) to \(X' = \langle x'_1, \ldots, x'_n \rangle\) containing only positive values by subtracting less than the minimal value:
\[
x'_i := x_i - \min_{1 \leq i \leq n} x_i + 1
\]
For \(1 \leq i \leq n\) we then map each value \(x'_i\) to two points with equal \(x\)-coordinates:
\[
\langle i, x'_i \rangle \text{ and } \langle i, -x'_i \rangle
\]
The last value \(x''_n\) is mapped to two points with distinct \(x\)-coordinates:
\[
\langle n+1, x''_n \rangle \text{ and } \langle n, -x''_n \rangle
\]
Let \(P \subset \mathbb{R}^2\) be the set of these \(2n\) points.

Considering only lines \(L\) through points in \(P\) with positive \(y\)-coordinates, there exist some number \(a \geq 0\) of lines with positive slope and some number \(b \geq 0\) of lines with negative slope. Yet \(L\) also contains a line with a slope of zero if and only if not all values in \(X'\) (and thus also in \(X\)) are distinct. Looking at the lines \(L'\) through points in \(P\) with negative \(y\)-coordinates the same holds true, except because of the inverted \(y\)-coordinates there are exactly \(a\) lines with negative and \(b\) lines with positive slope in \(L'\). \(L'\) also contains the same number of lines with slope zero as \(L\).

It remains to consider the lines through points with different sign of the \(y\)-coordinate. Because all \(x'_i\) are strictly positive and only looking at lines through points not mapped from the same \(x'_i\), there are exactly
\[
\sum_{i=0}^{n-1} i = \frac{(n-1)n}{2}
\]
lines with positive and negative slope, respectively, and no lines with slope zero. Since for each \(x'_i\) with \(1 \leq i < n\) the two points are mapped to the same \(x\)-coordinate, lines through these points are not considered according to the problem definition. The exception is the line through the points \(x''_n\) is mapped to, which has a positive slope.

In summary, there are
\[
c := a + b + \frac{(n-1)n}{2}
\]
lines with negative slope and \(c + 1\) lines with positive slope as well as an even number of lines with slope zero. Thus, the median slope is zero if and only if a line with slope zero exists. This is the case if and only if not all values from \(X\) are distinct.
\[\square\]

\(^7\)This disregards the offset \(p\) of the probe sequences \(I_i\). The argument still holds since we can consider the distribution of the sequences \(I_i\) shifted by \(-p\) instead.