

Supplemental Materials

Secure phasing of private genomes in a trusted execution environment with TX-Phase

Supplemental Figures

Supplemental Figure S1: TX-Phase’s phasing accuracy closely matches SHAPEIT4 and is significantly higher than Eagle2.

Supplemental Figure S2: TX-Phase minimizes the computational overhead of TEEs.

Supplemental Figure S3: Parallel sample processing with TX-Phase.

Supplemental Figure S4: Illustration of oblivious phasing via compressed haplotype selection.

Supplemental Figure S5: Illustration of our dynamic fixed-point arithmetic.

Supplemental Tables

Supplemental Table S1: Our dynamic fixed points enable practical performance of TX-Phase.

Supplemental Table S2: Per-operation cost of log-domain fixed-point arithmetic operations.

Supplemental Table S3: Consistency of performance measurements.

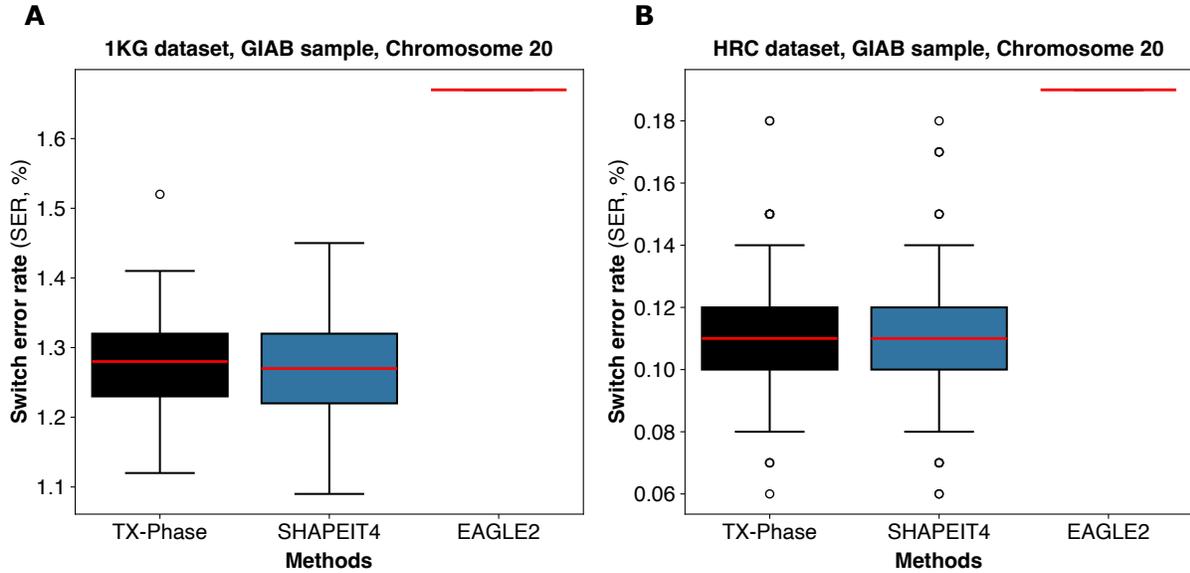
Supplemental Table S4: Notations for our oblivious compressed haplotype selection algorithms.

Supplemental Table S5: Operations for our oblivious compressed haplotype selection algorithms.

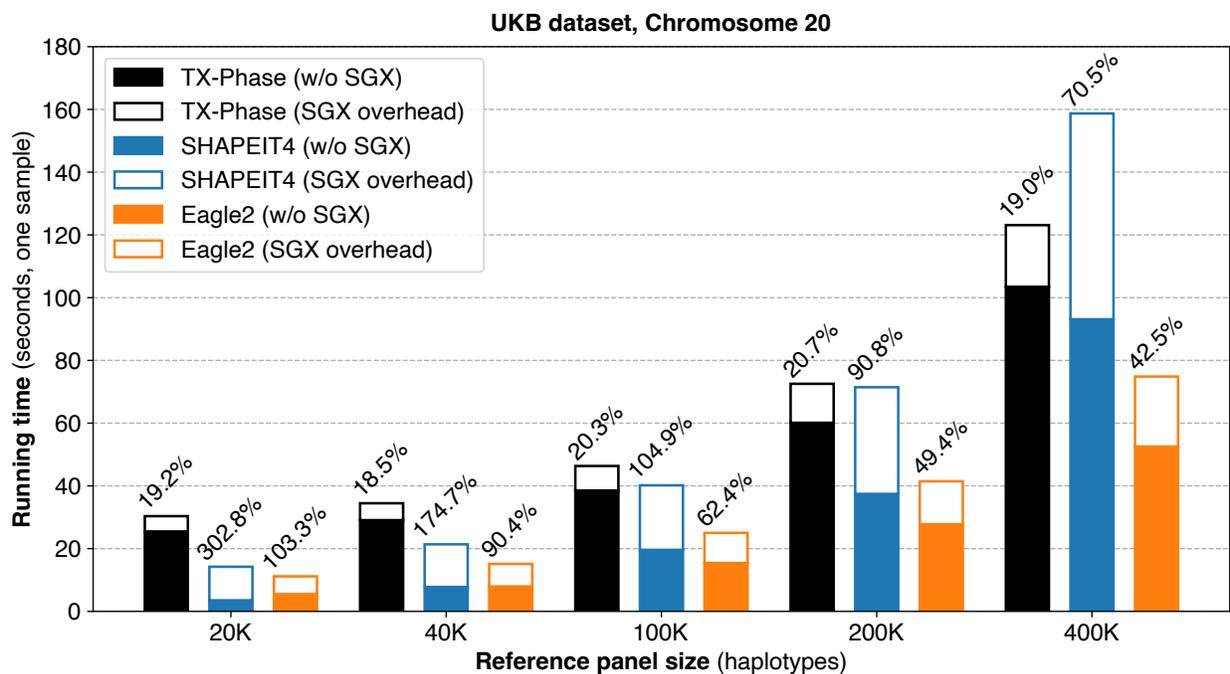
Supplemental Notes

Supplemental Note S1: Additional algorithmic modifications in TX-Phase for oblivious phasing.

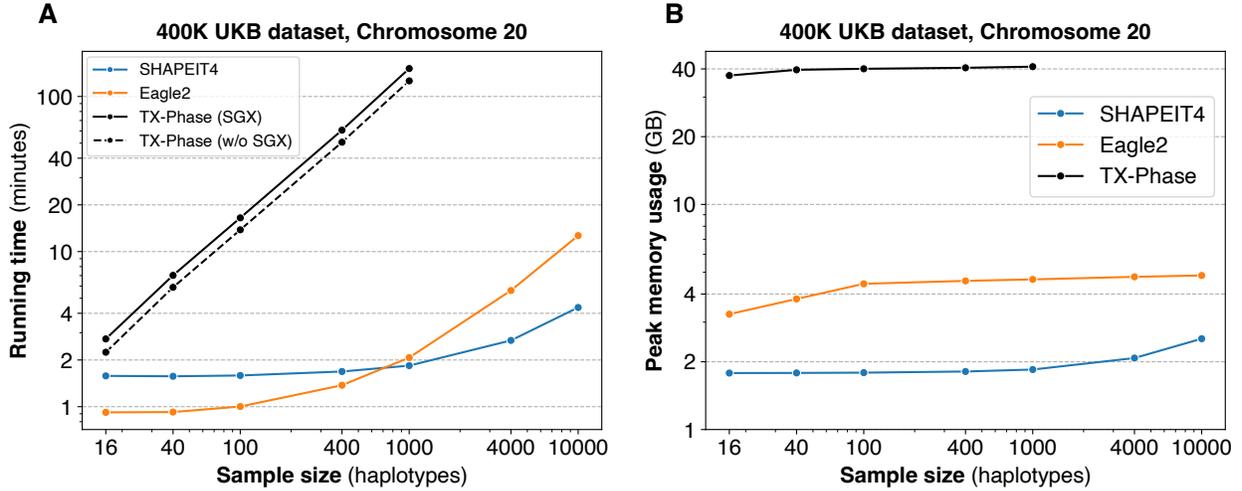
Supplemental Note S2: Our oblivious algorithms for compressed haplotype selection.



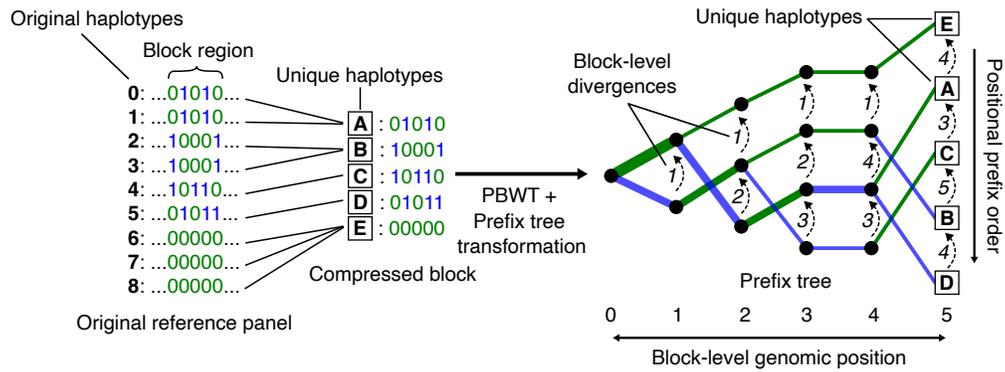
Supplemental Figure S1: TX-Phase’s phasing accuracy closely matches SHAPEIT4 and is significantly higher than Eagle2. We evaluated the phasing accuracy of TX-Phase, SHAPEIT4, and Eagle2 on a GIAB Ashkenazi trio (Chromosome 20) using the 1KG (**A**) and HRC (**B**) datasets as reference panels. We repeated the experiment with 100 random seeds for TX-Phase and SHAPEIT4; Eagle2’s algorithm is deterministic. Accuracy is measured by switch error rates (SERs). Red horizontal lines indicate the median SER, the boxes represent the interquartile range, and the whiskers extend to the most extreme data points, excluding outliers, which are marked by open circles. TX-Phase SER is not significantly different from SHAPEIT4 when using either the 1KG (two-sided Wilcoxon signed-rank test [WSR] $p = 0.707$) or HRC (WSR $p = 0.443$) reference panels. Eagle2 results in substantially higher SERs in both cases.



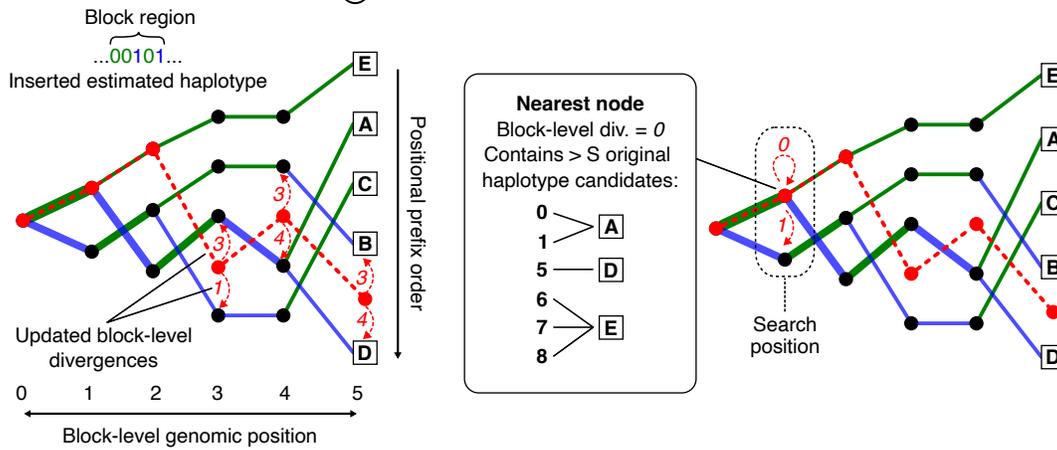
Supplemental Figure S2: TX-Phase minimizes the computational overhead of TEEs. We conducted experiments to compare the overheads of running TX-Phase, SHAPEIT4, and Eagle2 in an SGX enclave compared to outside of SGX. We report per-sample running times for phasing Chromosome 20 with different UKB reference panels. The numbers shown on top of the bars indicate the overhead as a fraction of the baseline running time without SGX. The results demonstrate that TX-Phase’s overheads in an SGX enclave are the lowest and consistent at around 20%. In contrast, the overheads are substantially larger in other methods and reduce as the reference panel size grows.



Supplemental Figure S3: Parallel sample processing with TX-Phase. We evaluated the running times of TX-Phase, SHAPEIT4, and Eagle2 for processing multiple samples using a 16-core CPU. TX-Phase is deployed in 16 independent enclaves in parallel (one enclave per core) to process samples in batches, and the user attests and upload batches of samples to each enclave separately. SHAPEIT4 and Eagle2 have native support for multi-threading and incorporate additional algorithmic modifications for handling multiple samples. We measured the running time (**A**) and peak memory usage (**B**) of each method on the 400K haplotype UKB dataset (Chromosome 20) with varying numbers of target samples. Additionally, we measure the running time of TX-Phase without SGX (**A**) to demonstrate the minimal SGX overhead of our method. Although existing tools perform better on large datasets due to additional optimizations for the multi-sample setting, these results confirm that TX-Phase can be easily parallelized to obtain practical performance on moderately sized datasets without incurring additional runtime overheads. TX-Phase’s memory usage scales linearly with the number of parallel enclaves rather than the sample size, allowing flexible deployment on servers with various CPU and memory configurations. For example, we estimate that running TX-Phase on a Microsoft Azure DC48s v3 server (48 CPU cores, 384 GB memory) can phase 5,000 whole-genome samples (805,426 markers in the SNP array) in 5.3 days while utilizing 162 GB of memory, with a total cost of \$586 (at \$4.608 per hour) based on cost estimates from <https://azure.microsoft.com/en-us/pricing/calculator>. The running time can be further reduced by leveraging more computational resources or incorporating multi-sample optimizations into TX-Phase, which is a meaningful direction for future work. Note that phasing services are mainly intended for target datasets of small to moderate sizes since reference-based phasing offers limited benefit over cohort-based phasing for large cohorts (Loh et al. 2016).

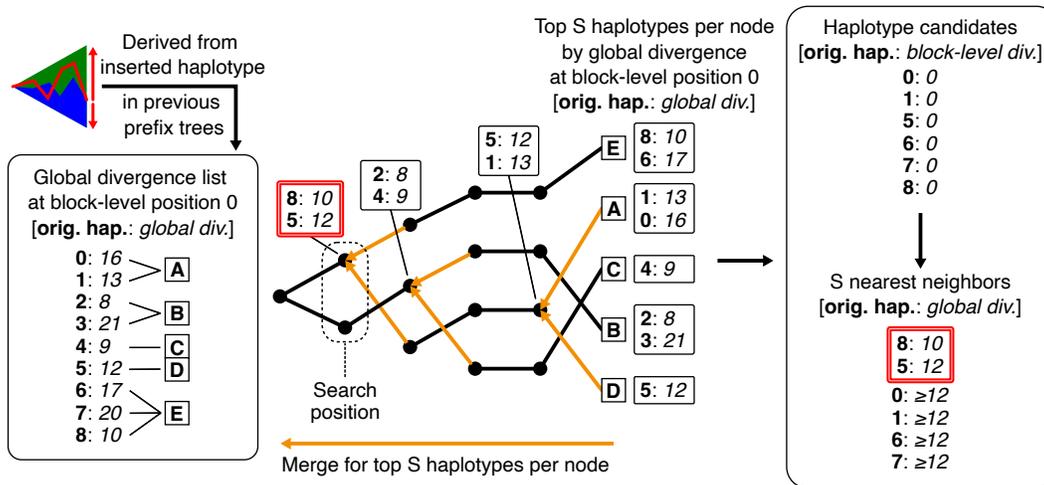


① PBWT on compressed blocks



② Estimated haplotype insertion

③ S nearest neighbor candidate searches (S = 2)



④ Finding the S nearest neighbors (S = 2)

(See caption on next page.)

Supplemental Figure S4: Illustration of oblivious phasing via compressed haplotype selection. We illustrate the workflow of our oblivious compressed haplotype selection, described in detail in Supplemental Note S2. **(1)** Each compressed reference panel block is transformed via PBWT into a prefix tree representation (Algorithm 1). In this tree, each node represents a group of haplotypes that share a common prefix, starting from the root at the beginning of the block region. Each leaf corresponds to a unique haplotype within the block. The nodes are ordered by positional prefixes, i.e., the reverse of suffixes, and block-level divergences (i.e., divergences within the block region) are maintained between adjacent nodes. The 0-edges are shown in green, and the 1-edges in blue, with edge thickness corresponding to the number of haplotypes following that path. **(2)** The portion of the estimated haplotype within the block region is inserted into the tree, while keeping the inserted information separate (Algorithm 2). This includes the updated block-level divergences and positional prefix orders that define the path through the tree. **(3)** At a search position, neighboring nodes with the smallest block-level divergence relative to the estimated haplotype are searched to construct the haplotype candidate set. This set includes original haplotypes along the paths of these nodes, and nodes are searched until at least S original haplotypes with the smallest block-level divergence are identified (Algorithm 3). The candidate set includes the S nearest neighbors because smaller block-level divergences imply smaller global divergences. In this example, where $S = 2$, the candidate set contains six original haplotypes in the nearest node. However, because their block-level divergence is 0, their global divergences are unknown, leaving the order ambiguous. Finding the two nearest neighbors requires further ranking in the next step. **(4)** In our oblivious solution, all original haplotypes within each node are ranked by their global divergences relative to the estimated haplotype at the starting position of the block for the top S haplotypes. The global divergence list can be recursively computed from the estimated haplotype inserted in previous prefix trees (Algorithm 6). By Equation 3, knowing the global divergence at the block’s starting position is sufficient to calculate the global divergence at any position within the block, assuming a block-level divergence of 0 (which is the case in this example). The rank of each node is computed using an oblivious merge-sorting algorithm (Algorithm 4): the rank of each leaf node is determined by obliviously sorting the original haplotypes in the node by their divergences from the global divergence list for the top S haplotypes. The rank of each node at higher levels is calculated by merging the ranks of nodes from the level below for the top S haplotypes, following the prefix tree structure. Once the ranks are determined for all nodes, the ranks at the search position are used to resolve the order of haplotype candidates, ultimately determining the S nearest neighbors (Algorithm 5).

$$\begin{array}{l}
\text{Genotype} \\
\text{graph node} \\
X_i = a
\end{array}
\left\{ \begin{array}{l}
P(X_{i+1} = b \mid X_i = a) \\
\left[\begin{array}{cccc}
0.0 & 4.2\text{e-}4 & 1.7\text{e-}10 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 \\
4.5\text{e-}1 & 3.4\text{e-}1 & 1.1\text{e-}4 & 1.1\text{e-}4 \\
8.9\text{e-}5 & 1.9\text{e-}1 & 1.3\text{e-}7 & 1.3\text{e-}7
\end{array} \right]
\end{array} \right.$$

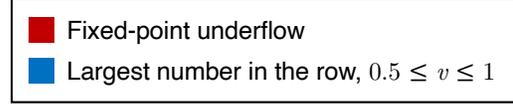
Genotype graph node $X_{i+1} = b$

A. Normalized matrix in standard fixed points

$$\begin{array}{l}
2^{-11} \\
2^{-66} \\
2^{-1} \\
2^{-2}
\end{array}
\times
\left[\begin{array}{cccc}
0.0 & 8.6\text{e-}1 & 3.5\text{e-}7 & 0.0 \\
6.6\text{e-}1 & 5.6\text{e-}1 & 2.6\text{e-}5 & 2.6\text{e-}5 \\
9.0\text{e-}1 & 6.8\text{e-}1 & 2.2\text{e-}4 & 2.2\text{e-}4 \\
3.6\text{e-}4 & 7.6\text{e-}1 & 5.2\text{e-}7 & 5.2\text{e-}7
\end{array} \right]$$

Scaling factors Fixed-point numbers

B. Dynamic fixed-point matrix



Scales adjusted (underlines)

$$\begin{array}{l}
2^{-1} \\
2^{-1} \\
2^{-1} \\
2^{-1}
\end{array}
\times
\left[\begin{array}{cccc}
0.0 & 8.4\text{e-}4 & 3.4\text{e-}10 & 0.0 \\
0.0 & 0.0 & 0.0 & 0.0 \\
9.0\text{e-}1 & 6.8\text{e-}1 & 2.2\text{e-}4 & 2.2\text{e-}4 \\
1.8\text{e-}4 & 3.8\text{e-}1 & 2.6\text{e-}7 & 2.6\text{e-}7
\end{array} \right]$$

↓ Summation by column

$$2^{-1} \times \left[\begin{array}{cccc}
9.0\text{e-}1 & 1.06 & 2.2\text{e-}4 & 2.2\text{e-}4
\end{array} \right]$$

↓ Adjust the scale

$$2^0 \times \left[\begin{array}{cccc}
4.5\text{e-}1 & 5.3\text{e-}1 & 1.1\text{e-}4 & 1.2\text{e-}4
\end{array} \right]$$

C. Sum-by-column of dynamically scaled matrix

Supplemental Figure S5: Illustration of our dynamic fixed-point arithmetic. (A) A transition probability matrix for $P(X_{i+1} = a \mid X_i = b)$, where X_i is a genotype graph node at position i and X_{i+1} at $i + 1$, is shown using standard fixed points. Even if the matrix is normalized, probabilities in an entire row can underflow (represented in red), resulting in an accuracy loss. (B) The dynamic scaling technique is applied to the matrix, where the scaling factors in powers of two are maintained per row (or column) so that the largest probability in each row (or column) remains between 0.5 and 1.0 (represented in blue). (C) A sum-by-column operation is demonstrated on the dynamically scaled matrix. First, the rows with smaller scales (marked with underlines) are adjusted to match the scale of the row with the largest scale to minimize precision loss. This may still result in underflows in some rows but the overall accuracy of the final sum is better preserved. Lastly, the matrix is summed by column, and the scale is readjusted.

Fixed-point arithmetic	Dynamic scaling		Log domain	
	# function calls	Approx. time	# function calls	Approx. time
Addition	2.9×10^{10}	19 s	1.1×10^{12}	17.5 m
Subtraction	3.1×10^6	< 1 s	1.2×10^{11}	2.4 m
Multiplication	3.6×10^9	5 s	9.8×10^{11}	21.1 m
Division	6.7×10^5	< 1 s	-	-
		Total: 24 s		Total: 41.0 m

Supplemental Table S1: Our dynamic fixed points enable practical performance of TX-Phase. We compare the runtime performance between dynamic fixed-point arithmetic techniques, introduced in this work, and the log-domain fixed-point arithmetic techniques from prior work (Dokmai et al. 2021). The comparison is based on a one-sample run on Chromosome 20 using the 400K UKB reference panel. Fixed-point division is implemented in constant time using a bit-wise long division algorithm. The performance of each scheme is measured by the number of function calls to the underlying standard fixed-point arithmetic operations during the run. We then estimate the required runtime by multiplying the number of operations with the observed per-operation costs based on a micro-benchmark, provided in Supplemental Table S2. ‘s’ stands for seconds, and ‘m’ stands for minutes. The results show that dynamic fixed-point techniques lead to over 100x speedup for TX-Phase compared to the existing techniques.

	# of function calls to standard fixed-point operations		
	Addition	Subtraction	Multiplication
Log-domain addition	55	6	51
Log-domain subtraction	54	5	51
Log-domain multiplication	1	-	-
Log-domain division	-	1	-

Supplemental Table S2: Per-operation cost of log-domain fixed-point arithmetic operations. We analyzed the implementation of log-domain fixed-point arithmetic operations in prior work (Dokmai et al. 2021) to assess the costs associated with each arithmetic operation, related to Supplemental Table S1. Specifically, since log-domain fixed points are implemented on top of standard fixed points, we examined the number of function calls made to the underlying standard fixed-point arithmetic operations. This analysis provides insight into how the costs of arithmetic operations multiply when performed in the log domain, in contrast to our dynamic fixed-point arithmetic, where each operation calls exactly one underlying standard fixed-point operation. The results show that log-domain addition and subtraction require a significant number of function calls due to their piecewise polynomial approximation implementations. Consequently, using log-domain fixed points would impose a prohibitively high computational cost for HMM inference in TX-Phase, as shown in Supplemental Table S1. On the other hand, log-domain multiplication and division translate to exactly one addition and one subtraction, respectively, in the normal domain.

	Runtime (secs)		Memory usage (MB)	
	Mean	RSD (%)	Mean	RSD (%)
TX-Phase (SGX)	1002.9	0.24	15423.4	0.00050
SHAPEIT4	67.7	2.19	689.6	6.90
Eagle2	25.6	0.94	1153.6	1.87

Supplemental Table S3: Consistency of performance measurements. We estimated the relative standard deviation (RSD) of the phasing runtime and memory usage statistics for TX-Phase (running in SGX secure enclave), SHAPEIT4, and Eagle2 across 20 different input samples held out from the 1000 Genomes reference panel using the first one third of Chromosome 20. The results show highly consistent measurements across different samples.

Notation	Definition
N	The number of haplotypes in the original reference panel.
U^b	The number of unique haplotypes in the compressed reference panel block b .
H^b	Compressed reference panel block b , where H_{id}^b denotes unique haplotype id in the block, and $H_{id}^b[i]$ denotes the allele of unique haplotype $id \in \{0, \dots, U^b - 1\}$ at genomic position $i \in \{0, \dots, l^b - 1\}$ within the block region. l^b denotes the length of the block region in the number of genomic positions.
l^b	The length of the block region b in the number of genomic positions.
B^b	The global genomic position (i.e., the position in the entire sequence) at the start of block b . Note that $B^b + l^b = B^{b+1}$
$trie_i^b$	Prefix tree level i of block region b , where $i = 0$ indicates the root level and $i = l^b$ the leaf level. $trie_i^b$ is a list of tree nodes.
node	A prefix tree node, consisting of the 0-edge (node.0), 1-edge (node.1), the divergence with the node above it (node.div), and the set of unique haplotypes in the node's path (node.haps).
h^b	An estimated haplotype of the target sample within block b 's region. $h^b[i]$ denotes the allele at block-level genomic position $i \in \{0, \dots, l^b - 1\}$.
$inst_i^b$	An haplotype insertion struct at block-level position i in block b , marking the insertion of allele $h^b[i - 1]$ into the prefix tree level $trie_i^b$. $inst_i^b.ord$ is the order of the inserted haplotype in $trie_i^b$, where $trie_i^b[inst_i^b.ord]$ indicates the node immediately below the inserted haplotype. $inst_i^b.div$ is the block-level divergence between the inserted haplotype and the node immediately above it, $trie_i^b[inst_i^b.ord - 1]$, and $inst_i^b.div'$ is the block-level divergence between the inserted haplotype and the node immediately below it, $trie_i^b[inst_i^b.ord]$.
C_i^b	The nearest-neighbor candidate list at position i in block b , where $j \in C_i^b$ represents node $trie_i^b[j]$, and j is ordered (ascending) in C_i^b by block-level divergence between node $trie_i^b[j]$ and estimated haplotype h^b at position i .
D_p	The global divergence (i.e., divergence in the entire sequence) list at position p , where for each original haplotype $id \in \{0, \dots, N - 1\}$, $D_p[id]$ is the global divergence between original haplotype id and an estimated haplotype at position p .
\tilde{D}_i^b	The block-level divergence (i.e., divergence within a block region) list at position i for block b , where for each original haplotype $id \in \{0, \dots, N - 1\}$, $\tilde{D}_i^b[id]$ is the block-level divergence between original haplotype id and an estimated haplotype in the block h^b at block-level position i .
R_i^b	The rank list of nodes in prefix tree $trie_i^b$, where $R_i^b[j]$ is the rank for node $trie_i^b[j]$. Rank is determined by the global divergences of original haplotypes.

Supplemental Table S4: Notations for our oblivious compressed haplotype selection algorithms. Supporting material for Supplemental Note S2.

Operations	Definition
$L[i : j]$	Sublist of list L between position i and j (inclusive).
$\text{prefix}(\text{node})$	The prefix of the unique haplotypes in node 's path in the prefix tree. Namely, suppose that $\text{node} \in \text{trie}_i^b$, then $\text{prefix}(\text{node}) := H_{\text{id}}^b[0 : i - 1]$ for any $\text{id} \in \text{node.haps}$.
$L \parallel i$	Concatenate list L with item or list i .
for $i \in L$ do	Iterate all items i in list L in the designated order or set L in an arbitrary order.
$\text{imap}^b(\text{id})$	Map unique haplotype $\text{id} \in \{0, \dots, U^b - 1\}$ to the set of original haplotypes (by their IDs) that are compressed to it in block b 's region.
$\text{lmap}^b(I)$	Map unique haplotype set I to the combined set of original haplotypes (by their IDs) that are compressed to each unique haplotype in I in block b 's region, i.e., $\text{lmap}^b(I) := \bigcup_{\text{id} \in I} \text{imap}^b(\text{id})$.

Supplemental Table S5: Operations for our oblivious compressed haplotype selection algorithms. Supporting material for Supplemental Note S2.

Supplemental Note S1: Additional algorithmic modifications in TX-Phase for oblivious phasing

In addition to TX-Phase’s novel algorithms for compressed haplotype selection designed to accelerate oblivious phasing and dynamic fixed-point arithmetic for accurate constant-time computation (see Methods, Supplemental Figure S4, Supplemental Figure S5, and Supplemental Note S2), TX-Phase applies several other key algorithmic transformations to SHAPEIT4. These changes ensure that sensitive user data remains completely decoupled from observable program behavior, including memory access and timing patterns, while maintaining practical computational performance. Sensitive components of the code are systematically detected by our secure typing system, where violations of our security rules are reported as compilation errors. The following summarizes our algorithmic modifications in each step of the phasing workflow.

Initialization. In the MCMC-based phasing approach, the initialization step estimates the initial phases of the target sample, serving as a starting point for the optimization. For this step, SHAPEIT4 uses the positional Burrows-Wheeler transform (PBWT) to order the target sample among the reference haplotypes and estimates the phase of each genotype independently across sites based on a heuristic scoring function taking the nearby (i.e., most similar) reference haplotypes in the PBWT array as input. However, obviously ordering target samples in the PBWT array is computationally expensive, and the calculation of the scoring function also involves sensitive logical branching that depends on the target sample. Instead, TX-Phase efficiently selects the nearest neighbors according to their divergences from the prefix trees based on our compressed reference panel—the same method we apply in constructing the conditioned reference panel in later MCMC steps. In addition, it modifies the scoring function to run in constant time and with a deterministic workflow independent of the input sample using side-channel mitigation techniques in Methods.

Genotype graph construction. In SHAPEIT4, the structure of a genotype graph is determined by the positions of heterozygous sites along the chromosome, where a segment is created for each consecutive set of 3 heterozygous sites. In TX-Phase, this presents a challenge since the positions of heterozygous sites cannot be disclosed. To address this, TX-Phase conceals segment boundaries by treating every genomic position as a potential segment boundary and using a secret bitmap to track the true boundaries during graph processing.

Forward-backward algorithm. The forward-backward algorithm for HMM inference computes transition probabilities between genotype graph nodes in adjacent segments. To hide segment boundaries, and thus the positions of heterozygous sites, TX-Phase follows the oblivious genotype graph structure, treating all genomic positions as segment boundaries. This is achieved by modifying the algorithm to pretend to update at all potential boundaries while correctly handling the true boundaries encoded in the oblivious genotype graph. Additionally, TX-Phase hides the size of the conditioned reference panel, representing the number of reference haplotypes considered the nearest neighbors of the user’s sample. In certain scenarios, this information could potentially expose sensitive genotypes. To mitigate this risk, our algorithm pads the conditioned reference panel to a fixed upper-bounded size with dummy haplotypes. These dummies are excluded from downstream HMM inference using a map that distinguishes them from real haplotypes.

Forward sampling and Viterbi. Similar to the forward-backward algorithm, both sampling the phases from an HMM and constructing the most likely phases using the Viterbi algorithm involve traversing the genotype graph, requiring modifications to protect the heterozygous positions. TX-Phase follows the oblivious genotype graph structure as before, treating all genomic positions as segment boundaries. Forward sampling and Viterbi are modified to incorporate all positions as potential boundaries while ensuring that only the true segment boundaries, tracked in the secret map, affect the final outcome.

Pruning. Pruning is a heuristic step performed during the MCMC sampling procedure, where high-confidence phases of adjacent segments are iteratively merged to reduce the search space of subsequent iterations. The segments to merge are identified by those with high transition probabilities between adjacent nodes in the genotype graph. In SHAPEIT4, pruning is guided by a heuristic that ranks candidate merges based on the sum of the top transition probabilities and the entropy of the joint distribution between adjacent segments. For oblivious computation, TX-Phase pretends to prune at all genomic positions, while keeping track of the true segment boundary positions according to the oblivious genotype graph. Instead of ranking the segments to merge, it applies a greedy left-to-right pruning strategy, merging segments as they

come when the confidence level exceeds a predefined threshold.

Determining phasing genomic windows. SHAPEIT4 divides the genome into windows, which are individually analyzed for HMM-based phasing before combining them together to produce the final, global phases. These windows align with genotype graph segment boundaries to ensure they always encompass full segments. To remove dependence on heterozygous positions, TX-Phase instead defines windows based on a fixed centi-Morgan distance with a permissive overlap between adjacent windows to ensure that at least one full segment is shared between adjacent windows for the purposes of combining their phases.

Supplemental Note S2: Our oblivious algorithms for compressed haplotype selection

This section provides additional details and pseudocodes for compressed haplotype selection in TX-Phase, introduced in Methods. The goal of this task is to select a small subset of haplotypes, referred to as *nearest neighbors*, from the reference panel based on their similarity to the phasing estimates. Similarity is determined by identifying the longest shared identity-by-state (IBS) segments. To efficiently identify these haplotypes without exposing the phasing estimates through side channels, our algorithms improve upon the positional Burrows-Wheeler transform (PBWT) (Durbin 2014) used in SHAPEIT4 (Delaneau et al. 2019).

In the PBWT-based search, random genomic positions in the sequence are chosen to locate the longest common suffix (LCS) between the reference haplotypes and the phasing estimates, representing the longest IBS segments. For each selected position, the S nearest neighbors with the longest LCS relative to the phasing estimates are identified. To achieve this efficiently, the haplotypes in the reference panel—encoded as 0’s and 1’s for reference and alternative alleles, respectively—are iteratively transformed via PBWT from the first to the last genomic position to determine their *positional prefix orders* and *divergences* at each position.

The positional prefix order represents the lexicographic ordering of haplotypes based on their reverse prefix, i.e., the sequence of alleles in reverse genomic order starting from the current position and extending back to the first position. A divergence between two haplotypes is defined as the earliest genomic position where they begin to share a common suffix before the current position, which corresponds to the LCS length between the two haplotypes up to that point. It is important to note that the reverse prefix and the suffix of a haplotype at a given position describe the same contiguous region of the haplotype; however, we refer to the reverse prefix when discussing haplotype ordering in the context of PBWT.

In the original PBWT algorithms, a *divergence array* is maintained for each position, storing the divergences between adjacent haplotypes in the positional prefix order. The positional prefix order and divergence arrays possess key properties that facilitate efficient searches for the S nearest neighbors with the longest LCS: (1) haplotypes with longer LCS (i.e., smaller divergence) are positioned closer together in the positional prefix order, and (2) the divergence between any two haplotypes can be calculated by finding the maximum of all divergences between them in the divergence array.

Our algorithms enhance these PBWT approaches by applying PBWT to compressed reference panels, enabling oblivious and efficient nearest-neighbor searches on large datasets. The core idea is to independently apply PBWT to compressed blocks of the reference panel, generating *block-level* divergences (i.e., divergences within a block). These divergences are used to identify nearest-neighbor candidates at each position with the longest block-level LCS. The global divergences (i.e., divergences across the entire sequence) are then computed among these candidates to identify the nearest neighbors with the longest global LCS at a given position.

We first outline the steps of our algorithmic workflow. Summaries of our notation and key operations are provided in Supplemental Tables S4 and S5, respectively. A graphical illustration of our algorithms is provided in Supplemental Figure S4.

Step 1: PBWT on compressed blocks and prefix tree transformation (Algorithm 1).

This algorithm transforms a compressed reference panel block via PBWT into a prefix tree representation of PBWT arrays, making haplotype insertion and nearest-neighbor searches both oblivious and efficient. The prefix trees compactly preserve all information from the PBWT arrays, grouping haplotypes with a common prefix into a tree node at each position, allowing searches to be performed on groups of haplotypes.

Step 2: Estimated haplotype insertion (Algorithm 2). This algorithm inserts the estimated haplotypes of the target sample for the current Markov Chain Monte Carlo (MCMC) iteration into the prefix trees, without revealing their paths within the trees, as such leakage could allow inference of the target sample.

Step 3: Nearest-neighbor candidate searches (Algorithm 3). This algorithm quickly identifies a set of nearest-neighbor candidates that is guaranteed to contain the S nearest neighbors of the estimated haplotype at each search position by leveraging the compact prefix tree structure. The S nearest neighbors are the original haplotypes with the smallest global divergence (i.e., longest LCS) with the estimated haplotype at the search position.

Step 4: Finding the S nearest neighbors (Algorithms 4 and 5). The algorithms in this step narrow down the haplotype candidates in the candidate node list to identify the S nearest neighbors. This is achieved by ranking the candidates based on their global divergences relative to the estimated haplotype and eliminating those with the largest global divergences. The process is efficient due to the relatively small size of the candidate node list, which is pre-ordered by block-level divergences. Smaller block-level divergences imply smaller global divergences. Therefore, the primary task of these algorithms is to resolve any ambiguities in the ranking based on block-level divergences by computing global divergences as needed.

Step 5: Computing global divergence lists (Algorithm 6). This algorithm computes the global divergence lists at the start of the block for all compressed reference panel blocks. These lists are used by Algorithm 4 to rank haplotypes and narrow down the candidate haplotype set to the S nearest neighbors.

Step 6: Constructing a conditioned reference panel. In this final step, the nearest neighbors are combined to form the conditioned haplotype set, which is then used to build the conditioned reference panel for HMM inference.

We now provide the details of each step with pseudocode.

Step 1: PBWT on compressed blocks and prefix tree transformation (Algorithm 1)

This algorithm (illustrated in Supplemental Figure S4.1) transforms a compressed reference panel block via PBWT into a prefix tree representation of PBWT arrays, making both haplotype insertion and nearest-neighbor searches oblivious and efficient. The prefix trees compactly preserve all information from the PBWT arrays, grouping haplotypes with a common prefix into a tree node at each position, allowing searches to be performed on groups of haplotypes. This step does not involve the sensitive target sample and thus does not require an oblivious implementation.

Algorithm 1 constructs one prefix tree for each compressed reference panel block with the following structure:

- *Prefix tree layers.* The prefix tree for block b consists of tree levels trie_i^b for each block-level position (i.e., position within the block region) $i \in \{0, \dots, l^b\}$, where $i = 0$ represents the root level and $i = l^b$ represents the leaf level. Each tree level trie_i^b contains a list of tree nodes.
- *Nodes.* A tree node node in tree level trie_i^b consists of the set node.haps of unique haplotypes in the block that share a common prefix from the root level to level i . Namely, let H^b represent the compressed reference panel block b , H_{id}^b the unique haplotype $\text{id} \in \{0, \dots, U^b - 1\}$ in block b , and $H_{\text{id}}^b[i] \in \{0, 1\}$ the allele at block-level position i of unique haplotype id . Then, for all $\text{id} \in \text{node.haps}$ of any node $\text{node} \in \text{trie}_i^b$, the prefixes $\text{prefix}(\text{node}) := H_{\text{id}}^b[0 : i - 1]$ between positions 0 and $i - 1$ (inclusive) are identical.
- *Prefix tree node order.* The nodes $\text{node} \in \text{trie}_i^b$ are ordered according to the positional prefix order based on the reverse of the prefix $\text{prefix}(\text{node})$.
- *Edges.* A node $\text{node} \in \text{trie}_i^b$ is connected to child nodes in the next tree level, trie_{i+1}^b , via 0-edge node.0 and 1-edge node.1 , representing the branching of 0 and 1 alleles, respectively, at the current position of the unique haplotypes in node.haps . Namely, if $\text{node.0} = u$ and $\text{node.1} = v$, then $\text{trie}_{i+1}^b[u].\text{haps} \subseteq \text{node.haps}$ and $\text{trie}_{i+1}^b[v].\text{haps} \subseteq \text{node.haps}$, such that unique haplotype $\text{id} \in \text{trie}_{i+1}^b[u].\text{haps}$ if and only if $\tilde{H}_{\text{id}}^b[i] = 0$, and $\text{id} \in \text{trie}_{i+1}^b[v].\text{haps}$ if and only if $\tilde{H}_{\text{id}}^b[i] = 1$.
- *Divergences.* Each node $\text{node} \in \text{trie}_i^b$ contains the block-level divergence node.div with the preceding node in the positional prefix order of trie_i^b .
- *Prefix tree root level.* The root tree level trie_0^b consists of a singleton list containing the root node, which includes all U^b unique haplotypes, i.e., $\text{node.haps} = \{0, \dots, U^b - 1\}$, and $\text{node.div} = 0$.
- *Prefix tree leaf level.* The leaf tree level $\text{trie}_{l^b}^b$ contains leaf nodes with no children, where each node holds a singleton set of one unique haplotype. Thus, every path in the tree represents a unique haplotype in the block.

Intuitively, Algorithm 1, which generates the above data structure, functions similarly to Algorithm 2 in (Durbin 2014). However, Algorithm 1 improves this process by independently applying the transformation to each compressed reference panel block, thereby generating block-level positional prefix orders and divergences. The algorithm groups unique haplotypes with a common prefix (i.e., divergence 0) into the same prefix tree node. To construct the full prefix tree, Algorithm 1 is applied iteratively at each level, from $i = 0$ to $i = l^b - 1$.

Algorithm 1 Build prefix tree level $i + 1$ for compressed block b . Given prefix tree level trie_i^b and compressed block H^b , build trie_{i+1}^b and connect the edges of the nodes in trie_i^b to their child nodes in trie_{i+1}^b . Save the number of 0-edges in level i to z_i^b . b is implied and thus omitted for clarity.

Input: trie_i, H

Output: $\text{trie}_i, \text{trie}_{i+1}, z_i$

$u \leftarrow 0, v \leftarrow 0, p \leftarrow i + 1, q \leftarrow i + 1$

Create empty lists L_0 and L_1

for node $\in \text{trie}_i$ **do**

$p \leftarrow \max(p, \text{node.div}), q \leftarrow \max(q, \text{node.div})$

 Create empty nodes x and y

for id $\in \text{node.haps}$ **do**

if $H_{\text{id}}[i] = 0$ **then**

if node.0 does not exist **then**

 node.0 $\leftarrow u, x.\text{div} \leftarrow p, p \leftarrow 0$

end if

$x.\text{haps} \leftarrow x.\text{haps} \cup \{\text{id}\}$

else

if node.1 does not exist **then**

 node.1 $\leftarrow v, y.\text{div} \leftarrow q, q \leftarrow 0$

end if

$y.\text{haps} \leftarrow y.\text{haps} \cup \{\text{id}\}$

end if

end for

if $x.\text{haps}$ is not empty **then**

$L_0[u] \leftarrow x, u \leftarrow u + 1$

end if

if $y.\text{haps}$ is not empty **then**

$L_1[v] \leftarrow y, v \leftarrow v + 1$

end if

end for

$\text{trie}_{i+1} \leftarrow L_0 \parallel L_1$

for node $\in \text{trie}_i$ **do**

if node.1 exists **then**

 node.1 $\leftarrow \text{node.1} + |L_0|$

\triangleright Offset 1-edges to the positions after 0-edges

end if

end for

$z_i \leftarrow |L_0|$

\triangleright Save the number of 0-edges

Step 2: Estimated haplotype insertion (Algorithm 2)

This algorithm (illustrated in Supplemental Figure S4.2) inserts the estimated haplotypes of the target sample for the current Markov Chain Monte Carlo (MCMC) iteration into the prefix trees, without revealing their paths within the trees, as such leakage could allow inference of the target sample. Although each iteration results in two estimated haplotypes per target sample, the insertion and nearest-neighbor searches are independent for each haplotype. Hence, we treat each haplotype independently for the remaining algorithms.

For each block b , let h^b represent the portion of the estimated haplotype within block b 's region, and $h^b[i] \in \{0, 1\}$ for $i \in \{0, \dots, l^b - 1\}$ denote the alleles of the haplotype within that region. Algorithm 2 inserts h^b into the prefix tree for block b , while storing the insertion orders and divergences separately in the haplotype insertion structs, inst_i^b , one for each block-level position i . Struct inst_i^b consists of the following elements: $\text{inst}_i^b.\text{ord}$, representing the order of h^b resulting from inserting allele $h^b[i - 1]$ into trie_i^b , where node $\text{trie}_i^b[\text{inst}_i^b.\text{ord}]$ is defined as the node directly below h^b at position i ; $\text{inst}_i^b.\text{div}$, representing the block-level divergence between h^b and the node directly *above* it, $\text{trie}_i^b[\text{inst}_i^b.\text{ord} - 1]$; and $\text{inst}_i^b.\text{div}'$, representing the block-level divergence between h^b and the node directly *below* it, $\text{trie}_i^b[\text{inst}_i^b.\text{ord}]$, with $\text{inst}_i^b.\text{div}'$ representing the updated the value of $\text{trie}_i^b[\text{inst}_i^b.\text{ord}].\text{div}$ after insertion. At the root level, we define $\text{inst}_0^b.\text{ord} = 0$, $\text{inst}_0^b.\text{div} = 0$, and $\text{inst}_0^b.\text{div}' = 0$.

Algorithm 2 iteratively inserts allele $h^b[i]$ into prefix tree level trie_{i+1}^b , from block-level position $i = 0$ to $i = l^b - 1$, generating struct inst_{i+1}^b for each position. At position i , the algorithm scans upwards and downwards from the current haplotype order, $\text{inst}_i^b.\text{ord}$, in trie_i^b until it identifies the first nodes above and below with an $h^b[i]$ -edge. Suppose $h^b[i] = E$ for some allele E and the identified node above is at order u and the node below at order v . Then, the next order of h^b , $\text{inst}_{i+1}^b.\text{ord}$, is set to $\text{trie}_i^b[u].E + 1$, effectively inserting h^b between node $\text{trie}_i^b[u].E$ and node $\text{trie}_i^b[v].E$ in tree level trie_{i+1}^b , following the positional prefix order.

To compute the block-level divergences $\text{inst}_{i+1}^b.\text{div}$ and $\text{inst}_{i+1}^b.\text{div}'$, the algorithm uses the PBWT property, taking the maximum of divergences between h^b and nodes u and v at position i :

$$\text{inst}_{i+1}^b.\text{div} = \max \left(\text{inst}_i^b.\text{div}, \text{trie}_i^b[j - 1].\text{div}, \text{trie}_i^b[j - 2].\text{div}, \dots, \text{trie}_i^b[u - 1].\text{div} \right) \quad (1)$$

$$\text{inst}_{i+1}^b.\text{div}' = \max \left(\text{inst}_i^b.\text{div}', \text{trie}_i^b[j + 1].\text{div}, \text{trie}_i^b[j + 2].\text{div}, \dots, \text{trie}_i^b[v].\text{div} \right), \quad (2)$$

where $j := \text{inst}_i^b.\text{ord}$.

In our oblivious implementation of Algorithm 2, the **for** loops always run between the lower bound (at 0 or 1) and the upper bound (at $|\text{trie}_i^b| - 1$), pretending to compute within the loop to mask the actual value of $\text{inst}_i^b.\text{ord}$. The **break** statement does not terminate the loop but triggers a condition where computation continues without further updates to the relevant variables.

Algorithm 2 Insert an estimated haplotype at position $i + 1$ within block b . Given haplotype insertion struct inst_i^b , allele $h^b[i]$, tree level trie_i^b , and the number of 0-edges in trie_i^b , z_i^b , build struct inst_{i+1}^b . b is implied and thus omitted for clarity.

Input: $\text{inst}_i, h[i], \text{trie}_i, z_i$

Output: inst_{i+1}

Create an empty struct inst_{i+1}

▷ Compute $\text{inst}_{i+1}.\text{ord}$ and $\text{inst}_{i+1}.\text{div}$ below

$d \leftarrow \text{inst}_i.\text{div}$; $\text{found} \leftarrow \text{false}$

for $j = \text{inst}_i.\text{ord} - 1 \rightarrow 0$ **do**

▷ Scan upward for the first $h[i]$ -edge

$\text{node} \leftarrow \text{trie}_i[j]$

if $h[i] = 0$ and $\text{node}.0$ exists **then**

$\text{inst}_{i+1}.\text{ord} \leftarrow \text{node}.0 + 1$; $\text{inst}_{i+1}.\text{div} \leftarrow d$; $\text{found} \leftarrow \text{true}$

break

else if $h[i] = 1$ and $\text{node}.1$ exists **then**

$\text{inst}_{i+1}.\text{ord} \leftarrow \text{node}.1 + 1$; $\text{inst}_{i+1}.\text{div} \leftarrow d$; $\text{found} \leftarrow \text{true}$

break

end if

$d \leftarrow \max(d, \text{node}.\text{div})$

▷ Compute Equation 1

end for

if not found **then**

if $h[i] = 0$ **then**

$\text{inst}_{i+1}.\text{ord} \leftarrow 0$

else

$\text{inst}_{i+1}.\text{ord} \leftarrow z_i$

end if

$\text{inst}_{i+1}.\text{div} \leftarrow i + 1$

end if

▷ Compute $\text{inst}_{i+1}.\text{div}'$ below

$d \leftarrow \text{inst}_i.\text{div}'$; $\text{found} \leftarrow \text{false}$

for $j = \text{inst}_i.\text{ord} \rightarrow |\text{trie}_i| - 1$ **do**

▷ Scan downward for the first $h[i]$ -edge

$\text{node} \leftarrow \text{trie}_i[j]$

if $j > \text{inst}.\text{ord}$ **then**

$d \leftarrow \max(d, \text{node}.\text{div})$

▷ Compute Equation 2

end if

if ($h[i] = 0$ and $\text{node}.0$ exists) or ($h[i] = 1$ and $\text{node}.1$ exists) **then**

$\text{inst}_{i+1}.\text{div}' \leftarrow d$; $\text{found} \leftarrow \text{true}$

break

end if

end for

if not found **then**

$\text{inst}_{i+1}.\text{div}' \leftarrow i + 1$

end if

Step 3: Nearest-neighbor candidate searches (Algorithm 3)

This algorithm (illustrated in Supplemental Figure S4.3) quickly identifies a set of nearest-neighbor candidates that is guaranteed to contain the S nearest neighbors of the estimated haplotype at each search position by leveraging the compact prefix tree structure. The S nearest neighbors are the original haplotypes with the smallest global divergence (i.e., longest LCS) with the estimated haplotype at the search position.

In the PBWT-based search, several genomic positions in the sequence are randomly chosen for a nearest-neighbor search. Suppose position i in block b is selected; this algorithm identifies the set of original haplotype candidates that includes all S original haplotypes with the smallest global divergences relative to the estimated haplotype at the given position. Since smaller block-level divergences imply smaller global divergences, these candidate original haplotypes can be found by searching for nodes containing original haplotypes with the smallest block-level divergences with the estimated haplotype until at least S original haplotypes are identified. If the candidate set contains more than S haplotypes, it will be narrowed down to exactly S haplotypes in the next step by their global divergences.

Algorithm 3 represents the candidate set at position i in block b using the candidate node list C_i^b , where for all $j \in C_i^b$, j represents node $\text{trie}_i^b[j]$ and is ordered by the smallest block-level divergence between the node (specifically, the node's prefix, $\text{prefix}(\text{trie}_i^b[j])$) and the estimated haplotype h^b . The candidate node list forms the candidate set by combining all the original haplotypes along the paths of these nodes. Namely, let $\text{imap}^b(\text{id})$ be the index map between the unique haplotype id in block b and the set of original haplotypes compressed to it within the block b 's region, and define $\text{lmap}^b(\text{node.haps}) := \bigcup_{\text{id} \in \text{node.haps}} \text{imap}^b(\text{id})$. The candidate set is the combined set $\bigcup_{j \in C_i^b} \text{lmap}^b(\text{trie}_i^b[j].\text{haps})$, which includes at least S members.

Intuitively, Algorithm 3 constructs the candidate list C_i^b by performing a local scan for nodes with the smallest block-level divergences relative to the estimated haplotype h^b in the prefix tree. This process leverages the properties of PBWT, where: (1) nodes with the smallest block-level divergences relative to the estimated haplotype h^b are nearest to its order, $\text{inst}_i^b.\text{ord}$, in trie_i^b , and (2) the nodes' block-level divergences with h^b can be dynamically computed from the nodes' divergences, similar to how Equations 1 and 2 calculate the divergence between h^b and its neighboring nodes. This approach allows Algorithm 3 to efficiently scan locally around order $\text{inst}_i^b.\text{ord}$ in trie_i^b to identify the candidate nodes with the smallest block-level divergences with respect to h^b .

In our oblivious implementation of Algorithm 3, we conceal the value of the haplotype insertion struct inst_i^b by computing candidate lists for all possible values of $\text{inst}_i.\text{ord}$, and then employing an oblivious algorithm (Methods) to select the candidate list that corresponds to the true value of $\text{inst}_i^b.\text{ord}$.

Algorithm 3 Search for nearest-neighbor candidates at positions i . Given haplotype insertion struct inst_i^b for the estimated haplotype h^b and tree level trie_i^b , compute the list C_i^b of candidate nodes with the smallest block-level divergences with h^b in block b . b is implied and thus omitted for clarity.

Input: $\text{inst}_i, \text{trie}_i$

Output: C_i

Create an empty list C_i

$n \leftarrow 0$

▷ Keep the count of original haplotype candidates

$u \leftarrow \text{inst}_i.\text{ord} - 1, v \leftarrow \text{inst}_i.\text{ord}$

$p \leftarrow \text{inst}_i.\text{div}, q \leftarrow \text{inst}_i.\text{div}'$

while $n < S$ **do**

if $p \leq q$ **then**

$C_i \leftarrow C_i \parallel u$

$n \leftarrow n + |\text{lmap}(\text{trie}_i[u].\text{haps})|$

▷ Add the node's original haplotypes

$p \leftarrow \max(p, \text{trie}_i[u].\text{div})$

$u \leftarrow u - 1$

else

$C_i \leftarrow C_i \parallel v$

$n \leftarrow n + |\text{lmap}(\text{trie}_i[v].\text{haps})|$

▷ Add the node's original haplotypes

$q \leftarrow \max(q, \text{trie}_i[v + 1].\text{div})$

$v \leftarrow v + 1$

end if

end while

Step 4: Finding the S nearest neighbors (Algorithms 4 and 5)

The algorithms in this step (illustrated in Supplemental Figure S4.4) narrow down the haplotype candidates in the candidate node list to identify the S nearest neighbors. This is achieved by ranking the candidates based on their global divergences relative to the estimated haplotype and eliminating those with the largest global divergences (i.e., smallest global LCS). The process is efficient due to the relatively small size of the candidate node list, which is pre-ordered by block-level divergences. Smaller block-level divergences imply smaller global divergences. Therefore, the primary task of these algorithms is to resolve any ambiguities in the ranking from block-level divergences by computing global divergences as needed.

To understand this process, consider the relationship between global divergences and block-level divergences. Let D_p be the *global divergence list* at global position p , where $D_p[\text{id}]$ represents the global divergence between the original haplotype $\text{id} \in \{0, \dots, N-1\}$ and the estimated haplotype at position p . Similarly, let \tilde{D}_i^b be the *block-level divergence list* at block-level position i within block b , where $\tilde{D}_i^b[\text{id}]$ denotes the block-level divergence between the original haplotype $\text{id} \in \{0, \dots, N-1\}$ and the estimated haplotype h^b at position i . The global divergence $D_p[\text{id}]$ can be decomposed into two components:

1. The global divergence $D_{B^b}[\text{id}]$ at the starting position of block b , denoted B^b .
2. The block-level divergence $\tilde{D}_i^b[\text{id}]$ at position i within block b , where $B^b + i = p$.

Given these components, the global divergence $D_p[\text{id}]$ can be computed using the following equation:

$$D_p[\text{id}] = \begin{cases} D_{B^b}[\text{id}], & \text{if } \tilde{D}_i^b[\text{id}] = 0 \\ B^b + \tilde{D}_i^b[\text{id}], & \text{if } \tilde{D}_i^b[\text{id}] > 0 \end{cases} \quad (3)$$

Intuitively, if $\tilde{D}_i^b[\text{id}] = 0$, it indicates that the LCS between the original haplotype id and the estimated haplotype spans from block-level position 0 to $i-1$ and may also extend before block b 's region. Thus, the global divergence at global position p is the same as the global divergence preceding block b 's region, $D_{B^b}[\text{id}]$. On the other hand, if $\tilde{D}_i^b[\text{id}] > 0$, the LCS is contained within block b 's region, so the global divergence at position p is updated by offsetting the block-level divergence $\tilde{D}_i^b[\text{id}]$ with the starting position of the block region, B^b .

Given Equation 3, we first identify scenarios where the ranking of haplotype candidates in the candidate node list C_i^b is *unambiguous*, and therefore does *not* require computing global divergences to determine the S nearest neighbors:

1. *The candidate set contains exactly S candidates.* Since these candidates are already known to have the smallest block-level divergences, and hence the smallest global divergences, no further action is required.
2. *The last node in the candidate node list contains haplotype candidates with a non-zero block-level divergence.* Specifically, let j be the last node in C_i^b . In this case, for all $\text{id} \in \text{Imap}^b(\text{trie}_i^b[j].\text{haps})$, it holds that $\tilde{D}_i^b[\text{id}] > 0$. Since C_i^b is ordered by the smallest block-level divergence between the candidate nodes and the estimated haplotype h^b , the haplotype candidates with the largest block-level divergences—and thus the largest global divergences—are located in the last node j . As all haplotypes $\text{id} \in \text{Imap}^b(\text{trie}_i^b[j].\text{haps})$ have the same block-level divergence $\tilde{D}_i^b[\text{id}] > 0$ due to their common prefix by definition, we can simply eliminate any haplotypes from this node until only S nearest neighbors remain.

This leaves us with the only scenario where the ranking is ambiguous: *when there is only one node on the candidate node list, and that node contains more than S haplotype candidates with zero block-level divergence.* Specifically, let j be the only node in C_i^b . In this case, for all $\text{id} \in \text{Imap}^b(\text{trie}_i^b[j].\text{haps})$, it holds that $\tilde{D}_i^b[\text{id}] = 0$, and $|\text{Imap}^b(\text{trie}_i^b[j].\text{haps})| > S$. Here, the ranking is ambiguous because, according to Equation 3, the global divergence is given by $D_p[\text{id}] = D_{B^b}[\text{id}]$ but $D_{B^b}[\text{id}]$ is unknown. Suppose that $D_{B^b}[\text{id}]$ can be computed (we detail how this is done in the next step), then the haplotype candidates can be ranked by the smallest $D_{B^b}[\text{id}]$ to keep the S nearest neighbors with the smallest global divergence.

The approach of the algorithm in this step is to distinguish between these scenarios and resolve the ambiguity as needed. However, doing this in the oblivious implementation could potentially expose the value of $\tilde{D}_i^b[\text{id}]$ via side channels, and thereby reveal the similarity between the estimated haplotype and the reference haplotypes. Moreover, the choice of which nodes are ranked by global divergences could disclose the path of the estimated haplotype in the prefix trees.

To mitigate these potential issues, our oblivious implementation addresses all scenarios simultaneously by following these two steps: first, in Algorithm 4, haplotypes in *all* nodes are ranked by their global divergence at the starting position of the block, $D_{B^b}[\text{id}]$; then, in Algorithm 5, the resulting ranks along the path of the estimated haplotype are chosen to resolve ranking ambiguities and determine the S nearest neighbors.

In Algorithm 4, our oblivious algorithm efficiently computes the ranks of haplotypes in all nodes using the values of $D_{B^b}[\text{id}]$ via an oblivious merge-sorting algorithm. Intuitively, the algorithm obviously sorts haplotypes in each leaf node independently. Then, at the next level—from the level above the leaves to the root—the top haplotypes are merged according to the prefix tree structure.

Let R_i^b be the *rank list* at block-level position i in block b . The *rank* $R_i^b[j]$ of node $\text{trie}_i^b[j]$ is defined as the list of original haplotypes $\text{id} \in \text{Imap}^b(\text{trie}_i^b[j].\text{haps})$, sorted by $D_{B^b}[\text{id}]$ in ascending order, with $|R_i^b[j]| = \min(S, |\text{Imap}^b(\text{trie}_i^b[j].\text{haps})|)$. The oblivious merge-sorting algorithm proceeds as described in Algorithm 4.

Then, in Algorithm 5, the ranks are combined to select the S nearest neighbors based on the candidate node list. This algorithm combines ranks, irrespective of whether the haplotype order is ambiguous, without affecting correctness, since the internal order within the final set of S nearest neighbors is insignificant.

The oblivious implementation of Algorithm 5 is integrated with the oblivious implementation of Algorithm 3, where the candidate lists C_i^b for all possible values of $\text{inst}_i^b.\text{ord}$ are computed. Then, Algorithm 5 is applied to all the resulting C_i^b , and ORAM is used to select the S nearest neighbors for the correct C_i^b at the end.

Algorithm 4 Compute ranks. Given the prefix tree for block b , trie_i^b for all $i \in \{1, \dots, l^b\}$, and the global divergence list D_{B^b} at the starting position of block b , B^b , compute the rank lists R_i^b for all $i \in \{1, \dots, l^b\}$.

Input: trie_i^b for all $i \in \{1, \dots, l^b\}$, D_{B^b}

Output: R_i^b for all $i \in \{1, \dots, l^b\}$

Create empty lists R_i^b for all $i \in \{1, \dots, l^b\}$

for $i = l^b \rightarrow 1$ **do**

for $j = 0 \rightarrow |\text{trie}_i^b| - 1$ **do**

$G \leftarrow \text{lmap}^b(\text{trie}_i^b[j].\text{haps})$

$s \leftarrow \min(S, |G|)$

if $i = l^b$ **then**

\triangleright Initialization step

$R_i^b[j] \leftarrow \text{top } s \text{ id} \in G, \text{ sorted by the smallest } D_{B^b}[\text{id}]$

else

\triangleright Merging step

$u \leftarrow \text{trie}_i^b[j].0, v \leftarrow \text{trie}_i^b[j].1$

$r_0 \leftarrow R_{i+1}^b[u], r_1 \leftarrow R_{i+1}^b[v]$

$R_i^b[j] \leftarrow \text{top } s \text{ id across } r_0 \text{ and } r_1, \text{ sorted by the smallest } D_{B^b}[\text{id}]$

end if

end for

end for

Algorithm 5 Find the S nearest neighbors. Given the candidate list C_i^b and the rank list R_i^b , find the S nearest neighbor set I_i^b of the estimated haplotype. b is implied and thus omitted for clarity.

Input: C_i, R_i

Output: I_i

Create an empty set I_i

for $j \in C_i$ **do**

$s \leftarrow \min(S - |I_i|, |R_i[j]|)$

$G \leftarrow (R_i[j])[0 : s - 1]$ as a set

$I_i \leftarrow I_i \cup G$

end for

Step 5: Computing global divergence lists (Algorithm 6)

This algorithm computes the global divergence lists D_{B^b} at the start of block b , B^b , for all compressed reference panel blocks. These lists are used by Algorithm 4 to rank haplotypes and narrow down the candidate haplotype set to the S nearest neighbors.

Algorithm 6 leverages the observation from Equation 3 that the global divergence list $D_{B^{b+1}}$ at the start of block $b+1$ can be recursively derived from the global divergence list D_{B^b} at the start of block b and the block-level divergence list $\tilde{D}_{l^b}^b$ at the last position of block b (where $B^b + l^b = B^{b+1}$), starting from the initial list $D_0 = \{0\}^N$. The value of $\tilde{D}_{l^b}^b[\text{id}]$ is calculated by the block-level divergence between the estimated haplotype h^b and node $\text{node} \in \text{trie}_{l^b}^b$, where $\text{id} \in \text{Imap}^b(\text{node.haps})$. This is done by utilizing the PBWT to compute divergences dynamically: by scanning up and down the prefix tree level $\text{trie}_{l^b}^b$ and determining the maximum divergence between the estimated haplotype at $\text{inst}_{l^b}^b.\text{ord}$ and the target node node such that $\text{id} \in \text{Imap}^b(\text{node.haps})$.

Algorithm 6 Compute global divergent lists. Given the global divergence list D_{B^b} , at the starting position of block b , prefix tree leaf level $\text{trie}_{l^b}^b$ at the last position of block b , and the haplotype insertion struct $\text{inst}_{l^b}^b$ at the last position of block b , compute $D_{B^{b+1}}$ at the starting position of block $b + 1$.

Input: $D_{B^b}, \text{trie}_{l^b}^b, \text{inst}_{l^b}^b$

Output: $D_{B^{b+1}}$

▷ Compute the block-level divergence list

$\tilde{D}_{l^b}^b \leftarrow \{0\}^N$

$d \leftarrow \text{inst}_{l^b}^b.\text{div}$

for $j = \text{inst}_{l^b}^b.\text{ord} - 1 \rightarrow 0$ **do** ▷ Scan upward

for $\text{id} \in \text{lmap}^b(\text{trie}_{l^b}^b[j].\text{haps})$ **do** ▷ Note that $|\text{trie}_{l^b}^b[j].\text{haps}| = 1$ by definition

$\tilde{D}_{l^b}^b[\text{id}] \leftarrow d$

end for

$d \leftarrow \max(d, \text{trie}_{l^b}^b[j].\text{div})$

end for

$d \leftarrow \text{inst}_{l^b}^b.\text{div}'$

for $j = \text{inst}_{l^b}^b.\text{ord} \rightarrow |\text{trie}_{l^b}^b| - 1$ **do** ▷ Scan downward

if $j > \text{inst}.\text{ord}$ **then**

$d \leftarrow \max(d, \text{trie}_i[j].\text{div})$

end if

for $\text{id} \in \text{lmap}^b(\text{trie}_{l^b}^b[j].\text{haps})$ **do**

$\tilde{D}_{l^b}^b[\text{id}] \leftarrow d$

end for

end for

▷ Compute the global divergence list

$D_{B^{b+1}} \leftarrow \{0\}^N$

for $j = 0 \rightarrow N - 1$ **do**

if $\tilde{D}_{l^b}^b[j] = 0$ **then** ▷ Compute Equation 3

$D_{B^{b+1}}[j] \leftarrow D_{B^b}[j]$

else

$D_{B^{b+1}}[j] \leftarrow B^b + \tilde{D}_{l^b}^b[j]$

end if

end for

Step 6: Constructing a conditioned reference panel

This is the final step of the oblivious compressed haplotype selection algorithms, where the nearest neighbors are combined to form the conditioned haplotype set, which is then used to build the conditioned reference panel for HMM inference. To construct the conditioned haplotype set, each of the two estimated haplotypes of the target sample undergoes independent S nearest neighbor searches at the same randomly chosen positions. All the S nearest neighbors of both estimated haplotypes at these positions are combined to form a unified conditioned haplotype set. The haplotype IDs in this set are used to look up and copy the corresponding unique haplotypes from the compressed reference panel into the conditioned reference panel. The efficiency gain of this approach is a result of the smaller memory footprint of the compressed reference panel, allowing for faster lookups and copying.

Our oblivious implementation of this step ensures that both the selected haplotypes and the size of the conditioned haplotype set remain hidden. This involves two key operations: first, obviously computing the set union of all the nearest neighbors, and second, obviously filtering the compressed reference panel based on the conditioned haplotype set.

To compute the set union obliviously, we create an oblivious bitmap, where each bit indicates whether the original haplotype $\text{id} \in \{0, \dots, N - 1\}$ is part of the conditioned haplotype set. To incorporate a new S -nearest-neighbor set into the conditioned haplotype set, we iterate through all haplotypes in the S -nearest-neighbor set and use ORAM access (Methods) to look up the haplotypes and flip the corresponding bits in the oblivious bitmap. Once the bitmap is fully updated, each original haplotype ID is mapped to its corresponding unique haplotype ID for each compressed block. Using ORAM access, the unique haplotype IDs are then used to look up and copy the unique haplotypes from the compressed reference panel blocks to build the conditioned reference panel. Given the small size of the array due to bit-packing, we use a linear scanning ORAM, which leads to better concrete efficiency than other more complex ORAM methods.

References

- Delaneau, O, JF Zagury, MR Robinson, JL Marchini, and ET Dermitzakis (2019). “Accurate, scalable and integrative haplotype estimation”. In: *Nature Communications* 10.1, p. 5436.
- Dokmai, N, C Kockan, K Zhu, X Wang, SC Sahinalp, and H Cho (2021). “Privacy-preserving genotype imputation in a trusted execution environment”. In: *Cell Systems* 12.10, 983–993.e7.
- Durbin, R (2014). “Efficient haplotype matching and storage using the positional Burrows-Wheeler transform (PBWT)”. In: *Bioinformatics* 30.9, pp. 1266–1272.
- Loh, PR et al. (2016). “Reference-based phasing using the Haplotype Reference Consortium panel”. In: *Nature Genetics* 48.11, pp. 1443–1448.