AN IMPROVED ERROR BOUND FOR DIFFERENTIALLY PRIVATE FREQUENT-SUBSTRING MINING

Anonymous authors
Paper under double-blind review

ABSTRACT

Given a dataset of n user-contributed strings, each of length at most ℓ, we study the problem of identifying frequent substrings under user-level ε-differential privacy. Bernardini et al. (PODS '25) gave an ε-differentially private algorithm with near-optimal error guarantees, but its $O(n^2\ell^4)$ time and space complexity makes it impractical at scale. More recent work by Guo et al. (ICDT '27) achieves near-linear efficiency, but at the cost of additional polylogarithmic factors in the error. We obtain the best of both works, providing a new ε-differentially private algorithm that improves the near-optimal error guarantees of Bernardini et al. while achieving near-linear time and space.

Our main technical contribution is a differentially private virtual search tree that succinctly represents all sufficiently frequent substrings contained in the dataset. We search this tree with a private top-down search framework for monotone trees, which combines heavy-light tree decomposition with the sparse vector technique. The framework is implemented through constant-time frequency and subtree-size oracles, enabling private path searches and heavy-path decisions without fully materializing the underlying tree. This yields a scalable private frequent-substring mining algorithm with near-optimal privacy-utility tradeoffs and near-linear resource bounds. Specifically, with failure probability at most β, our algorithm achieves additive error $O\left(\frac{\ell \log(n\ell)}{\varepsilon} \log \frac{n\ell}{\beta}\right)$.

1 INTRODUCTION

String-valued data are ubiquitous in modern data analysis, capturing rich sequential structure in user-generated text, search and interaction logs, mobility traces, click streams, and genomic sequences. A fundamental task on such data is frequent-substring mining, which seeks to identify recurring sequences such as phrases in text, route segments in trajectories, genomic motifs, and event patterns in logs. This task underlies applications such as phrase discovery, autocomplete, response suggestion, tokenization, and corpus analysis in natural-language processing (Kim et al., 2021), and also appears in trajectory analysis (Chen et al., 2012b) and genomic-data analysis (Khatri et al., 2019).

Recurring substrings can also pose substantial privacy risks. A rare phrase, route segment, or genomic marker can reveal sensitive information about an individual contributor. Moreover, each user contributes an entire string, whose many substrings may be highly correlated. Thus, protecting individual substrings would not protect the user's contribution as a whole. The natural privacy notion is therefore user-level *differential privacy* (DP), in which the output distribution should be insensitive to changing the entire string contributed by any one user.

In this work, we study user-level differentially private frequent-substring mining. Given n user-contributed strings over an alphabet Σ, each of length at most ℓ, the goal is to identify substrings with large user frequency while satisfying ε-differential privacy.

Differentially private frequent sequential pattern mining has been extensively studied (Chen et al., 2012b; Khatri et al., 2019; Kim et al., 2021; Zhang et al., 2016). For the special case of frequent-substring mining, the closest works are Bernardini, Bille, Gørtz, and Steiner (2025) and Guo, Holland, and Wu (2027). Bernardini et al. (2025) initiated the systematic study of this problem, and gave a data structure (BBGS) with near-optimal additive error. However, their construction requires high polynomial time and space, making it prohibitive at scale. More recently, Guo et al. (2027) (GHW)

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

Reference	Additive error α	Time	Space
BBGS (Bernardini et al., 2025)	$O\Big(\frac{\ell \ln \ell}{\varepsilon}\Big(\ln^2 \frac{n\ell}{\beta} + \ln\lvert\Sigma\rvert\Big)\Big)$	$O\big(n^2\ell^4 + \lvert\Sigma\rvert\big)$	$O\big(n^2\ell^4\big)$
GHW (Guo et al., 2027)	$O\Big(\frac{\ell \ln^2 \bar{\ell}}{\varepsilon} \ln^2 \frac{n\bar{\ell}}{\beta}\Big)$	$O(n\ell \ln\lvert\Sigma\rvert + \lvert\Sigma\rvert)$	$O(n\ell + \lvert\Sigma\rvert)$
This work	$O\Big(\frac{\ell \log(n\ell)}{\varepsilon} \log \frac{n\ell}{\beta}\Big)$	$O(n\ell \log \ell + \lvert\Sigma\rvert)$	$O(n\ell + \lvert\Sigma\rvert)$

Table 1: Comparison of user-level differentially private frequent-substring discovery algorithms. For target threshold τ, additive error α guarantees inclusion at frequency at least $\tau + \alpha$ and exclusion at frequency below $\tau - \alpha$, except with probability β. The comparison assumes that $n, \ell \geq 2$, $\varepsilon \in O(\log(n\ell))$, $|\Sigma| \leq (n\ell)^{O(1)}$, and $\tau \geq 2\alpha$. We write $\bar{\ell} \doteq \ell(\lceil \log|\Sigma| \rceil + 1)$ for the length after binary encoding. Any ε-DP algorithm must incur $\alpha \in \Omega\big(\min\big\{n, \varepsilon^{-1}\ell \ln|\Sigma|\big\}\big)$ (Balcer & Vadhan, 2019; Bernardini et al., 2025).

reduced the time and space to near-linear, but at the cost of additional logarithmic factors in the error. This raises the central question of our work.

Can we improve the additive-error bound of BBGS while achieving near-linear time and space?

We answer this question affirmatively. Our algorithm improves the near-optimal additive error of BBGS while achieving near-linear time and space, up to logarithmic factors comparable to those of GHW. Table 1 summarizes the comparison with the two closest theoretical baselines.

1.1 Our Contributions

We introduce the framework of private heavy-hitter search on monotone trees. In this framework, node labels are nonnegative and monotone along ancestor–descendant paths, and neighboring instances differ only along a single root-to-node path. This is a generalization of the prior "counting on trees" setting (Ghazi et al., 2023; Lebeda et al., 2026), where leaf node counts propagate up the tree. The framework may therefore be useful beyond frequent-substring mining. We give a pure ε-DP top-down search algorithm with additive error $\tilde{O}(\Delta/\varepsilon)$, where Δ is the node frequency sensitivity, using only frequency and subtree-size oracles.

The prior algorithms of BBGS and GHW follow the same high-level template. Both approaches (i) reduce the problem to heavy-hitter search on monotone trees, (ii) identify all frequent substrings at dyadic lengths, named the *dyadic candidate sets*, and (iii) build the monotone search tree(s) from the dyadic candidate sets. We provide improvements to all three ingredients.

First, our solution to heavy-hitter search on monotone trees reduces additive error by a logarithmic factor over both BBGS and GHW. Second, our dyadic candidate set construction improves the additive error of GHW by a logarithmic factor and does not incur the polynomial complexity costs of BBGS. Lastly, to avoid combinatorial blow-ups in runtime, GHW performs heavy-hitter search on monotone trees separately for each dyadic candidate set. This leads to a logarithmic increase in additive error due to privacy composition. We remove the need for composition by showing how a single large virtual search tree, encompassing all string lengths, can be encoded in the dyadic candidate sets in small space. The sum of these parts yields a Pareto improvement over prior work for this problem.

Notably, our search tree construction contributes a differentially private representation of the substring structure of a sensitive string dataset. Beyond frequent-substring discovery, it can serve as a compact form of synthetic sequential data, supporting privacy-preserving training of AI models without exposing raw records. Its accompanying frequency and subtree-size oracles also enable private statistical analysis of arbitrary prefixes and substrings, making the representation useful for tasks such as language-model analysis, motif discovery, anomaly detection, and private feature extraction.

1.2 Technical Overview

A Private Tree-Search Framework. Our starting point is an abstract private search problem on a rooted tree T of size m. Each node $v \in T$ is associated with a frequency value $f(v)$, and these values are monotone along root-to-node paths. That is, if u is an ancestor of v, then $f(u) \geq f(v)$. In the privacy analysis, we consider neighboring instances with the same tree topology but whose frequency

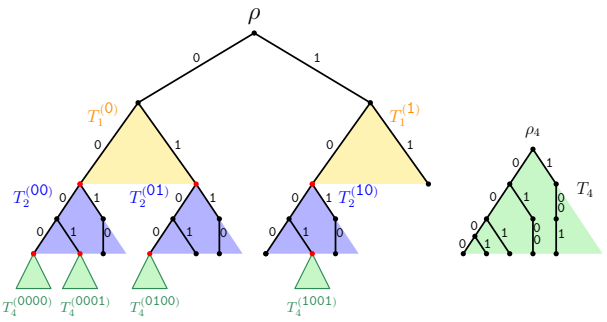

Figure 1: The virtual tree $\mathcal{T}$ for $\mathcal{C}_1 = \{0, 1\}$, $\mathcal{C}_2 = \{00, 01, 10\}$, and $\mathcal{C}_4 = \{0000, 0001, 0100, 1001\}$. Colors indicate dyadic scales; $T_k^{(s)}$ is the copy of the shared suffix trie T_k attached below candidate s, and T_4 is shown on the right. Red nodes are boundary nodes, identified with the roots of next-level copies. Copies are not materialized; the encoding of Appendix F.1 is omitted.

values differ only along a single root-to-node path, and by at most Δ at each node on that path. The goal is to identify all nodes whose frequency is at least a threshold τ. This abstraction is general and may be useful beyond the frequent-substring problem studied in this paper.

The key observation is that, by monotonicity, every root-to-node path has a single threshold crossing. Therefore, nodes above the crossing have frequency at least τ, while nodes below it have frequency below τ. A single invocation of the *sparse vector technique* (Dwork & Roth, 2014) can privately locate this crossing point, up to an additive error, while paying only one unit of privacy loss. The remaining question is which paths should be searched.

This is where *heavy-light decomposition* enters: we decompose T into heavy paths according to subtree sizes. Since the decomposition depends only on the topology of T, it is identical on neighboring instances and incurs no privacy cost. Moreover, every root-to-node path intersects only $O(\ln m)$ heavy paths (Sleator & Tarjan, 1983). Consequently, changing the frequencies along one root-to-node path can affect only $O(\ln m)$ sparse-vector invocations. By allocating the privacy budget across these affected paths and union-bounding over all path searches, we obtain an additive error $\alpha \in O\left(\frac{\Delta \ln m}{\varepsilon} \ln \frac{m}{\beta}\right)$ with failure probability β.

Our method assumes (constant-time) oracle access to both node frequencies and subtree sizes. With these oracles, the search time is proportional to the number of nodes with frequency at least $\tau - \alpha$.

From Substrings to Monotone Tree Search. To apply this framework, we search a trie whose nodes represent strings, each labeled by its number of occurrences in the dataset. The labels are monotone, since every occurrence of a string is also an occurrence of its prefixes, and adding or removing one user's string changes them along at most ℓ root-to-node paths, one per suffix. The full trie of $\Sigma^{1:\ell}$, however, has $|\Sigma|^{\Theta(\ell)}$ nodes, and $\log m = \Theta(\ell \log|\Sigma|)$ would add an ℓ factor to the error. We therefore privately construct a subtree $\mathcal{T}$ of the full trie, of size $(n\ell)^{O(1)}$, that still contains every sufficiently frequent substring.

Similar to BBGS and GHW, the reduction begins by identifying candidate sets $\mathcal{C}_k$ for dyadic lengths $k = 1, 2, 4, \ldots, \ell$ (assuming for simplicity that ℓ is a power of two). Each $\mathcal{C}_k$ contains all sufficiently frequent substrings of length k. Significantly, as noted by Guo et al. (2027), if a substring P of length $t \in (k, 2k]$ is frequent, then its length-k prefix belongs to $\mathcal{C}_k$, and its remaining suffix is a suffix of some string in $\mathcal{C}_k$.

To build the monotone search tree $\mathcal{T}$ (Figure 1), for each dyadic length k, we construct a shared suffix trie T_k from the suffixes of strings in $\mathcal{C}_k$. Then, for every candidate $s \in \mathcal{C}_k$, we attach a virtual copy of T_k below the node in $\mathcal{T}$ that encodes s. A node in this copy represents a string of the form sq, where q is a prefix of a suffix represented in T_k. Thus, the copy attached to s contains all frequent-substring extensions of s at lengths between $k + 1$ and $2k$. Whenever such an extension belongs to $\mathcal{C}_{2k}$, the corresponding boundary node is identified with the root of the next-level copy attached to that candidate. The resulting virtual tree $\mathcal{T}$ contains all sufficiently frequent substrings, while avoiding the materialization of the full substring trie.

Oracle Access to the Virtual Tree. Our solution to heavy-hitter search on monotone trees assumes (constant-time) oracle access to both node frequencies and subtree sizes. Thus, we need to instantiate these oracles on the virtual tree. Both oracles are used in a top-down manner. That is, once a parent

node is reached, the algorithm retrieves the corresponding information for each of its children in constant expected time.

The frequency oracle is relatively standard. We build a suffix-tree data structure over the input strings, augmented with occurrence counts. During the traversal of $\mathcal{T}$, the oracle maintains the suffix-tree locus corresponding to the current node, so the frequency of each child can be obtained by following the appropriate transition and reading the stored count.

The subtree-size oracle is more subtle. One could materialize the virtual tree $\mathcal{T}$ and compute subtree sizes directly, but this would incur the same kind of space blow-up ($\Omega(n^2)$) that the virtual representation is designed to avoid. Instead, we give a bottom-up dynamic program that exploits the shared structure of the subtrees in $\mathcal{T}$. Recall that, between two consecutive dyadic levels $\mathcal{C}_k$ and $\mathcal{C}_{2k}$, all tree extensions attached to nodes in $\mathcal{C}_k$ share the same underlying suffix trie T_k. Thus, the local contribution to a subtree size is identical across copies, and only the downstream contribution from boundary nodes leading to $\mathcal{C}_{2k}$ depends on the particular prefix.

We decompose the subtree size of each virtual node into two parts. The local part is contributed by the shared suffix trie T_k attached at the current candidate $s \in \mathcal{C}_k$, and the forward part is contributed by deeper copies attached at boundary nodes corresponding to candidates in $\mathcal{C}_{2k}$. The local parts are computed once for each shared suffix trie T_k by a standard post-order traversal. The forward parts are computed from longer candidates to shorter candidates, propagating already-computed global subtree sizes upward along the active paths that lead to boundary nodes. Since we store only the active paths that connect candidates in $\mathcal{C}_k$ to candidates in $\mathcal{C}_{2k}$, the total space remains near-linear in the number of dyadic candidates, rather than proportional to the size of the fully materialized virtual tree.

1.3 Comparison with Prior Work

We briefly compare our approach with the two closest theoretical baselines (Bernardini et al., 2025; Guo et al., 2027), focusing on the sources of our improvements. The detailed bounds are summarized in Table 1; a more comprehensive comparison is in Appendix A.

Compared with BBGS (Bernardini et al., 2025), our improvement is both algorithmic and structural. On the utility side, we use the sparse vector technique in place of the binary tree mechanism (Chan et al., 2011; Dwork et al., 2010) used in their heavy-light-based counting framework. For the task of identifying heavy hitters, without releasing their frequencies, this replacement saves a logarithmic factor in the error. On the efficiency side, we avoid their quadratic candidate-generation step. Rather than testing all pairs in $\mathcal{C}_k \times \mathcal{C}_k$ to construct $\mathcal{C}_{2k}$, we enumerate only the concatenations that actually occur in the input and simulate the zero-frequency candidates by a sparse-histogram-style procedure. Moreover, the structural information needed for heavy-light decomposition in their approach is obtained by explicitly constructing the candidate search tree, which contributes to the large time and space bound. In contrast, our virtual tree representation supports the required subtree-size queries without materializing the full tree.

Compared with GHW (Guo et al., 2027), the improvement comes from avoiding repeated private searches over separate dyadic structures. Their algorithm applies heavy-light-based noisy counting separately across dyadic scales, which introduces additional logarithmic factors in the error. In contrast, we construct a single virtual tree containing all candidate heavy hitters and run only one global private top-down search on this tree. This is made possible by two ingredients. The first is a near-linear construction of the dyadic candidate sets. The second is an efficient subtree-size oracle that supports heavy-light decomposition on the virtual tree without materializing it.

Organization. Section 2 formalizes private frequent-substring discovery. Section 3 presents our private top-down search framework for monotone trees, combining the sparse vector technique with heavy-light decomposition. Section 4 reduces frequent-substring discovery to this framework through a private search-tree construction, and discusses this construction and its significance. Appendix A discusses related work, Appendix B recalls the privacy and string-processing tools used in the algorithm, and Appendices C–H contain the proofs and the search-tree construction.

2 PROBLEM SETUP

Notation and Data Model. We use $\mathbb{N}$, $\mathbb{Z}$, and $\mathbb{R}$ to denote the natural numbers, integers, and real numbers, respectively. Let $\mathbb{N}_+ \doteq \{1, 2, \ldots\}$, $\mathbb{R}_+ \doteq (0, \infty)$, and $[n] \doteq \{1, \ldots, n\}$ for $n \in \mathbb{N}_+$.

Let Σ be a finite alphabet. Without loss of generality, every symbol in Σ is encoded by a distinct integer in $\{0, \ldots, |\Sigma| - 1\}$, and we henceforth identify Σ with this integer alphabet. Let Σ^* denote the set of finite strings over Σ. For $S \in \Sigma^*$, let $|S|$ denote its length. For $0 \leq a \leq b$, define $\Sigma^{a:b} \doteq \{S \in \Sigma^* : a \leq |S| \leq b\}$.

Let $\mathcal{D} \doteq (\mathcal{S}_1, \ldots, \mathcal{S}_n)$ be a dataset of n strings in $\Sigma^{1:\ell}$, where $\ell \in \mathbb{N}_+$. For $S \in \Sigma^{1:\ell}$, let $f_{\mathcal{S}_i}(S)$ denote the number of occurrences of S in $\mathcal{S}_i$, and define $f_{\mathcal{D}}(S) \doteq \sum_{i=1}^{n} f_{\mathcal{S}_i}(S)$. We write $f_{\mathcal{D}} \doteq \big(f_{\mathcal{D}}(S)\big)_{P \in \Sigma^{1:\ell}}$ for the substring-frequency vector of $\mathcal{D}$.

We now use this notation to define the utility target, the privacy requirement, and finally the private frequent substring discovery problem studied in this work.

2.1 FREQUENT SUBSTRING DISCOVERY

We first define the discovery objective. Given a target threshold τ, the ideal goal is to output exactly the substrings with frequency at least τ. Under differential privacy, however, substrings whose frequencies are close to τ cannot in general be reliably distinguished. We therefore use the following additive-error formulation. Substrings with frequency at least $\tau + \alpha$ must be included, substrings with frequency below $\tau - \alpha$ must be excluded, and substrings with frequency in the ambiguity region $[\tau - \alpha, \tau + \alpha)$ may be treated arbitrarily.

Definition 2.1 (Frequent and infrequent substrings)**.** *Let $\tau_\top, \tau_\perp \in \mathbb{R}_{\geq 0}$ be two thresholds with $\tau_\top \geq \tau_\perp$. For a dataset $\mathcal{D}$, a string $P \in \Sigma^{1:\ell}$ is called* $\tau_\top$-frequent *if $f_{\mathcal{D}}(P) \geq \tau_\top$, and* $\tau_\perp$-infrequent *if $f_{\mathcal{D}}(P) < \tau_\perp$.*

Definition 2.2 (Substring heavy hitters with additive error)**.** *Let $\tau, \alpha \in \mathbb{R}_{\geq 0}$. For a dataset $\mathcal{D}$, a set $\mathcal{C} \subseteq \Sigma^{1:\ell}$ is called a* (τ, α)-heavy-hitter set *for $\mathcal{D}$ if the following conditions hold:*

1. *every $\tau_\top$-frequent substring belongs to $\mathcal{C}$, where $\tau_\top \doteq \tau + \alpha$; and*
2. *no $\tau_\perp$-infrequent substring belongs to $\mathcal{C}$, where $\tau_\perp \doteq \max\{0, \tau - \alpha\}$.*

Thus, substrings with frequencies in the interval $[\tau_\perp, \tau_\top)$ may be either included or excluded. The parameter α is the additive error of the discovery problem.

2.2 DIFFERENTIAL PRIVACY

In our setting, each string $\mathcal{S}_i$ is contributed by one user. We therefore require user-level privacy, meaning that the output distribution should be stable under changes to the string of any one user. We recall the standard definition of differential privacy.

Definition 2.3 (ε-Differential Privacy (Dwork et al., 2006))**.** *Let $\mathcal{M} : \mathcal{Y} \to \mathcal{Z}$ be a randomized algorithm, and let $\sim$ be a symmetric relation on $\mathcal{Y}$. For $\varepsilon \in \mathbb{R}_{\geq 0}$, we say that $\mathcal{M}$ is* ε-differentially private *(ε-DP) if, for every neighboring pair $y \sim y'$ and every measurable set $E \subseteq \mathcal{Z}$,*

$$\Pr[\mathcal{M}(y) \in E] \leq e^{\varepsilon} \Pr[\mathcal{M}(y') \in E].$$

For frequent substring discovery, the input space is $(\Sigma^{1:\ell})^n$. Two datasets $\mathcal{D} = (\mathcal{S}_1, \ldots, \mathcal{S}_n)$ and $\mathcal{D}' = (\mathcal{S}'_1, \ldots, \mathcal{S}'_n)$ in $(\Sigma^{1:\ell})^n$ are neighboring, denoted $\mathcal{D} \sim \mathcal{D}'$, if they differ in exactly one coordinate. That is, for some $i \in [n]$, we have $\mathcal{S}_j = \mathcal{S}'_j$ for every $j \neq i$, while $\mathcal{S}_i$ and $\mathcal{S}'_i$ may be arbitrary strings in $\Sigma^{1:\ell}$. This is the standard replacement neighboring relation in the DP literature (Balcer & Vadhan, 2019).

2.3 PRIVATE FREQUENT SUBSTRING DISCOVERY

We now combine the utility requirement with the privacy requirement.

Definition 2.4 (Private frequent substring discovery)**.** *Let $\varepsilon \in \mathbb{R}_{\geq 0}$, $\beta \in (0,1)$, and $\tau, \alpha \in \mathbb{R}_{\geq 0}$. A randomized mechanism $\mathcal{M} : (\Sigma^{1:\ell})^n \to 2^{\Sigma^{1:\ell}}$ solves the* $(\varepsilon, \beta, \tau, \alpha)$-private frequent substring discovery problem *if the following conditions hold:*

1. *$\mathcal{M}$ is ε-DP with respect to the user-level neighboring relation above; and*
2. *for every dataset $\mathcal{D} \in (\Sigma^{1:\ell})^n$, with probability at least $1 - \beta$, the output $\mathcal{M}(\mathcal{D})$ is a (τ, α)-heavy-hitter set for $\mathcal{D}$.*

The algorithmic goal is to solve this problem with low time and space complexity while achieving the strongest possible utility guarantee by minimizing the additive error α. It is known that any ε-DP algorithm for this problem must have an additive error $\alpha \in \Omega\big(\min\{n, \varepsilon^{-1}\ell \ln|\Sigma|\}\big)$ (Balcer & Vadhan, 2019; Bernardini et al., 2025). Therefore, a nontrivial additive error is unavoidable.

Output Representation. One must also take care with how the output set $\mathcal{M}(\mathcal{D})$ is represented. Storing each reported string separately incurs memory overhead proportional to the total length of the reported strings. We therefore represent the output as a trie. As the prefixes of a frequent substring are also frequent, this representation is natural in our setting.

3 Private Heavy-Hitter Search on Monotone Trees

Our algorithm builds on a framework for private search in monotone trees. At a high level, we reduce frequent substring discovery to a top-down search problem on a rooted tree. Nodes in the tree represent candidate patterns with associated frequencies that are monotone along ancestor-descendant relations. Consequently, the nodes exceeding any fixed threshold form a prefix-closed region of the tree. This induces the following natural pruning principle. Once a node is certified to lie below the target threshold, its entire descendant subtree may be discarded from further consideration.

The framework separates the problem into two conceptually distinct tasks. The first is the private search of a single monotone path to locate the pruning point. For this, we employ the sparse vector technique (SVT) (Dwork & Roth, 2014), which is well suited to detecting threshold crossings in monotone sequences. The second task determines which paths should be searched without allowing the choice of paths itself to depend on the private frequencies. We resolve this using a heavy-light decomposition (Section 3.2), which provides a canonical, topology-dependent path partition of the tree.

This abstraction isolates the core search mechanism from the substring-specific constructions introduced later. In Section 4, we show how frequent-substring discovery can be reduced to this framework through an appropriate search tree.

3.1 Monotone Top-Down Search

We begin with an abstract search model. Let T be a rooted tree of polynomial size and constant branching factor. Each node $u \in T$ is associated with a private frequency value $f_T(u) \in \mathbb{R}_{\geq 0}$, satisfying the monotonicity condition that

$$f_T(u) \geq f_T(v)$$

whenever u is an ancestor of v. The monotonicity condition extends from nodes to trees.

Definition 3.1 (Monotone trees)**.** *A rooted tree T, with an associated frequency function, is* monotone *if every node $u \in T$ satisfies the monotonicity condition.*

We analyze privacy under the following neighboring relation.

Definition 3.2 (Neighboring monotone trees)**.** *Let T and T' be monotone trees over the same topology. We say that T and T' are* Δ-neighbors *if their frequency labels differ only on nodes along a single root-to-node path and*

$$\max_{u \in V(T)} \big| f_T(u) - f_{T'}(u) \big| \leq \Delta, \quad \Delta \in \mathbb{R}_+.$$

Monotonicity implies that, along every root-to-node path, the nodes above any threshold form a prefix. Consequently, if one could efficiently identify the point where the frequency drops below a target threshold τ, the remaining suffix of the path could be safely pruned from the search.

The SVT provides exactly this capability. Given a monotone sequence of frequencies along a path, SVT privately identifies the first location where the sequence crosses below the threshold τ, up to an additive error. Thus, SVT naturally yields a primitive for private search in monotone paths. The formal guarantee appears in Lemma B.3 in Appendix B.

However, the paths on which the SVT is applied must be chosen with care. A large search tree may contain exponentially many root-to-node paths, and naively running SVT independently on all of them would be computationally infeasible. More importantly, selecting which paths to inspect based on the private data would itself violate privacy. We therefore require a path-selection rule that depends only on the topology of the tree.

3.2 Heavy-Light Decomposition as a Structural Path Selection Strategy

To resolve this issue, we use a heavy-light tree decomposition (Sleator & Tarjan, 1983). In a heavy-light decomposition, every internal node designates one child of maximum subtree size as its heavy child, breaking ties arbitrarily, with the remaining children classified as light. The heavy edges partition the tree into a collection of vertex-disjoint heavy paths.

The crucial structural property is that every root-to-node path intersects only logarithmically many heavy paths (Lemma B.5 in Appendix B.2). Consequently, if the search proceeds at the granularity of heavy paths, rather than individual edges, then every node can be reached through only $O(\log |T|)$ independent (heavy) path searches. This yields a natural division of labor between the two tasks:

- SVT privately searches a single monotone heavy path and determines where it should be pruned;
- heavy-light decomposition specifies, using only subtree sizes, which paths must be explored next.

Importantly, the heavy-light decomposition depends only on the topology of the tree and not on the private frequency values. The resulting path-selection rule is therefore data independent and incurs no privacy cost. The overall algorithm can thus be viewed as a private top-down traversal of the tree at the level of heavy paths.

3.3 Abstract Search Framework

Our abstract framework assumes access to the tree through two local oracles:

- $\mathcal{O}^{\mathrm{freq}}$: a frequency oracle returning $f_T(u)$ for queried nodes;
- $\mathcal{O}^{\mathrm{size}}$: a subtree-size oracle returning the size of the subtree rooted at u.

The subtree-size oracle determines the heavy child of each node during the search, allowing the heavy-light structure to be explored locally without explicitly materializing the full decomposition in advance. This is particularly important in settings where the ambient search tree is only conceptual or virtual (as is the case in our reduction), and the algorithm touches only a small explored portion of it.

Definition 3.3 (Oracle Monotone Trees)**.** *A rooted tree T with node set $V(T)$ is an* oracle monotone tree *if it is monotone and can be accessed through the oracles $\mathcal{O}^{\mathrm{freq}}$ and $\mathcal{O}^{\mathrm{size}}$.*

The full procedure for heavy-hitter search on oracle monotone trees is presented in Algorithm 1. Under this abstraction, the interaction between SVT and heavy-light decomposition yields an efficient private search procedure for recovering all sufficiently frequent nodes. The properties of the algorithm are formalized below.

Theorem 3.4 (Private top-down heavy-hitter search)**.** *Let T be an oracle monotone search tree of size $m \doteq |T|$ and constant branching factor. For every $\varepsilon \in \mathbb{R}_+$, $\beta \in (0,1)$, and $\tau \in \mathbb{R}_{\geq 0}$, there is an ε-DP algorithm, with respect to Δ-neighboring oracle monotone trees, such that, with probability at least $1-\beta$, the following guarantees hold:*

- *The algorithm outputs a (τ, α)-heavy-hitter set in T, where $\alpha = O\Big(\frac{\Delta \cdot \ln m}{\varepsilon} \ln \frac{m}{\beta}\Big)$.*

Algorithm 1: PRIVATETOPDOWNSEARCH

Input: Tree T, oracles $\mathcal{O}^{\mathrm{freq}}, \mathcal{O}^{\mathrm{size}}$, privacy parameter ε, failure probability β, threshold τ
Output: A candidate set $\mathcal{C} \subseteq T$

$m \leftarrow |T|$, $L \leftarrow \lceil \log m \rceil + 1$, $\varepsilon_0 \leftarrow \varepsilon / L$;
$\mathcal{C} \leftarrow \emptyset$, $\mathcal{Q} \leftarrow \{\text{root of } T\}$; `// starting nodes of active heavy paths`
while $\mathcal{Q} \neq \emptyset$ **do**
  Remove a node u from $\mathcal{Q}$;
  Initialize a fresh copy of $\mathsf{SVT}_{\varepsilon_0,\tau}$ with failure probability β/m;
  while u *exists* **do**
    Feed $\mathcal{O}^{\mathrm{freq}}(u)$ to $\mathsf{SVT}_{\varepsilon_0,\tau}$ and let $a \in \{\perp, \top\}$ be the output;
    if $a = \top$ **then**
      break; `// prune the remaining suffix of this heavy path`
    $\mathcal{C} \leftarrow \mathcal{C} \cup \{u\}$;
    if u *is a leaf* **then**
      break;
    Let $v^* \in \arg\max_{v:\, v \text{ child of } u} \mathcal{O}^{\mathrm{size}}(v)$, breaking ties arbitrarily;
    $\mathcal{Q} \leftarrow \mathcal{Q} \cup (\{\text{children of } u\} \setminus \{v^*\})$;
    $u \leftarrow v^*$;
return $\mathcal{C}$;

- *The running time is* $O(|T_{\geq \tau_\perp}|)$, $\tau_\perp \doteq \max\{0, \tau - \alpha\}$, $T_{\geq \tau_\perp} \doteq \{u \in T : f_T(u) \geq \tau_\perp\}$.

The full proof appears in Appendix C.

4 Reduction from Frequent-Substring Discovery to Monotone Tree Search

In this section, we prove the main algorithmic guarantee of the paper.

Theorem 4.1 (Scalable private frequent-substring discovery)**.** *Let* $n, \ell \geq 2$, $\varepsilon \in O(\log(n\ell))$, $\beta \in (0,1)$, *and* $|\Sigma| \leq (n\ell)^{O(1)}$. *There exists an error parameter* $\alpha \in O\Big(\frac{\ell \log(n\ell)}{\varepsilon} \log \frac{n\ell}{\beta}\Big)$ *such that, for every threshold* $\tau \in \mathbb{R}_{\geq 0}$ *satisfying* $\tau \geq 2\alpha$, *there is a randomized mechanism* $\mathcal{M} : (\Sigma^{1:\ell})^n \longrightarrow 2^{\Sigma^{1:\ell}}$ *that solves the* $(\varepsilon, \beta, \tau, \alpha)$*-private frequent substring discovery problem. The mechanism can be implemented in* $O(n\ell \log \ell + |\Sigma|)$ *expected time using* $O(n\ell + |\Sigma|)$ *words.*

The proof reduces differentially private frequent-substring discovery to differentially private heavy-hitter search on monotone trees. We state a search-tree lemma that provides a polynomial-size oracle monotone tree, constructed from the input dataset of strings, prove Theorem 4.1 assuming this search-tree lemma, and discuss its significance.

For the problem at hand, we can fortunately construct such a search tree privately. The following lemma summarizes the properties of this tree.

Lemma 4.2 (Search tree)**.** *Let* $n, \ell \geq 2$, $\varepsilon_{\mathrm{tree}} \in O(\log(n\ell))$, $\beta_{\mathrm{tree}} \in (0,1)$, *and* $|\Sigma| \leq (n\ell)^{O(1)}$. *There is an* $\varepsilon_{\mathrm{tree}}$*-DP construction of a tree representation that, on input* $\mathcal{D}$, *with probability at least* $1 - \beta_{\mathrm{tree}}$, *does not abort and yields a conceptual monotone tree* $\mathcal{T}$ *satisfying the following properties:*

1. *The tree* $\mathcal{T}$ *admits a succinct representation. Given the representation of a conceptual node, all its children can be obtained in* $O(1)$ *expected time.*
2. *The tree* $\mathcal{T}$ *has constant branching factor, depth* $O(\ell \log|\Sigma|)$, *and polynomial conceptual size* $|\mathcal{T}| \leq (n\ell)^{O(1)}$.
3. *The nodes of* $\mathcal{T}$ *carry nonnegative frequency labels that are monotone along root-to-node paths. Whenever a node represents a substring* P, *its label equals the true frequency* $f_{\mathcal{D}}(P)$.
4. *At every depth* d, *the sum of the frequency labels of the nodes of* $\mathcal{T}$ *at depth* d *is at most* $n\ell$.

5. *For $\tau_\top \in O\Big(\frac{\ell \log(n\ell)}{\varepsilon_{\text{tree}}} \log \frac{n\ell}{\beta_{\text{tree}}}\Big)$, every $\tau_\top$-frequent substring in $\mathcal{D}$ is represented by a node of $\mathcal{T}$.*

6. *The representation supports the frequency oracle $\mathcal{O}^{\text{freq}}$ and the subtree-size oracle $\mathcal{O}^{\text{size}}$, each with $O(1)$ expected query time.*

The construction takes $O(n\ell \log \ell + |\Sigma|)$ expected time and uses $O(n\ell + |\Sigma|)$ words. The exact-frequency oracle is internal and is not released by this construction.

Conditions (1)-(3) of Lemma 4.2 imply that $\mathcal{T}$ is a polynomial-size oracle monotone tree, as required by Theorem 3.4. In particular, although $\mathcal{T}$ may be much larger than its explicit representation, the succinct linear-space representation and the two oracles allow the top-down search to run efficiently without materializing $\mathcal{T}$.

Condition 5 guarantees that every substring with frequency at least $\tau_\top$ appears as a node of $\mathcal{T}$. The threshold $\tau_\top$ is smaller than the final additive error α in Theorem 4.1. Thus, any substring whose frequency is at least $\tau + \alpha$ is certainly represented in $\mathcal{T}$, and the private tree-search algorithm can recover it as a heavy node. There is no need for $\mathcal{T}$ to exclude low-frequency substrings; it may contain many light nodes. This is harmless because the abstract search algorithm provides the exclusion guarantee, ensuring that nodes with frequency below $\max\{0, \tau - \alpha\}$ are not output.

The only mismatch with Theorem 3.4 is the neighboring relation. Neighboring monotone trees observe frequency functions that differ along a single root-to-node path. Neighboring string datasets change the frequencies of $O(\ell)$ distinct suffixes, therefore affecting $O(\ell)$ root-to-node paths in their corresponding oracle monotone trees. This mismatch is handled by the standard group-privacy argument (Dwork & Roth, 2014). That is, between two user-level neighboring datasets, we can interpolate through $O(\ell)$ intermediate frequency assignments so that each consecutive pair differs on only one root-to-node path. Equivalently, we run Theorem 3.4 with privacy parameter scaled down by this $O(\ell)$ factor. The full proof of Theorem 4.1 appears in Appendix D.

Lemma 4.2 is our main technical result: a single private search tree for all lengths, of polynomial size, stored in near-linear space, with constant-time frequency and subtree-size oracles. As outlined in Section 1.2 (Figure 1), all candidates of length k share one suffix trie T_k of the suffixes of strings in $\mathcal{C}_k$; a copy of T_k hangs below every candidate, and consecutive lengths are glued at nodes spelling candidates of $\mathcal{C}_{2k}$. Each node is therefore a pair of a candidate and a node of T_k, which gives an $O(n\ell + |\Sigma|)$-word representation (Lemma H.1), and its subtree size splits into a local part, computed once per T_k, and a forward part, nonzero only on paths to glued nodes (Lemma G.4).

Unlike the constructions of BBGS and GHW, Lemma 4.2 achieves three properties at once: (a) a single tree for all lengths, so one private search suffices; (b) candidate sets as accurate as those of BBGS; and (c) near-linear time and space. BBGS achieves (a) and (b) but not (c): it forms $\mathcal{C}_{2k}$ by testing all pairs in $\mathcal{C}_k \times \mathcal{C}_k$ and materializes its search space, which requires $\Omega(n^2\ell^4)$ space in the worst case. GHW achieves (c) but neither (a) nor (b): its candidate sets are less accurate by a logarithmic factor, and it searches each dyadic scale separately. We obtain (b) with (c) through a private sparse histogram over the concatenations that occur in the data, and (a) with (c) through the shared tries and the local and forward parts, which give the subtree sizes of the glued tree without materializing it.

By Theorem 3.4, a single search over $\mathcal{T}$ has error $O((\ell/\varepsilon)\log|\mathcal{T}|\log(|\mathcal{T}|/\beta))$, which is the bound of Theorem 4.1 since $|\mathcal{T}| \le (n\ell)^{O(1)}$. The candidate stage does split its budget over the scales, but its error lacks the $\log|\mathcal{T}|$ factor, so this costs only a factor of $\log \ell \le \log(n\ell)$. The search also explores only a small part of $\mathcal{T}$: by Items 2 and 4, at most $O(n\ell)$ nodes have frequency at least $\tau_\perp \ge \alpha$ (Appendix D), so it takes $O(n\ell)$ time. Finally, the topology of $\mathcal{T}$ and its subtree sizes are post-processing of the private candidates, so, conditioned on them, neighboring datasets share the topology (Definition 3.2), and heavy-light decomposition consumes no privacy budget.

AI USE STATEMENT

We used large language models (LLMs) to aid or polish writing, such as editing and restructuring the text for clarity. The authors reviewed all LLM-assisted text and take full responsibility for the content of this paper.

REFERENCES

Alfred V. Aho and Margaret J. Corasick. Efficient string matching: An aid to bibliographic search. *Commun. ACM*, 18(6):333–340, 1975. doi: 10.1145/360825.360855.

Victor Balcer and Salil P. Vadhan. Differential privacy on finite computers. *J. Priv. Confidentiality*, 9 (2), 2019. doi: 10.29012/JPC.679. URL https://doi.org/10.29012/jpc.679.

Giulia Bernardini, Philip Bille, Inge Li Gørtz, and Teresa Anna Steiner. Differentially private substring and document counting. *Proceedings of the ACM on Management of Data*, 3(2):1–27, 2025.

Luca Bonomi and Li Xiong. A two-phase algorithm for mining sequential patterns with differential privacy. In *Proc. 22nd CIKM*, pp. 269–278, 2013. doi: 10.1145/2505515.2505553. URL https://doi.org/10.1145/2505515.2505553.

T.-H. Hubert Chan, Elaine Shi, and Dawn Song. Private and continual release of statistics. *ACM Trans. Inf. Syst. Secur.*, 14(3):26:1–26:24, 2011.

Rui Chen, Gergely Ács, and Claude Castelluccia. Differentially private sequential data publication via variable-length n-grams. In *Proc. 19th CCS*, pp. 638–649, 2012a. doi: 10.1145/2382196.2382263. URL https://doi.org/10.1145/2382196.2382263.

Rui Chen, Benjamin C. M. Fung, Bipin C. Desai, and Nériah M. Sossou. Differentially private transit data publication: a case study on the montreal transportation system. In *Proc. 18th KDD*, pp. 213–221, 2012b. doi: 10.1145/2339530.2339564. URL https://doi.org/10.1145/2339530.2339564.

Graham Cormode, Cecilia M. Procopiuc, Divesh Srivastava, and Thanh T. L. Tran. Differentially private summaries for sparse data. In Alin Deutsch (ed.), *15th International Conference on Database Theory, ICDT '12, Berlin, Germany, March 26-29, 2012*, pp. 299–311. ACM, 2012. doi: 10.1145/2274576.2274608. URL https://doi.org/10.1145/2274576.2274608.

Cynthia Dwork and Aaron Roth. The algorithmic foundations of differential privacy. *Found. Trends Theor. Comput. Sci.*, 9(3-4):211–407, 2014.

Cynthia Dwork, Frank McSherry, Kobbi Nissim, and Adam D. Smith. Calibrating noise to sensitivity in private data analysis. In Shai Halevi and Tal Rabin (eds.), *Theory of Cryptography, Third Theory of Cryptography Conference, TCC 2006, New York, NY, USA, March 4-7, 2006, Proceedings*, volume 3876 of *Lecture Notes in Computer Science*, pp. 265–284. Springer, 2006.

Cynthia Dwork, Moni Naor, Toniann Pitassi, and Guy N. Rothblum. Differential privacy under continual observation. In Leonard J. Schulman (ed.), *Proceedings of the 42nd ACM Symposium on Theory of Computing, STOC 2010, Cambridge, Massachusetts, USA, 5-8 June 2010*, pp. 715–724. ACM, 2010.

Martin Farach. Optimal suffix tree construction with large alphabets. In *38th Annual Symposium on Foundations of Computer Science, (FOCS)*, pp. 137–143. IEEE Computer Society, 1997. doi: 10.1109/SFCS.1997.646102.

Badih Ghazi, Pritish Kamath, Ravi Kumar, Pasin Manurangsi, and Kewen Wu. On differentially private counting on trees. In *Proc. 50th ICALP*, volume 261, pp. 66:1–66:18, 2023. doi: 10.4230/LIPICS.ICALP.2023.66. URL https://doi.org/10.4230/LIPIcs.ICALP.2023.66.

Peaker Guo, Rayne Holland, and Hao Wu. Fast and optimal differentially private frequent-substring mining. In *Proceedings of the 30th International Conference on Database Theory (ICDT 2027)*, Leibniz International Proceedings in Informatics (LIPIcs), Lille, France, 2027. Schloss Dagstuhl – Leibniz-Zentrum für Informatik. URL https://arxiv.org/abs/2603.09166. To appear.

Haim Kaplan, Yishay Mansour, and Uri Stemmer. The sparse vector technique, revisited. In Mikhail Belkin and Samory Kpotufe (eds.), *Conference on Learning Theory, COLT 2021, 15-19 August 2021, Boulder, Colorado, USA*, Proceedings of Machine Learning Research, pp. 2747–2776. PMLR, 2021. URL http://proceedings.mlr.press/v134/kaplan21a.html.

Tanya Khatri, Gaby G. Dagher, and Yantian Hou. Privacy-preserving genomic data publishing via differentially-private suffix tree. In *Proc. 15th EAI*, pp. 569–584, 2019. doi: 10.1007/978-3-030-37228-6_28. URL https://doi.org/10.1007/978-3-030-37228-6_28.

Kunho Kim, Sivakanth Gopi, Janardhan Kulkarni, and Sergey Yekhanin. Differentially private n-gram extraction. In *Proc. 34th NeurIPS*, pp. 5102–5111, 2021. URL https://proceedings.neurips.cc/paper/2021/hash/28ce9bc954876829eeb56ff46da8e1ab-Abstract.html.

Christian Janos Lebeda, David Erb, Tudor Cebere, and Aurélien Bellet. Lumberjack: Better differentially private random forests through heavy hitter detection in trees. *arXiv preprint arXiv:2605.22756*, 2026.

Miguel Nunez-del Prado, Yoshitomi Maehara-Aliaga, Julián Salas, Hugo Alatrista-Salas, and David Megías. A graph-based differentially private algorithm for mining frequent sequential patterns. *Applied Sciences*, 12(4), 2022. doi: 10.3390/app12042131. URL https://www.mdpi.com/2076-3417/12/4/2131.

Daniel Dominic Sleator and Robert Endre Tarjan. A data structure for dynamic trees. *J. Comput. Syst. Sci.*, 26(3):362–391, 1983. doi: 10.1016/0022-0000(83)90006-5.

Shengzhi Xu, Xiang Cheng, Sen Su, Ke Xiao, and Li Xiong. Differentially private frequent sequence mining. *IEEE Trans. Knowl. Data Eng.*, 28(11):2910–2926, 2016. doi: 10.1109/TKDE.2016.2601106. URL https://doi.org/10.1109/TKDE.2016.2601106.

Jun Zhang, Xiaokui Xiao, and Xing Xie. Privtree: A differentially private algorithm for hierarchical decompositions. In *Proc. SIGMOD Conference 2016*, pp. 155–170, 2016. doi: 10.1145/2882903.2882928. URL https://doi.org/10.1145/2882903.2882928.

APPENDIX

TABLE OF CONTENTS

A Related Work **13**

B Preliminaries **14**

B.1 Differential Privacy Tools 14

B.2 Heavy-Light Decomposition 15

B.3 Strings 16

B.4 Encoded Sparse Suffix Trees 16

C Proof of the Private Tree-Search Theorem **17**

D Proof of the Main Theorem **18**

E Search Tree Foundation **19**

E.1 Candidate Domain 20

E.2 Sparse Private Frequency Estimation 20

E.3 Analysis 21

F Search Tree Structure **22**

F.1 Preprocessing Dyadic Candidates 22

F.2 Shared Component Trees 23

F.3 The Cross-Scale Virtual Tree 23

G Search Tree Oracles **24**

G.1 Frequency Oracle 24

G.2 Subtree-Size Oracle 25

H The Oracle Monotone Search Tree **27**

H.1 Proof of Lemma 4.2 28

A Related Work

Private Substring and Document Counting. The closest line of work studies user-level differentially private counting and mining of contiguous patterns in string databases. Bernardini et al. (2025) initiated the theoretical study of differentially private substring and document counting. Their data structure answers all pattern-counting queries with near-optimal additive error, and also yields near-optimal guarantees for approximate frequent-substring and q-gram mining. However, its main construction requires $O(n^2\ell^4)$ time and space, which is prohibitive for large corpora.

Guo, Holland, and Wu (2027) subsequently reduced the computational cost to near-linear scale. Their algorithm uses a binary encoding of the alphabet, searches over geometrically increasing lengths, reuses suffix-trie structure for candidate generation, and estimates frequencies through heavy-light decomposition together with the binary tree mechanism (Chan et al., 2011; Dwork et al., 2010). This avoids the quadratic pairwise concatenation step used by Bernardini et al. (2025), but incurs additional logarithmic factors in the additive error.

We next explain where our improvements over the two closest baselines (Bernardini et al., 2025; Guo et al., 2027) come from, separating the utility improvement from the efficiency improvement.

On the utility side, the main difference is our private tree-search framework. For the task of identifying heavy hitters, without releasing their frequencies, we use the sparse vector technique (Kaplan et al., 2021) rather than the binary tree mechanism. This saves a logarithmic factor in the error. Moreover, in contrast to Guo et al. (2027), which performs heavy-light-based noisy searches separately across dyadic scales, our algorithm performs a single private top-down search on one virtual tree containing all candidate heavy hitters. This single-tree formulation is enabled by the dyadic candidate construction and by an efficient subtree-size oracle for the virtual tree.

The improvement in time and space over Bernardini et al. (2025) has two sources. First, we construct dyadic candidate sets without materializing all pairs in $\mathcal{C}_k \times \mathcal{C}_k$. Given $\mathcal{C}_k$, the construction by Bernardini et al. (2025) tests all such pairs to form $\mathcal{C}_{2k}$, leading to a quadratic blow-up. We instead enumerate only the nonzero-frequency concatenations using multiple-pattern matching, and account for the zero-frequency candidates through a sparse-histogram-style simulation. This constructs $\mathcal{C}_{2k}$ in near-linear time while retaining the same asymptotic candidate-discovery guarantee.

Second, our virtual-tree representation supports heavy-light decomposition without materializing the full search space. The representation is built from shared suffix tries linking $\mathcal{C}_k$ and $\mathcal{C}_{2k}$, and its subtree sizes can be computed by a bottom-up dynamic program. By contrast, the construction by Bernardini et al. (2025) relies on a stronger pairwise extension view: a frequent substring of length $t \in (k, 2k]$ is generated from its length-k prefix and length-k suffix. This viewpoint naturally leads to pairwise candidate generation and does not expose the shared suffix-trie structure exploited here. As a result, their algorithm materializes a much larger virtual search space, requiring $\Omega(n^2\ell^4)$ space in the worst case.

Private n-gram Extraction. Another closely related line studies differentially private extraction or publication of n-grams. Kim et al. (2021) consider user-level differentially private n-gram extraction from private text corpora. Their algorithm first privately learns short grams and then constructs longer candidates using the fact that a frequent k-gram should have frequent length-$(k-1)$ prefix and suffix witnesses. However, their objective is to extract grams up to a prescribed length under approximate differential privacy, and the privacy budget is spent across the successive gram lengths.

Chen et al. (2012a) proposed an earlier differentially private sequential-data publication method based on variable-length n-grams. Their method explores an n-gram tree from short to long grams, and then synthesizes a private sequence database from the noisy model. These works demonstrate the usefulness of top-down exploration for contiguous sequence data, but they are primarily designed for practical n-gram publication rather than near-optimal all-length frequent-substring recovery with asymptotic guarantees.

Private Tree Methods for Sequential Data. Several works use private prefix, suffix, or hierarchical trees to summarize sequential data. Chen et al. (2012b) construct a differentially private prefix tree for public-transit sequences, where each record is a time-ordered sequence of stations. Because the

representation is prefix-oriented, however, it does not directly capture arbitrary substrings that are frequent away from the beginning of a sequence.

Khatri et al. (2019) study genomic-data publication through a differentially private suffix-tree representation. They build a noisy top-down partitioning tree over nucleotide suffixes, and construct a private suffix tree as post-processing. However the work does not provide a near-optimal asymptotic recovery theorem for all frequent substrings.

Bonomi & Xiong (2013) propose a two-phase algorithm for differentially private sequential-pattern mining. The first phase constructs a model-based private prefix tree to find frequent prefixes and candidate substring patterns, while the second phase refines the candidate counts using a fingerprint-style transformation of the original data. They provide formal privacy analysis and error bounds for prefix and substring queries obtained from the prefix tree; nevertheless, the final top-k procedure depends on candidate-refinement heuristics, rather than giving a near-optimal end-to-end frequent-substring recovery guarantee.

Zhang et al. (2016) introduced PrivTree, a general framework for differentially private hierarchical decompositions. PrivTree shows how to privately release an adaptive tree topology without paying a privacy cost proportional to the final tree height, and it has applications to spatial and sequence data. However, its asymptotic error does not improve over the simple baseline that evenly splits the privacy budget across levels and applies a sparse-histogram mechanism to identify heavy hitters at each level.

Most recently, Lebeda et al. (2026) introduced Lumberjack, a differentially private random-forest algorithm whose main technical component is a private heavy-hitter detector for rooted trees. Their detector uses a recursive noisy-binary-search-style pruning procedure to reduce the dependence of the noise on the tree height, by querying middle layers and propagating heavy and light labels upward and downward. They also use sparse-histogram techniques to avoid materializing empty subtrees. Adapting this technique to differentially private frequent-substring mining is an interesting direction for future work.

Other Sequential-Pattern Models. A separate line of work (Xu et al., 2016; Nunez-del Prado et al., 2022) studies privacy-preserving frequent sequential patterns, where patterns are not required to occur contiguously. This problem is combinatorially different from contiguous substring mining, since a sequential pattern need not appear in consecutive positions.

B Preliminaries

This section reviews the standard privacy primitives and combinatorial tools that we use as building blocks in our algorithms.

B.1 Differential Privacy Tools

We use the following standard tools from differential privacy.

Lemma B.1 (Laplace Mechanism (Dwork et al., 2006)). *Let $\mathcal{Y}$ be a domain equipped with a neighboring relation $\sim$. Let $F : \mathcal{Y} \to \mathbb{R}^d$ have ℓ_1-sensitivity at most Δ, i.e., $\|F(y) - F(y')\|_1 \leq \Delta$, for every $y \sim y'$. For every $\varepsilon \in \mathbb{R}_+$, the following mechanism is ε-DP*

$$\mathcal{M}(y) = F(y) + (Z_1, \ldots, Z_d), \qquad Z_i \overset{\text{iid}}{\sim} \text{Laplace}(\Delta/\varepsilon).$$

Lemma B.2 (Composition (Dwork & Roth, 2014)). *Let $\mathcal{M}_1, \ldots, \mathcal{M}_k$ be a sequence of randomized algorithms, where $\mathcal{M}_1 : \mathcal{Y} \to \mathcal{Z}_1$ and, for every $i = 2, \ldots, k$, $\mathcal{M}_i : \mathcal{Z}_1 \times \cdots \times \mathcal{Z}_{i-1} \times \mathcal{Y} \to \mathcal{Z}_i$. Thus, $\mathcal{M}_i$ may depend on the dataset and on the previous outputs of $\mathcal{M}_1, \ldots, \mathcal{M}_{i-1}$. Suppose that, for every $i \in [k]$ and every fixed transcript $a_1 \in \mathcal{Z}_1, \ldots, a_{i-1} \in \mathcal{Z}_{i-1}$, the randomized mapping $\mathcal{M}_i(a_1, \ldots, a_{i-1}, \cdot)$ is ε_i-DP. Then the joint mechanism $\mathcal{M} : \mathcal{Y} \to \mathcal{Z}_1 \times \cdots \times \mathcal{Z}_k$ obtained by running $\mathcal{M}_1, \ldots, \mathcal{M}_k$ sequentially is ε-DP for $\varepsilon = \sum_{i=1}^k \varepsilon_i$.*

The core privacy primitive that we use when searching for frequent substrings is the sparse vector technique (SVT). This method processes a stream of query answers and privately selects the first index where the queries drop below a threshold. Thus, as $f_{\mathcal{D}}(P) \geq f_{\mathcal{D}}(P')$ for any P that is a prefix of P', it is ideal for privately querying substring frequencies of increasing length. The properties of SVT are summarized as follows.

Lemma B.3 (Sparse Vector Technique (SVT) (Dwork & Roth, 2014)). *Let $m \in \mathbb{N}_+$ and let $\Delta \in \mathbb{R}_+$. Consider the neighboring relation on vectors $\vec{x}, \vec{x}' \in \mathbb{R}^m$ defined by $\|\vec{x} - \vec{x}'\|_\infty \le \Delta$. For every threshold $\tau \in \mathbb{R}$, privacy parameter $\varepsilon \in \mathbb{R}_+$, and failure probability $\beta \in (0,1)$, there exists an ε-DP mechanism $\mathsf{SVT}_{\varepsilon,\tau}$ that scans $\vec{x}$ and outputs $(a_1, \dots, a_t) \in \{\perp, \top\}^t$ for some $t \le m$. The output is either $(\perp, \dots, \perp, \top)$, stopping at the first $\top$, or $(\perp, \dots, \perp)$ with $t = m$. Moreover, let $\eta_{\mathrm{SVT}} \doteq \frac{8\Delta}{\varepsilon} \ln \frac{2m}{\beta}$. Then, for every input vector $\vec{x} \in \mathbb{R}^m$, with probability at least $1-\beta$, the following guarantees hold simultaneously for every $i \in [t]$:*

- *if $a_i = \top$, then $x_i \le \tau + \eta_{\mathrm{SVT}}$; and*
- *if $a_i = \perp$, then $x_i \ge \tau - \eta_{\mathrm{SVT}}$.*

We use the following sparse-histogram primitive, which is implemented by the filter mechanism of Cormode et al. (2012). It can be viewed as an efficient simulation of adding Laplace noise to every coordinate in a large universe and retaining only the coordinates whose noisy values exceed a threshold. Algorithm 2 gives the implementation, and the proof follows below.

Lemma B.4 (Private sparse histogram (Cormode et al., 2012)). *Let $m \in \mathbb{N}_+$ and $\Delta \in \mathbb{R}_+$. Consider the neighboring relation on vectors $\vec{x}, \vec{x}' \in \mathbb{R}^m$ defined by $\|\vec{x} - \vec{x}'\|_1 \le \Delta$. For every $\varepsilon \in \mathbb{R}_+$ and $\beta \in (0,1)$, there is an ε-DP mechanism that, given a sparse representation of $\vec{x}$, outputs a sparse vector $\tilde{x} \in \mathbb{R}^m$ in expected time and space $O(\|\vec{x}\|_0)$. Moreover, with probability at least $1-\beta$, $\|\tilde{x} - \vec{x}\|_\infty \in O\Big(\frac{\Delta}{\varepsilon} \ln \frac{m}{\beta}\Big)$ and $\|\tilde{x}\|_0 \le \|\vec{x}\|_0$.*

Proof of Lemma B.4. Let $S \doteq \{i \in [m] : x_i \neq 0\}$ and $s \doteq |S| = \|\vec{x}\|_0$. Consider the full-universe mechanism that draws independent noises $\eta_i \sim \mathsf{Lap}(\sigma)$ for all $i \in [m]$, where $\sigma \doteq \Theta(\Delta/\varepsilon)$, and outputs coordinate i if and only if $x_i + \eta_i \ge \tau$, where $\tau \doteq \sigma \ln(4m/\beta)$. This mechanism is ε-DP by the Laplace mechanism and post-processing.

We implement the same distribution without enumerating the zero coordinates. For $i \in S$, we sample η_i explicitly and test whether $x_i + \eta_i \ge \tau$. For $i \notin S$, the coordinate would be selected independently with probability $p_\tau \doteq \Pr_{\eta \sim \mathsf{Lap}(\sigma)}[\eta \ge \tau]$. Thus, we sample $Z \sim \mathbb{B}\mathrm{in}(m-s, p_\tau)$, choose a uniformly random size-Z subset of $[m] \setminus S$, and sample the corresponding noises from the conditional distribution of $\mathsf{Lap}(\sigma)$ given $\eta \ge \tau$. This exactly simulates the full-universe mechanism and is therefore ε-DP.

Since $p_\tau \le \beta/(4m)$, the probability that any zero coordinate is output is at most $\beta/4$. Moreover, by a union bound over the nonzero coordinates, with probability at least $1 - \beta/4$, all explicitly sampled noises satisfy $|\eta_i| \le O(\sigma \ln(m/\beta))$. Condition on both events. Then no zero coordinate is output, so $\|\tilde{x}\|_0 \le \|\vec{x}\|_0$.

For every retained nonzero coordinate, the error is exactly $|\eta_i|$. For every non-retained nonzero coordinate, we have $x_i + \eta_i < \tau$, and hence $|x_i| \le \tau + |\eta_i| = O(\sigma \ln(m/\beta))$. Therefore, $\|\tilde{x} - \vec{x}\|_\infty = O\Big(\frac{\Delta}{\varepsilon} \ln \frac{m}{\beta}\Big)$.

Finally, $\mathbb{E}[Z] = (m-s)p_\tau \le O(1)$, so the expected time and space are $O(s+1) = O(\|\vec{x}\|_0)$. □

B.2 Heavy-Light Decomposition

We use the standard heavy-light decomposition of rooted trees. Let T be a rooted tree. For every non-leaf node u, choose a child v whose subtree has maximum size, breaking ties arbitrarily, and declare the edge uv to be *heavy*. All other edges from u to its children are declared *light*. The heavy edges form a collection of edge-disjoint paths, called *heavy paths*.

Lemma B.5 (Heavy-Light Decomposition (Sleator & Tarjan, 1983)). *Let T be a rooted tree with m nodes. A heavy–light decomposition of T can be constructed in $O(m)$ time. Moreover, every root-to-leaf path in T contains at most $O(\ln m)$ light edges.*

The second guarantee follows from a standard halving argument. Whenever a root-to-leaf path traverses a light edge, the size of the current subtree decreases by at least a factor of two.

B.3 STRINGS

Let ϵ denote the empty string. For $S \in \Sigma^*$, let $S[i]$ denote its ith symbol and $S[i \dots j]$ the substring from position i to position j. Such a substring is a *prefix* of S if $i = 1$, and a *suffix* of S if $j = |S|$. For $p \in [|S|]$, let $\mathrm{suffix}(S, p) \doteq S[p \dots |S|]$. For $S_1, S_2 \in \Sigma^*$, let $S_1 \circ S_2$ denote their concatenation. We require the following data structure for storing a sparse set of suffixes from a set of strings.

Definition B.6 (r-spaced sparse suffix tree)**.** *Let $X = \{X_i\}$ be a set of strings on alphabet Σ and let $r \geq 1$. The* r-spaced sparse suffix tree *of X is the trie of the suffixes $\{X_i[j \dots |X_i|] : j \equiv 1 \pmod r\}$.*

Each node u in an r-spaced suffix tree of $X = \{X_i\}$ represents a prefix $\mathtt{str}(u) = \alpha$ of a string in $\{X_i[j \dots |X_i|] : j \equiv 1 \pmod r\}$. In other words, there exists $\beta \in \Sigma^*$ such that $\alpha \circ \beta \in \{X_i[j \dots |X_i|] : j \equiv 1 \pmod r\}$.

We use the following standard consequence of the Aho-Corasick automaton (Aho & Corasick, 1975). For a set of equal-length patterns, one can scan the dataset once and identify all adjacent pairs of matched patterns. We prove the guarantee below using standard properties of Aho–Corasick.

Lemma B.7 (Pattern-concatenation counting)**.** *Let $\mathcal{P} \subseteq \Sigma^k$ be a finite set of equal-length patterns. Given a dataset $\mathcal{D} = (\mathcal{S}_1, \dots, \mathcal{S}_n)$, one can compute*

$$\{(P_1 \circ P_2, f_{\mathcal{D}}(P_1 \circ P_2)) : P_1, P_2 \in \mathcal{P}, f_{\mathcal{D}}(P_1 \circ P_2) > 0\}$$

in time and space $O\big(\sum_{P\in\mathcal{P}}|P| + \sum_{i=1}^{n}|\mathcal{S}_i|\big)$.

Lemma B.8 (Aho–Corasick (Aho & Corasick, 1975))**.** *Let $\mathcal{P} \subseteq \Sigma^k$ be a finite set of equal-length patterns. Given a dataset $\mathcal{D} = (\mathcal{S}_1, \dots, \mathcal{S}_n)$, one can compute $f_{\mathcal{D}}(P)$, together with all occurrence locations, for every pattern $P \in \mathcal{P}$ in time and space $O\big(\sum_{P\in\mathcal{P}}|P| + \sum_{i=1}^{n}|\mathcal{S}_i|\big)$.*

Proof of Lemma B.7. Apply Lemma B.8 to compute all occurrence locations of patterns in $\mathcal{P}$. For each document $\mathcal{S}_i$, create an array A_i indexed by the starting positions of length-k substrings, initialized to $\perp$. Whenever a pattern $P \in \mathcal{P}$ occurs in $\mathcal{S}_i$ starting at position j, set $A_i[j] \leftarrow P$. Since all patterns in $\mathcal{P}$ have length k, at most one pattern can start at any fixed position.

We then scan each A_i. For every position j such that both $A_i[j] \neq \perp$ and $A_i[j+k] \neq \perp$, we increment the counter of the concatenation $A_i[j] \circ A_i[j+k]$. This counts exactly the occurrences of strings $P_1 \circ P_2$ with $P_1, P_2 \in \mathcal{P}$: such a string occurs starting at position j iff P_1 occurs at j and P_2 occurs at $j + k$.

The Aho–Corasick step takes time and space $O\big(\sum_{P\in\mathcal{P}}|P| + \sum_{i=1}^{n}|\mathcal{S}_i|\big)$ by Lemma B.8. The subsequent scan of the arrays takes additional time and space $O(\sum_{i=1}^{n}|\mathcal{S}_i|)$. Thus the total time and space are as claimed. □

B.4 ENCODED SPARSE SUFFIX TREES

Appendices G and H use the following encoded sparse suffix-tree construction.

Lemma B.9 (Encoded sparse suffix tree for a string collection, complete version)**.** *Let $X_1, \dots, X_m$ be nonempty strings over an ordered alphabet Σ. Let $b \doteq \lceil \log|\Sigma| \rceil$ and $r \doteq b + 1$. Let $\mathsf{enc} : \Sigma \longrightarrow \{0,1\}^b$ be an injective order-preserving encoding. Define $\mathsf{Enc}(a) \doteq \mathsf{enc}(a) \circ \$$ for every $a \in \Sigma$, and extend Enc to Σ^+ by concatenation. Assume a word-RAM model in which every symbol in Σ, every integer in $[m]$, and every codeword $\mathsf{Enc}(a)$ fit in $O(1)$ words. The path-compressed trie of*

$$\big\{\mathsf{Enc}\big(X_i[p \dots |X_i|]\big) : i \in [m],\ p \in [|X_i|]\big\}$$

can be constructed in $O\big(\sum_{i=1}^{m}|X_i|\big)$ expected time and stored using $O\big(\sum_{i=1}^{m}|X_i|\big)$ words.

Proof. Let $N \doteq \sum_{i=1}^{m}|X_i|$. We first construct the generalized suffix array of $X_1, \dots, X_m$ and its LCP array, respecting the boundaries of the input strings. From these arrays, we construct in $O(N)$ additional time the compact trie T of

$$\{X_i[p \dots |X_i|] : i \in [m],\ p \in [|X_i|]\}\,.$$

Duplicate suffixes are represented once with their multiplicities, and suffix endpoints are recorded explicitly. The outgoing edges of every explicit node of T are ordered by their first symbols. The generalized suffix array, its LCP array, and T can be constructed in $O(N)$ expected time using $O(N)$ words (Farach, 1997).

We next transform T into the compact trie of the encoded suffixes without explicitly expanding the encoded edge labels. For every explicit node of T, we construct a local compact trie representing the codewords of the first symbols on its outgoing edges. For an original edge labeled aY, the local compact trie represents $\mathsf{Enc}(a)$, and we append the remaining encoded label $\mathsf{Enc}(Y)$ to the corresponding compact edge.

Fix an explicit node u of T, and let $a_1, \dots, a_d$ be the first symbols of its outgoing edges, listed in increasing order. If $d > 0$, we construct the compact trie of $\mathsf{Enc}(a_1), \dots, \mathsf{Enc}(a_d)$. Since enc is order preserving, these codewords are already in lexicographic order. Each codeword fits in $O(1)$ words, so the LCP of two consecutive codewords can be computed in $O(1)$ time. The local compact trie at u can therefore be constructed from the ordered codewords and their consecutive LCP values in $O(d)$ time using $O(d)$ words. If u is a leaf, its local compact trie consists only of its root. Let $\widehat{u}$ denote the root of the local compact trie associated with u. Any suffix-endpoint information and multiplicity stored at u are assigned to $\widehat{u}$.

We now connect the local compact tries according to the edges of T. Consider an edge $e = (u, w)$ of T labeled by aY, where $a \in \Sigma$ is its first symbol and $Y \in \Sigma^*$ is the remainder of the edge label. The codeword $\mathsf{Enc}(a)$ is represented in the local compact trie at u. Let y_a denote the endpoint of the path spelling $\mathsf{Enc}(a)$ in this local compact trie. The preceding construction also associates a local compact trie with the explicit endpoint w; let $\widehat{w}$ denote its root.

Since y_a is a leaf of the local compact trie at u, it has a unique incoming compact edge. If $Y \neq \epsilon$, we append $\mathsf{Enc}(Y)$ to the label of this edge and replace its endpoint y_a by $\widehat{w}$. If $Y = \epsilon$, we replace y_a directly by $\widehat{w}$, thereby splicing the local compact trie at w into the local compact trie at u. In either case, the path replacing e spells $\mathsf{Enc}(a) \circ \mathsf{Enc}(Y) = \mathsf{Enc}(aY)$. The replacement introduces no suppressible nonbranching node, and any suffix-endpoint information and multiplicity stored at w remain associated with $\widehat{w}$.

We store the resulting compact-edge labels implicitly. An edge label within a local compact trie is represented by a codeword and two offsets, while an appended label $\mathsf{Enc}(Y)$ is represented by the source-string interval spelling Y. Thus, every resulting compact-edge label is represented by a constant number of words.

For every suffix represented in T, the root-to-suffix-endpoint string in the transformed tree is its encoding. Moreover, the transformation preserves the longest common prefixes of the encoded suffixes. Indeed, two original suffixes agree on a sequence of symbols if and only if their encodings agree on the corresponding sequence of codewords. At the first distinct symbols, the local compact trie branches at the first position where their codewords differ. The edge-label updates preserve the encoded labels of the original compact edges while maintaining path compression. Hence, the resulting tree is precisely the path-compressed trie of

$$\left\{ \mathsf{Enc}\big(X_i[p \dots |X_i|]\big) : i \in [m],\ p \in [|X_i|] \right\}.$$

Let d_u denote the outdegree of an explicit node u of T. The local compact trie at u has $O(d_u + 1)$ nodes. Since $|V(T)| + \sum_{u \in V(T)} d_u = O(|T|) = O(N)$, all local compact tries can be constructed in $O(N)$ total time using $O(N)$ words. Moreover, T has $O(N)$ edges and suffix endpoints, each of which produces $O(1)$ records in the transformed tree. Together with the construction of the generalized suffix and LCP arrays and their conversion into T, the complete construction takes $O(N)$ expected time and uses $O(N)$ words. □

C Proof of the Private Tree-Search Theorem

Formal Proof of Theorem 3.4. Let $L \doteq \lceil \log m \rceil + 1$. By Lemma B.5, every root-to-node path in the heavy–light decomposition of T intersects at most L heavy paths. Fix such a decomposition and set

$$\varepsilon_0 \doteq \frac{\varepsilon}{L}, \qquad \beta_0 \doteq \frac{\beta}{m}, \qquad \gamma \doteq \frac{8\Delta}{\varepsilon_0} \ln \frac{2m}{\beta_0} \in O\left(\frac{\Delta \cdot \log m}{\varepsilon} \ln \frac{m}{\beta} \right), \qquad \alpha \doteq 2\gamma.$$

Each searched heavy path is processed by one invocation of $\mathsf{SVT}_{\varepsilon_0,\tau}$. Consider any heavy path $Q = (u_1, \ldots, u_r)$ listed in top-down order, and define

$$\vec{f}_Q \doteq (f_T(u_1), \ldots, f_T(u_r)).$$

By monotonicity, $\vec{f}_Q$ is non-increasing. Hence, for any threshold τ, the set of indices i with $\vec{f}_{Q,i} \geq \tau$ is a prefix of $[r]$. Thus, running $\mathsf{SVT}_{\varepsilon_0,\tau}$ on $\vec{f}_Q$ finds the first point on Q whose frequency is likely below τ. Nodes before this point are retained, while the remaining suffix of the heavy path is pruned.

Privacy Guarantee. Let T and T' be Δ-neighboring oracle monotone trees. As their topologies are identical, the heavy-light decomposition is the same for both. Moreover, their frequency attributes differ only on nodes lying on a single root-to-node path, with $\max_{u \in T} |f_T(u) - f_{T'}(u)| \leq \Delta$. This path intersects at most L heavy paths. Therefore, there are at most L *adjacent* SVT invocations that receive input vectors that differ between T and T', and each such vector has ℓ_∞-sensitivity at most Δ. Each adjacent invocation is ε_0-DP by Lemma B.3. The remaining invocations receive identical inputs. For $\varepsilon = L\varepsilon_0$, by adaptive composition, the full procedure is ε-DP.

Utility Guarantee. There are at most m heavy paths of length at most m, and hence at most m invocations of SVT. The output of each SVT invocation is either $(\bot, \ldots, \bot, \top)$, stopping at the first $\top$, or $(\bot, \ldots, \bot)$. Let $\bar{T} \subseteq T$ denote the subtree of nodes that receive a label from $\{\bot, \top\}$. By Lemma B.3 and a union bound, with probability at least $1 - \beta_0/m$, the following event $\mathfrak{E}$ occurs:

- if label of $v \in \bar{T}$ is $\top$, then $f_T(v) \leq \tau + 2\gamma = \tau + \alpha$; or
- if label of $v \in \bar{T}$ is $\bot$, then $f_T(v) \geq \tau - 2\gamma = \tau - \alpha$.

Condition on event $\mathfrak{E}$ and consider any searched heavy path. If SVT stops at a node w, by providing the label $\top$, then $f_T(w) \leq \tau + \alpha$. By monotonicity, every descendant node on that path has frequency at most $\tau + \alpha$. Hence, no node with frequency at least $\tau + \alpha$ is pruned on its heavy path. Since only ancestor nodes of w can have frequency at least $\tau + \alpha$, the top-down traversal reaches and outputs every $(\tau + \alpha)$-frequent node.

Conversely, every output node corresponds to a $\bot$ label, and therefore, by event $\mathfrak{E}$, has frequency at least $\tau - \alpha$. If $\tau \geq \alpha$, the observed frequency is larger than $\tau - \alpha$. If $\tau < \alpha$, the lower threshold is 0, and the claim is trivial since frequencies are nonnegative. Thus, no node with frequency below $\max\{0, \tau - \alpha\}$ is output. Therefore, with probability at least $1 - \beta$, the output is a (τ, α)-heavy-hitter set in T.

Running Time. Condition on event $\mathfrak{E}$. Every retained node has frequency at least $\tau - \alpha \geq \tau_\bot$. Moreover, each searched heavy path contributes its retained prefix and at most one stopping node. Since T has constant branching factor, the number of stopping nodes is within a constant factor of the number of retained nodes. Hence the total number of scanned nodes and oracle calls is $O(|T_{\geq \tau_\bot}|)$. This proves the claimed running-time bound. □

D Proof of the Main Theorem

Proof of Theorem 4.1. We split the privacy budget and failure probability equally between the search-tree construction and the private top-down search.

Search-tree construction. Apply Lemma 4.2 to $\mathcal{D}$ with privacy parameter $\varepsilon/2$ and failure probability $\beta/2$. Let $\mathcal{T}$ be the resulting oracle monotone tree, and let $\alpha_{\mathrm{tree}} \doteq \tau_\top$ denote the coverage threshold guaranteed by Lemma 4.2. Under the parameters $\varepsilon/2$ and $\beta/2$, we have $\alpha_{\mathrm{tree}} \in O\Big(\frac{\ell \log(n\ell)}{\varepsilon} \log \frac{n\ell}{\beta}\Big)$. Except with probability at most $\beta/2$, all guarantees of Lemma 4.2 hold. In particular, $\mathcal{T}$ has constant branching factor and polynomial conceptual size, and represents every substring of frequency at least α_{tree}.

Private tree search and privacy. Condition on any fixed transcript of the first stage, which fixes the representation of $\mathcal{T}$. Run Algorithm 1 on $\mathcal{T}$ with threshold τ, failure probability $\beta/2$, and tree-level privacy parameter $\varepsilon_{\mathrm{search}} \doteq \varepsilon/(4\ell)$.

Replacing one input string removes at most ℓ suffixes and inserts at most ℓ suffixes. Each such suffix changes the frequency labels along one root-to-node path by at most one. Hence, user-level neighboring datasets induce frequency assignments connected by at most 2ℓ single-path changes. By Theorem 3.4 and group privacy, the private search is $2\ell\varepsilon_{\mathrm{search}} = \varepsilon/2$-DP at the user level.

The search-tree construction is $\varepsilon/2$-DP by Lemma 4.2. Adaptive composition (Lemma B.2) therefore implies that the complete mechanism is ε-DP.

Let α_{search} denote the additive error of the private search. By Theorem 3.4, $\alpha_{\mathrm{search}} \in O\Big(\frac{\ell \log|\mathcal{T}|}{\varepsilon} \log \frac{|\mathcal{T}|}{\beta}\Big)$. Since $|\mathcal{T}| \le (n\ell)^{O(1)}$, this simplifies to $\alpha_{\mathrm{search}} \in O\Big(\frac{\ell \log(n\ell)}{\varepsilon} \log \frac{n\ell}{\beta}\Big)$.

Utility. For a sufficiently large constant C, set $\alpha \doteq \alpha_{\mathrm{tree}} + \alpha_{\mathrm{search}} + C\ell \log(n\ell)$. Since $\varepsilon \in O(\log(n\ell))$ and $\beta < 1$, the last term is absorbed by the claimed error bound. Thus, $\alpha \in O\Big(\frac{\ell \log(n\ell)}{\varepsilon} \log \frac{n\ell}{\beta}\Big)$. The final mechanism retains the nodes returned by the private search that represent substrings and reports their represented strings.

Let P be a substring satisfying $f_{\mathcal{D}}(P) \ge \tau + \alpha$. Since $\tau \ge 0$ and $\alpha \ge \alpha_{\mathrm{tree}}$, we have $f_{\mathcal{D}}(P) \ge \alpha_{\mathrm{tree}}$. Lemma 4.2 therefore provides a node $u_P \in \mathcal{T}$ representing P. Its frequency label is the true frequency of P, and hence $f_{\mathcal{T}}(u_P) = f_{\mathcal{D}}(P) \ge \tau + \alpha \ge \tau + \alpha_{\mathrm{search}}$. Theorem 3.4 guarantees that u_P is returned. Thus, every substring of frequency at least $\tau + \alpha$ appears in the final output.

Conversely, every reported substring P is represented by an output node $u_P \in \mathcal{T}$. By Lemma 4.2 and Theorem 3.4, $f_{\mathcal{D}}(P) = f_{\mathcal{T}}(u_P) \ge \max\{0, \tau - \alpha_{\mathrm{search}}\}$. Since $\tau \ge 2\alpha$ and $\alpha \ge \alpha_{\mathrm{search}}$, we have $\tau - \alpha_{\mathrm{search}} \ge \tau - \alpha \ge \alpha \ge 0$. Therefore, no substring of frequency below $\tau - \alpha$ is reported. The final output is a (τ, α)-heavy-hitter set.

Success probability. The search-tree construction fails with probability at most $\beta/2$. Conditioned on any successful first-stage transcript, the private search fails with probability at most $\beta/2$. A union bound therefore gives total failure probability at most β.

Running time and space. By Lemma 4.2, the succinct representation of $\mathcal{T}$ and its oracle data structures can be constructed in $O(n\ell \log \ell + |\Sigma|)$ expected time and stored using $O(n\ell + |\Sigma|)$ words.

By Theorem 3.4, the private search takes $O(|\mathcal{T}_{\ge \tau_\perp}|)$ time, where $\tau_\perp \doteq \max\{0, \tau - \alpha_{\mathrm{search}}\}$. Since $\alpha \ge \alpha_{\mathrm{search}}$ and $\tau \ge 2\alpha$, we have $\tau_\perp = \tau - \alpha_{\mathrm{search}} \ge \tau - \alpha \ge \alpha$.

By Items 2 and 4 of Lemma 4.2, the depth of $\mathcal{T}$ is $O(\ell \log|\Sigma|)$, and the sum of the frequency labels at every depth is at most $n\ell$. Consequently, $\sum_{u \in \mathcal{T}} f_{\mathcal{T}}(u) = O(n\ell^2 \log|\Sigma|)$. Every node in $\mathcal{T}_{\ge \tau_\perp}$ contributes at least $\tau_\perp$ to this sum, and hence $|\mathcal{T}_{\ge \tau_\perp}| = O\Big(\frac{n\ell^2 \log|\Sigma|}{\tau_\perp}\Big)$. Since $|\Sigma| \le (n\ell)^{O(1)}$, we have $\log|\Sigma| = O(\log(n\ell))$. Moreover, the definition of α ensures that $\alpha \ge C\ell \log(n\ell) = \Omega(\ell \log|\Sigma|)$. Together with $\tau_\perp \ge \alpha$, this gives $|\mathcal{T}_{\ge \tau_\perp}| = O(n\ell)$. Thus, the private-search time is subsumed by the search-tree construction time. The complete mechanism runs in $O(n\ell \log \ell + |\Sigma|)$ expected time and uses $O(n\ell + |\Sigma|)$ words. □

E Search Tree Foundation

We now move the exposition to the construction of the oracle monotone tree required by Lemma 4.2. Similar to prior work (Bernardini et al., 2025; Guo et al., 2027), the tree is built from private candidate sets $\mathcal{C}_k$ at dyadic lengths k.

For simplicity, we assume throughout that ℓ is a power of two. Let $\mathcal{H} \doteq \{1, 2, 4, \ldots, \ell\}$. For every $k \in \mathcal{H}$, we privately construct a small candidate set $\mathcal{C}_k \subseteq \Sigma^k$ that contains every $\tau_\top$-frequent length-k substring. The candidates at scale k provide the prefixes and suffixes needed to represent all sufficiently frequent substrings with lengths between $k+1$ and $2k$. The following lemma is the combinatorial basis of the construction.

Lemma E.1 (Dyadic Extension Principle (Guo et al., 2027)). *Let $k, 2k \in \mathcal{H}$, and let $P \in \Sigma^{k+1:2k}$ be $\tau_\top$-frequent. Write $P = X \circ Y$, $|X| = k$. Then $X \in \mathcal{C}_k$, and Y is a suffix of a string in $\mathcal{C}_k$.*

The same family of suffix-prefix extensions can be used after every candidate prefix at scale k. Thus, the role of the candidates is to provide a shared family of possible extensions.

Algorithm 2: PRIVATESPARSEHISTOGRAM (Cormode et al., 2012)

Input: A finite universe Ω, a sparse representation of $\vec{x} \in \mathbb{R}^{\Omega}$, sensitivity bound $\Delta \in \mathbb{R}_+$, privacy parameter $\varepsilon \in \mathbb{R}_+$, and failure probability $\beta \in (0, 1)$

Output: A sparse representation of $\tilde{x} \in \mathbb{R}^{\Omega}$

Set $\sigma \leftarrow \Delta/\varepsilon$, $\tau \leftarrow \sigma \ln(4|\Omega|/\beta)$, and $S \leftarrow \{i \in \Omega : x_i \neq 0\}$;
Sample i.i.d. noises $\eta_i \sim \mathsf{Lap}(\sigma)$, for $i \in S$, and set $R_{\neq 0} \leftarrow \{i \in S : x_i + \eta_i \geq \tau\}$;
Set $p_\tau \leftarrow \Pr_{\eta \sim \mathsf{Lap}(\sigma)}[\eta \geq \tau]$, and sample $Z \sim \mathbb{B}\mathrm{in}(|\Omega| - |S|, p_\tau)$;
Sample a uniformly random size-Z subset $R_{=0} \subseteq \Omega \setminus S$;
Sample i.i.d. noises η_i from $\mathsf{Lap}(\sigma)$ conditioned on $\eta_i \geq \tau$, for $i \in R_{=0}$;
return $\{(i, x_i + \eta_i) : i \in R_{\neq 0}\} \cup \{(i, \eta_i) : i \in R_{=0}\}$;

E.1 CANDIDATE DOMAIN

The sets are constructed inductively over the dyadic lengths. If a length-$2k$ string is frequent, then its length-k prefix and suffix are also frequent. Therefore, the search for dyadic candidates at scale $2k$ can be restricted to the set of substrings $\mathcal{C}_k \circ \mathcal{C}_k$. The main challenge in generating the candidates at successive scales is to avoid explicitly enumerating the quadratic universe of candidate concatenations. For efficiency, we enumerate only the concatenations that occur in the input and use a private sparse histogram to select the candidates for this scale.

We process the scales in $\mathcal{H}$ in increasing order. At scale one, the candidate universe is $\Omega_1 \doteq \Sigma$. At every subsequent scale $2k$, it suffices to consider strings formed by concatenating two candidates from $\mathcal{C}_k$. Indeed, if $\mathcal{C}_k$ contains every $\tau_\top$-frequent length-k string, then both length-k halves of every $\tau_\top$-frequent length-$2k$ string belong to $\mathcal{C}_k$. We therefore use the implicit universe $\Omega_{2k} \doteq \mathcal{C}_k \circ \mathcal{C}_k$, which contains every $\tau_\top$-frequent length-$2k$ string. At each scale, we first compute a private sparse approximation of the frequency vector over Ω_k, and then threshold this approximation to select a bounded candidate set.

E.2 SPARSE PRIVATE FREQUENCY ESTIMATION

For every $k \in \mathcal{H}$, let $f^{(k)} \doteq \left(f_{\mathcal{D}}(P)\right)_{P \in \Omega_k}$ denote the frequency vector over Ω_k. We use PRIVATESPARSEHISTOGRAM (Algorithm 2) to obtain a private sparse approximation $\tilde{f}^{(k)}$ of $f^{(k)}$. The sensitivity, privacy, and failure parameters of each invocation are specified in the analysis below. The algorithm simulates adding independent Laplace noise to every coordinate of Ω_k without enumerating its zero-frequency coordinates. It perturbs the positive-frequency coordinates explicitly and collectively samples the zero-frequency coordinates that pass its internal filtering threshold. It therefore returns a sparse representation of the same filtered noisy histogram that would result from explicitly processing the entire universe.

It remains to compute the sparse representation of $f^{(k)}$ required by PRIVATESPARSEHISTOGRAM. At scale one, we obtain this representation by directly counting the symbols occurring in the dataset. At every subsequent scale $2k$, the *adjacent-pattern-pair algorithm* (Lemma B.7) scans the dataset for adjacent length-k strings $\alpha, \gamma \in \mathcal{C}_k$. For every pair that occurs, it reports the coordinate $P = \alpha \circ \gamma \in \Omega_{2k}$ together with its total number of occurrences. These reported coordinate-value pairs form a sparse representation of $f^{(2k)}$, and every coordinate of Ω_{2k} that is not reported has frequency zero. Thus, the algorithm computes the complete sparse representation of $f^{(2k)}$ without enumerating $\mathcal{C}_k \times \mathcal{C}_k$.

The internal filtering threshold of Algorithm 2 serves only to obtain a sparse representation of the noisy histogram. We use two additional steps to control the size of the candidate set. First, we apply the separate candidate threshold $2\tau_{\mathrm{cand}}$, where τ_{cand} is specified in the analysis below, and define

$$\mathcal{C}_k \doteq \left\{ P \in \Omega_k : \tilde{f}^{(k)}_P \geq 2\tau_{\mathrm{cand}} \right\}.$$

Second, we abort if $|\mathcal{C}_k| > B$, where $B \doteq \varepsilon_{\mathrm{tree}} n / \log^2(n\ell)$. Otherwise, we proceed to the next dyadic scale.

E.3 Analysis

The properties of the dyadic candidate set construction algorithm are summarized as follows.

Lemma E.2 (Dyadic candidate sets)**.** *Let $\varepsilon_{\text{tree}} \in \mathbb{R}_+$ and $\beta_{\text{tree}} \in (0,1)$. Assume that $n, \ell \geq 2$, $\varepsilon_{\text{tree}} \in O(\log(n\ell))$, and $|\Sigma| \leq (n\ell)^{O(1)}$. There is an $\varepsilon_{\text{tree}}$-DP algorithm which, with probability at least $1 - \beta_{\text{tree}}$, does not abort and computes the sets $\mathcal{C}_k \subseteq \Sigma^k$, for all $k \in \mathcal{H}$, such that the following guarantees hold:*

- *for every $k \in \mathcal{H}$, the set $\mathcal{C}_k$ contains every $\tau_\top$-frequent string in Σ^k, where $\tau_\top \in O\Big(\frac{\ell \log(n\ell)}{\varepsilon_{\text{tree}}} \log \frac{(n\ell)}{\beta_{\text{tree}}}\Big)$;*
- *for every $k \in \mathcal{H}$, we have $|\mathcal{C}_k| \leq B \doteq \frac{\varepsilon_{\text{tree}} n}{\log^2(n\ell)}$;*
- *for every $k, 2k \in \mathcal{H}$, we have $\mathcal{C}_{2k} \subseteq \mathcal{C}_k \circ \mathcal{C}_k$;*
- *the total expected running time is $O(|\Sigma| + n\ell \log \ell)$, and the space usage is $O(|\Sigma| + n\ell)$.*

Proof of Lemma E.2. We first establish privacy, then prove simultaneous accuracy and candidate coverage, and finally bound the candidate-set sizes and the total resource usage.

Privacy. Fix $k \in \mathcal{H}$, and condition on the transcript of all preceding scales, which fixes Ω_k. Replacing one input string removes and inserts at most ℓ length-k occurrences each, so $\|f^{(k)}(\mathcal{D}) - f^{(k)}(\mathcal{D}')\|_1 \leq 2\ell$. Accordingly, at every scale we invoke PrivateSparseHistogram with sensitivity $\Delta = 2\ell$ and privacy parameter $\varepsilon_{\text{tree}}/|\mathcal{H}|$. Each invocation is $\varepsilon_{\text{tree}}/|\mathcal{H}|$-DP by Lemma B.4. Adaptive composition over the $|\mathcal{H}|$ invocations (Lemma B.2) gives $\varepsilon_{\text{tree}}$-DP. The candidate threshold, subsequent implicit universes, and abort rule are post-processing.

Simultaneous Accuracy and Coverage. At every scale, we invoke PrivateSparseHistogram with failure probability $\beta_{\text{tree}}/|\mathcal{H}|$. On every non-aborting transcript, $|\mathcal{C}_k| \leq B$, and hence $|\Omega_{2k}| \leq B^2$. Moreover, $|\Omega_1| = |\Sigma|$.

These universe-size bounds determine the accuracy required from the sparse-histogram invocations. We choose τ_{cand} to simultaneously bound their ℓ_∞-error, dominate their internal filtering thresholds, and ensure that the retained candidate sets have size at most B. Specifically, for a sufficiently large constant C, let

$$\tau_{\text{cand}} \doteq C \max\left\{ \frac{\ell|\mathcal{H}|}{\varepsilon_{\text{tree}}} \log \frac{|\mathcal{H}| \max\{|\Sigma|, B^2, 2\}}{\beta_{\text{tree}}}, \frac{\ell \log^2(n\ell)}{\varepsilon_{\text{tree}}} \right\}.$$

Since $|\mathcal{H}| = O(\log \ell)$ and $|\Sigma|, B^2 \leq (n\ell)^{O(1)}$, this choice satisfies

$$\tau_{\text{cand}} = O\left(\frac{\ell \log(n\ell)}{\varepsilon_{\text{tree}}} \log \frac{n\ell}{\beta_{\text{tree}}} \right).$$

The first term controls the error and the internal filtering threshold of every sparse-histogram invocation. The second term provides the lower bound on τ_{cand} needed in the candidate-size analysis. In particular, the constant C can be chosen so that every internal filtering threshold is at most $2\tau_{\text{cand}}$. Finally, set $\tau_\top \doteq 3\tau_{\text{cand}}$.

By Lemma B.4, the invocation at each scale has ℓ_∞-error at most τ_{cand}, except with probability $\beta_{\text{tree}}/|\mathcal{H}|$. A union bound over the dyadic scales shows that, with probability at least $1 - \beta_{\text{tree}}$,

$$\|\tilde{f}^{(k)} - f^{(k)}\|_\infty \leq \tau_{\text{cand}} \qquad \text{for every } k \in \mathcal{H}.$$

Condition on this simultaneous accuracy event. We prove by induction over the dyadic scales that $\mathcal{C}_k$ contains every $\tau_\top$-frequent string in Σ^k. For $k = 1$, every such string lies in $\Omega_1 = \Sigma$, and its released frequency is at least $\tau_\top - \tau_{\text{cand}} = 2\tau_{\text{cand}}$. It therefore belongs to $\mathcal{C}_1$.

Now fix $k, 2k \in \mathcal{H}$, and assume the claim at scale k. Let $P \in \Sigma^{2k}$ have frequency at least $\tau_\top$. Both length-k halves of P have frequency at least $\tau_\top$, and hence belong to $\mathcal{C}_k$ by the induction hypothesis. Therefore, $P \in \Omega_{2k} = \mathcal{C}_k \circ \mathcal{C}_k$. The simultaneous accuracy guarantee gives $\tilde{f}_P^{(2k)} \geq \tau_\top - \tau_{\text{cand}} = 2\tau_{\text{cand}}$, so $P \in \mathcal{C}_{2k}$. Thus, every $\tau_\top$-frequent string belongs to the corresponding candidate set. Moreover, the construction directly ensures that $\mathcal{C}_{2k} \subseteq \mathcal{C}_k \circ \mathcal{C}_k$.

Candidate Size and Non-Abortion. Continue to condition on the simultaneous accuracy event. If $P \in \mathcal{C}_k$, then $\tilde{f}_P^{(k)} \geq 2\tau_{\mathrm{cand}}$, and hence $f_{\mathcal{D}}(P) \geq \tau_{\mathrm{cand}}$. The total frequency mass at any fixed length k is at most $n\ell$. Consequently,

$$|\mathcal{C}_k| \leq \frac{n\ell}{\tau_{\mathrm{cand}}} \leq B,$$

where the last inequality follows from the second term in the definition of τ_{cand} and the definition of B. Thus, the construction does not abort on the simultaneous accuracy event.

Resources. At scale $k > 1$, adjacent-pattern-pair counting takes $O\big((k/2)|\mathcal{C}_{k/2}| + n\ell\big)$ time. The sparse representation of $f^{(k)}$ has at most $n\ell$ nonzero coordinates, so the private sparse-histogram invocation takes $O(n\ell)$ expected time. Summing over the dyadic scales, the total expected running time is

$$O\left(|\Sigma| + n\ell|\mathcal{H}| + \sum_{k\in\mathcal{H}} k|\mathcal{C}_k|\right).$$

Since $|\mathcal{H}| = O(\log \ell)$, $|\mathcal{C}_k| \leq B = O(n)$, and $\sum_{k\in\mathcal{H}} k = O(\ell)$, this simplifies to

$$O(|\Sigma| + n\ell \log \ell).$$

The temporary workspaces used for adjacent-pattern-pair counting and the private sparse histogram are reused across scales. The peak space consists of the packed input, the alphabet dictionary, the retained candidate sets, and the temporary workspace for one scale. It is therefore

$$O\left(|\Sigma| + n\ell + \sum_{k\in\mathcal{H}} k|\mathcal{C}_k|\right) = O(|\Sigma| + n\ell + B\ell).$$

Since $\varepsilon_{\mathrm{tree}} \in O(\log(n\ell))$, we have $B = O(n)$. Thus, the space usage is

$$O(n\ell + |\Sigma|).$$

This proves Lemma E.2. □

F Search Tree Structure

We next construct a shared extension structure for the candidates at each dyadic scale. For the remainder of the exposition, we condition on the event that the algorithm in Lemma E.2 does not abort and all its guarantees hold simultaneously.

F.1 Preprocessing Dyadic Candidates

A direct trie over Σ may have degree $|\Sigma|$, whereas Theorem 3.4 requires constant node degree. We therefore encode the candidate strings over a constant-size alphabet prior to constructing the component trees.

Put $b \doteq \lceil \log|\Sigma| \rceil$ and $r \doteq b + 1$. Use the order-preserving binary encoding $\mathsf{enc} : \Sigma \longrightarrow \{0,1\}^b$ of the integer alphabet. Let $\$ \notin \{0,1\}$ be a delimiter symbol, and define $\mathsf{Enc}(a) \doteq \mathsf{enc}(a) \circ \$$ for every $a \in \Sigma$. Thus, each symbol of Σ is represented by a length-r codeword over $\{0,1,\$\}$, with $\$$ marking the end of the codeword. We extend Enc to strings by concatenating their codewords. Specifically, for $P = a_1 a_2 \cdots a_t \in \Sigma^*$, define

$$\mathsf{Enc}(P) \doteq \mathsf{Enc}(a_1) \circ \mathsf{Enc}(a_2) \circ \cdots \circ \mathsf{Enc}(a_t).$$

For every $k \in \mathcal{H}$, let $\mathsf{Enc}(\mathcal{C}_k) \doteq \{\mathsf{Enc}(P) : P \in \mathcal{C}_k\}$.

The terminal at the end of each encoded character is necessary to prevent the extraction of frequent substrings that are not character aligned. For example, the substring '`AC`', for `A`, `C` $\in \Sigma$, may occur frequently in $\mathcal{D}$. However, if the symbols are encoded as $\mathsf{enc}(\texttt{A}) = \texttt{0000}$ and $\mathsf{enc}(\texttt{C}) = \texttt{0101}$, we may incorrectly output '`0001`' if the boundary between `A` and `C` is not recognized.

Further, when operating on the binary alphabet, we need to ensure that output substrings are well-defined. That is, we want to avoid outputting frequent substrings that cannot be correctly decoded. Continuing the example above, the substring '00$0101$' can be frequent in the encoded dataset, but it does not decode to a substring in Σ^*. To enable correct outputs, we only generate *character-aligned* substrings, which are defined as follows.

Definition F.1 (Character-aligned Strings)**.** *A length-k string $P \in \{0, 1, \$\}^k$ is* character-aligned *if there exist integers $a \geq 0$ and $0 \leq c \leq r$ such that $k = ar + c$ and $P[1 \dots ar]$ can be written as $S_1 \cdots S_a$ where each $S_i \in \mathsf{Enc}(\Sigma)$.*

All suffixes used in the component trees begin at codeword boundaries. Traversal may stop inside a codeword, so intermediate nodes need not represent complete strings over Σ; codeword boundaries identify the nodes that can be decoded.

F.2 Shared Component Trees

Let $\mathsf{Enc}(\mathcal{C}_k)$ denote the encoded candidates at scale k. We next construct a shared extension structure for the encoded candidates at each dyadic scale. Motivated by Lemma E.1, for every $k \in \mathcal{H}$, we build a trie T_k from the character-aligned suffixes of all candidates in $\mathsf{Enc}(\mathcal{C}_k)$. We call this trie the scale-k *component tree*. For a fixed candidate $X \in \mathsf{Enc}(\mathcal{C}_k)$, prefixing the root of the component tree by X represents the strings obtained by extending X with candidate suffixes. Lemma E.1 guarantees that this prefixed tree contains every sufficiently frequent substring in $\Sigma^{k+1:2k}$ that begins with X.

As character encodings have identical length, T_k is built from an r-spaced sparse suffix tree (Definition B.6). Each suffix inserted into T_k is character-aligned and begins at a codeword boundary. Therefore, any traversal from the root constitutes a well-defined substring extension. For each node v of T_k, let $\mathtt{str}(v)$ denote the string spelled by the root-to-v path. Thus, every node of T_k represents a prefix of an encoded suffix of some candidate in $\mathsf{Enc}(\mathcal{C}_k)$.

For every $k \in \mathcal{H}$ and $\alpha \in \mathsf{Enc}(\mathcal{C}_k)$, we conceptually create a prefixed instance of T_k, denoted by $T_k^{(\alpha)}$. Its nodes are represented by pairs (α, v), where v is a node of T_k. The node represented by (α, v) spells

$$\mathtt{str}((\alpha, v)) \doteq \alpha \circ \mathtt{str}(v).$$

Thus, $T_k^{(\alpha)}$ represents all strings of the form $\alpha \circ q$, where q is a prefix of an encoded suffix of some candidate in $\mathsf{Enc}(\mathcal{C}_k)$. The instances $T_k^{(\alpha)}$ are not materialized. All instances at scale k share the same underlying component tree T_k and differ only in their prefixes.

F.3 The Cross-Scale Virtual Tree

We now assemble the component-tree instances into a single conceptual tree $\mathcal{T}$ spanning all dyadic scales. Algorithm 3 formalizes the construction, and Figure 1 illustrates it. The algorithm initializes $\mathcal{T}$ as the trie of $\mathsf{Enc}(\mathcal{C}_1)$ and connects consecutive dyadic scales from $\mathsf{Enc}(\mathcal{C}_k)$ through the candidates in $\mathsf{Enc}(\mathcal{C}_{2k})$.

To describe a cross-scale connection, fix $k \in \mathcal{H}$ such that $2k \in \mathcal{H}$ and $\beta \in \mathsf{Enc}(\mathcal{C}_{2k})$. Since $\mathsf{Enc}(\mathcal{C}_{2k}) \subseteq \mathsf{Enc}(\mathcal{C}_k) \circ \mathsf{Enc}(\mathcal{C}_k)$, write $\beta = \alpha \circ \gamma$ for $\alpha, \gamma \in \mathsf{Enc}(\mathcal{C}_k)$. Let v_γ be the unique node of T_k satisfying $\mathtt{str}(v_\gamma) = \gamma$. Then $\mathtt{str}((\alpha, v_\gamma)) = \alpha \circ \gamma = \beta$, so (α, v_γ) is the unique node of $T_k^{(\alpha)}$ that spells β. Since the dyadic scales are processed in increasing order, this node is already present in the partially constructed tree. The algorithm attaches $T_{2k}^{(\beta)}$ at this node by merging its root (β, ρ_{2k}) into (α, v_γ), where ρ_{2k} is the root of T_{2k}. Repeating this operation over all consecutive dyadic scales defines $\mathcal{T}$.

For subsequent navigation and oracle queries, we describe nodes of $\mathcal{T}$ by their representations within the component-tree instances. Every node of $\mathcal{T}$ admits a representation (α, v). For a node in the initial trie, we take $\alpha = \epsilon$ and let v be the corresponding initial-trie node. Otherwise, $\alpha \in \mathsf{Enc}(\mathcal{C}_k)$ and v is a node of T_k for some $k \in \mathcal{H}$. *A node created by a cross-scale merge may admit multiple representations*. In particular, the merged node above is represented both as (α, v_γ) at scale k and as (β, ρ_{2k}) at scale $2k$. These representations refer to the same conceptual node of $\mathcal{T}$.

Algorithm 3: ASSEMBLEVIRTUALTREE

Input: Encoded candidate sets $\{\mathsf{Enc}(\mathcal{C}_k) : k \in \mathcal{H}\}$ and component trees $\{T_k : k \in \mathcal{H}\}$
Output: The conceptual cross-scale virtual tree $\mathcal{T}$

Initialize $\mathcal{T}$ as the uncompressed trie of $\mathsf{Enc}(\mathcal{C}_1)$, and denote its root by ρ;
foreach $\alpha \in \mathsf{Enc}(\mathcal{C}_1)$ **do**
  Attach $T_1^{(\alpha)}$ at the node labeled by α in the initial trie by merging the root of $T_1^{(\alpha)}$ into that node;
foreach $k \in \mathcal{H}$ *such that* $2k \in \mathcal{H}$*, in increasing order* **do**
  Let ρ_{2k} denote the root of T_{2k};
  foreach $\beta \in \mathsf{Enc}(\mathcal{C}_{2k})$ **do**
    Decompose $\beta = \alpha \circ \gamma$, where $\alpha, \gamma \in \mathsf{Enc}(\mathcal{C}_k)$;
    Locate the unique node v_γ of T_k satisfying $\mathtt{str}(v_\gamma) = \gamma$;
    Attach $T_{2k}^{(\beta)}$ at the node (α, v_γ) in $\mathcal{T}$ by merging the root (β, ρ_{2k}) of $T_{2k}^{(\beta)}$ into that node;
return $\mathcal{T}$;

The resulting tree $\mathcal{T}$, its component-tree instances, and their cross-scale connections are conceptual and are not materialized.

G SEARCH TREE ORACLES

Algorithm 1 accesses $\mathcal{T}$ through the frequency oracle $\mathcal{O}^{\mathrm{freq}}$ and the subtree-size oracle $\mathcal{O}^{\mathrm{size}}$. Given a representation of a conceptual node, the former returns the frequency of its represented substring, while the latter returns the number of conceptual nodes in its subtree. We implement both oracles without materializing $\mathcal{T}$.

G.1 FREQUENCY ORACLE

Since $\mathcal{T}$ is defined over the encoded alphabet, we must compute the frequency of all character-aligned substrings in $\mathsf{Enc}(\mathcal{S}_1), \ldots, \mathsf{Enc}(\mathcal{S}_n)$. A natural solution is to build a counter-based suffix tree over $\mathsf{Enc}(\mathcal{S}_1), \ldots, \mathsf{Enc}(\mathcal{S}_n)$. However, as, under the encoding, every original symbol is encoded by r symbols, the encoded strings have total length $O(n\ell r)$, and the resulting data structure would require $O(n\ell r)$ construction time and memory.

Recall that $\mathrm{suffix}(\mathcal{S}_i, p)$ denotes the suffix of $\mathcal{S}_i$ starting at position p. For every $z \in \{0, 1, \$\}^*$, define

$$f_{\mathcal{D}}^{\mathrm{enc}}(z) \doteq |\{(i,p) : i \in [n],\ p \in [|\mathcal{S}_i|],\ z \text{ is a prefix of } \mathsf{Enc}(\mathrm{suffix}(\mathcal{S}_i, p))\}|.$$

Equivalently, $f_{\mathcal{D}}^{\mathrm{enc}}(z)$ is the number of occurrences of z in the encoded input strings that begin at codeword boundaries. Thus, if z' is not character-aligned, $f_{\mathcal{D}}^{\mathrm{enc}}(z') = 0$. This definition applies whether or not z ends at a codeword boundary. Accordingly, for every node $x \in \mathcal{T}$, define

$$f_{\mathcal{T}}(x) \doteq f_{\mathcal{D}}^{\mathrm{enc}}(\mathtt{str}(x)).$$

The value $f_{\mathcal{T}}(x)$ is well defined because all representations of x spell the same encoded string. In particular, the two representations of a cross-scale attachment node receive the same frequency.

The oracle can be implemented with an r-spaced sparse suffix tree augmented with frequency counters at branching nodes. On input $\mathsf{Enc}(\mathcal{D})$, each substring starting at a codeword boundary is inserted. Thus, each branching node u in the tree counts the number of character-aligned suffixes that share the prefix $\mathtt{str}(u)$.

Lemma G.1 (Frequency oracle)**.** *Let $\mathcal{G}$ be the augmented r-spaced sparse suffix tree built from $\mathsf{Enc}(\mathcal{D})$, where each branching node z is assigned the counter*

$$|\{(i,p) : i \in [n],\ p \in [|\mathcal{S}_i|],\ z \text{ is a prefix of } \mathsf{Enc}(\mathrm{suffix}(\mathcal{S}_i, p))\}|.$$

Then $\mathcal{G}$ implements a frequency oracle $\mathcal{O}^{\mathrm{freq}}$ satisfying $\mathcal{O}^{\mathrm{freq}}(x) = f_{\mathcal{T}}(x)$ for every node $x \in \mathcal{T}$. If x represents a substring P, then $\mathcal{O}^{\mathrm{freq}}(x) = f_{\mathcal{D}}(P)$. Moreover, the frequency labels are monotone

along root-to-node paths, and their sum at every fixed depth of $\mathcal{T}$ is at most $n\ell$. For any node $x \in \mathcal{T}$, the frequencies of all its children can be obtained in $O(1)$ time. $\mathcal{G}$ can be constructed in $O(n\ell)$ expected time and stored using $O(n\ell)$ words.

The labels are monotone along root-to-node paths. Indeed, if x is an ancestor of y, then every encoded suffix having $\mathtt{str}(y)$ as a prefix also has $\mathtt{str}(x)$ as a prefix. They also agree with the original substring frequencies: for every $P \in \Sigma^*$,

$$f_{\mathcal{D}}^{\mathrm{enc}}(\mathsf{Enc}(P)) = f_{\mathcal{D}}(P). \tag{1}$$

This follows because occurrences of P are in one-to-one correspondence with encoded input suffixes having $\mathsf{Enc}(P)$ as a prefix.

Finally, adding or removing one input suffix changes $f_{\mathcal{T}}$ by at most one along the root-to-node path spelling its encoding and leaves all other labels unchanged. Thus, for a single input suffix, the labels satisfy the neighboring relation of Theorem 3.4.

Proof of Lemma G.1. Let $N = \sum_{i=1}^{n} |\mathcal{S}_i| \le n\ell$. Apply Lemma B.9 to the original input strings $\mathcal{S}_1, \ldots, \mathcal{S}_n$. This constructs the compact trie of their encoded, character-aligned suffixes in $O(N)$ expected time and $O(N)$ words, without expanding all Nr encoded positions. The word assumptions hold under the stated integer-alphabet model and $|\Sigma| \le (n\ell)^{O(1)}$: each original symbol, input index, and encoded codeword occupies $O(1)$ words.

Retain all suffix endpoints explicitly, including endpoints that are not branching nodes, and record their multiplicities. There are at most N such endpoints. A postorder traversal assigns each explicit node the total multiplicity of suffix endpoints in its subtree, including any suffix ending at that node. This adds $O(N)$ time and words.

For any string z having a locus in this trie, the encoded suffixes with prefix z are exactly the suffix endpoints below that locus. Their total multiplicity is stored at the locus when it is explicit, and at the lower endpoint of its compact edge otherwise. A string with no locus has frequency zero. Therefore the returned count is $f_{\mathcal{D}}^{\mathrm{enc}}(z)$, including when z ends inside a codeword. In particular, $\mathcal{O}^{\mathrm{freq}}(x) = f_{\mathcal{T}}(x)$, and Equation (1) gives the original substring frequency whenever $\mathtt{str}(x) = \mathsf{Enc}(P)$. Extending a prefix cannot increase the number of matching encoded suffixes, which proves monotonicity.

Nodes at a fixed depth of $\mathcal{T}$ spell distinct strings of the same length. Each encoded input suffix therefore contributes to at most one such node. There are $N \le n\ell$ input suffixes counted with multiplicity, proving the bound on frequency mass.

During a top-down traversal of $\mathcal{T}$, maintain the current locus, represented by an explicit node or an edge and an offset. Moving to a child requires one transition over the constant-size encoded alphabet. The next edge-label symbol is obtained in constant time from the source interval and its codeword offset; codewords occupy $O(1)$ words. The child's count is then read from the appropriate explicit endpoint. A missing transition is represented by a zero-frequency state, which persists for all descendants. Thus the loci and frequencies of all children are obtained in $O(1)$ time, using $O(1)$ additional words per traversal state. Together with the encoded trie construction, this proves the claimed preprocessing time and space bounds. □

G.2 Subtree-Size Oracle

We next implement the subtree-size oracle required by the private top-down search. Given a representation (α, v) of a conceptual node, the oracle returns the exact size of its subtree in $\mathcal{T}$ without materializing that subtree. We decompose this size into a local contribution from T_k and a downstream contribution from larger-scale instances attached below (α, v).

The local contribution is stored once per shared component tree, while downstream contributions are propagated from larger to smaller scales.

Definition G.2 (Attachment points)**.** *Fix $k \in \mathcal{H}$, $\alpha \in \mathsf{Enc}(\mathcal{C}_k)$, and a node v of T_k. We define the attachment points in the subtree of $\mathcal{T}$ rooted at (α, v) as follows. If $2k \in \mathcal{H}$, let*

$$\mathsf{Attach}_k(\alpha, v) \doteq \big\{(\alpha, w) : \ w \text{ lies in the subtree of } T_k \text{ rooted at } v, \quad \mathtt{str}((\alpha, w)) \in \mathsf{Enc}(\mathcal{C}_{2k})\big\}.$$

Otherwise, let $\mathsf{Attach}_k(\alpha, v) = \varnothing$.

These are precisely the nodes at which Algorithm 3 attaches scale-$2k$ component-tree instances. The local contribution depends only on the position v in the shared component tree T_k, whereas the remaining contribution comes from the subtrees introduced at these attachment points. This motivates the following two quantities.

Definition G.3 (Local and downstream subtree sizes)**.** *Fix $k \in \mathcal{H}$, $\alpha \in \mathsf{Enc}(\mathcal{C}_k)$, and a node v of T_k. We distinguish the following two contributions to the subtree of $\mathcal{T}$ rooted at (α, v):*

- *The* local subtree size $L_k(v)$ *is the number of nodes in the subtree rooted at v within the uncompressed component tree T_k, including v itself.*
- *The* downstream subtree size $D_k(\alpha, v)$ *is defined as follows. For each $x \in \mathsf{Attach}_k(\alpha, v)$, let $\mathcal{T}_x^{\downarrow}$ denote the entire subtree of $\mathcal{T}$ rooted at x. Then*

$$D_k(\alpha, v) \doteq \sum_{x \in \mathsf{Attach}_k(\alpha, v)} \big(|\mathcal{T}_x^{\downarrow}| - 1\big).$$

The subtraction excludes each attachment point x, which is already counted by $L_k(v)$. Consequently, the local and downstream terms count disjoint sets of nodes whose union is the subtree of $\mathcal{T}$ rooted at (α, v). The subtree-size query is therefore answered by

$$\mathcal{O}^{\mathrm{size}}(\alpha, v) = L_k(v) + D_k(\alpha, v). \tag{2}$$

It remains to preprocess $L_k(v)$ and $D_k(\alpha, v)$ without materializing the component-tree instances. The following lemma summarizes the resulting subtree-size oracle.

Lemma G.4 (Subtree-size oracle)**.** *Assume that $n, \ell \geq 2$, $\varepsilon_{\mathrm{tree}} \in O(\log(n\ell))$, and $|\Sigma| \leq (n\ell)^{O(1)}$. For every $k \in \mathcal{H}$, $\alpha \in \mathsf{Enc}(\mathcal{C}_k)$, and node v of T_k, the subtree-size oracle returns the exact number of conceptual nodes in the subtree of $\mathcal{T}$ rooted at (α, v). $\mathcal{O}^{\mathrm{size}}$ can be constructed in $O(n\ell + |\Sigma|)$ expected time and stored using $O(n\ell + |\Sigma|)$ words. Given a representation of (α, v), the oracle answers the corresponding subtree-size query in $O(1)$ expected time.*

Proof of Lemma G.4. Since the local and downstream contributions are disjoint and together comprise the subtree of $\mathcal{T}$ rooted at (α, v), correctness follows immediately from Definition G.3.

Local Subtree Sizes. Retain the root, branching nodes, and suffix endpoints as explicit nodes of the compact representation. Write $u \prec_{\mathrm{e}} w$ when u and w are the endpoints of one compact edge, with no explicit node between them. We first compute the local subtree sizes $L_k(v)$. For each compact edge uw, let $\lambda(u, w)$ denote its length in the conceptual tree. A postorder traversal of the compact representation computes, for every explicit node u,

$$L_k(u) = 1 + \sum_{w : u \prec_{\mathrm{e}} w} (\lambda(u, w) - 1 + L_k(w)).$$

A leaf has local size one. Now suppose an implicit node v of T_k lies inside the compact edge from u to w. Then

$$L_k(v) = \lambda(u, w) - \lambda(u, v) + L_k(w).$$

Thus, $L_k(v)$ can be recovered in $O(1)$ time given the edge length, the offset, and $L_k(w)$ stored at the endpoint w. Since $L_k(v)$ is independent of attachment α, these values are computed and stored only once for each shared component tree T_k. By Lemma B.9, that representation has $O(k|\mathcal{C}_k|)$ explicit nodes and edges. Thus all local-size computations take $O(\sum_{k \in \mathcal{H}} k|\mathcal{C}_k|) = O(B\ell) = O(n\ell)$ time and words.

Downstream Subtree Sizes. We compute the downstream subtree sizes in decreasing order of the dyadic scales. At the largest scale, there are no attachment points, and hence $D_\ell(\alpha, v) = 0$. Now fix k such that $2k \in \mathcal{H}$, and let ρ_{2k} denote the root of T_{2k}. By construction, every attachment point $x \in \mathsf{Attach}_k(\alpha, v)$ is also represented as $(\mathtt{str}(x), \rho_{2k})$, the root of the attached scale-$2k$ instance. Therefore,

$$|\mathcal{T}_x^{\downarrow}| = \mathcal{O}^{\mathrm{size}}(\mathtt{str}(x), \rho_{2k}).$$

Since the scales are processed in decreasing order, the right-hand side has already been computed. It follows from Definition G.3 that

$$D_k(\alpha, v) = \sum_{x \in \mathsf{Attach}_k(\alpha, v)} \big(\mathcal{O}^{\mathrm{size}}(\mathtt{str}(x), \rho_{2k}) - 1\big). \tag{3}$$

To implement this recurrence, we propagate each attachment contribution upward within the corresponding component-tree instance. The values $D_k(\alpha, v)$ are initialized implicitly to zero. Only nonzero values are stored, and a missing record is interpreted as zero.

Fix $\beta \in \mathsf{Enc}(\mathcal{C}_{2k})$, and write $\beta = \alpha \circ \gamma$, where $\alpha, \gamma \in \mathsf{Enc}(\mathcal{C}_k)$. Let v_γ be the node of T_k satisfying $\mathtt{str}(v_\gamma) = \gamma$. Then $(\alpha, v_\gamma) \in \mathsf{Attach}_k(\alpha, v)$ if and only if v lies on the root-to-v_γ path in T_k. We therefore add $\mathcal{O}^{\mathrm{size}}(\beta, \rho_{2k}) - 1$ to $D_k(\alpha, v)$ for every node v on this path.

Recall that $r \doteq \lceil \log|\Sigma| \rceil + 1$ is the encoded length of each original symbol. Since $\mathtt{str}(v_\gamma) = \gamma$ and γ has length kr, the root-to-v_γ path in T_k contains $kr + 1$ conceptual nodes. We traverse this conceptual path through the branching nodes, using edge offsets for non-branching nodes. Thus, processing β requires $O(kr)$ updates. Since every $\beta \in \mathsf{Enc}(\mathcal{C}_{2k})$ determines one attachment point, the total number of downstream updates from scale k to scale $2k$ is

$$O\left(kr|\mathcal{C}_{2k}|\right).$$

Using $|\mathcal{C}_{2k}| \le B \doteq \varepsilon_{\mathrm{tree}} n / \log^2(n\ell)$, $|\Sigma| \le (n\ell)^{O(1)}$, $r = \lceil \log|\Sigma| \rceil + 1 \in O(\log(n\ell))$, and $\varepsilon_{\mathrm{tree}} \in O(\log(n\ell))$, and summing over all dyadic scales, the total number of updates is

$$\sum_{k \in \mathcal{H}: 2k \in \mathcal{H}} O\left(kr|\mathcal{C}_{2k}|\right) = O\left(rB \sum_{k \in \mathcal{H}: 2k \in \mathcal{H}} k\right) = O(rB\ell) = O(n\ell). \tag{4}$$

The downstream values are stored in a hash table indexed by (α, v), and each update takes $O(1)$ expected time. Since each update creates or modifies at most one hash-table entry, the number of stored entries is at most the total number of updates. Thus, all downstream values can be computed in $O(n\ell)$ expected time and represented using $O(n\ell)$ words.

Initial Trie and Query Time. At a leaf of the initial trie labeled by $\alpha \in \mathsf{Enc}(\mathcal{C}_1)$, use $\mathcal{O}^{\mathrm{size}}(\alpha, \rho_1)$ as the size of the merged node. Compute the sizes at all remaining initial-trie nodes by the recurrence one plus the sum of the children's sizes. This takes $O(1 + r|\mathcal{C}_1|) = O(n\ell)$ additional time and words, and handles representations (ϵ, v). An empty candidate set leaves just the root.

For a node in a component instance, recover $L_k(v)$ from its explicit-node or edge-offset representation and retrieve $D_k(\alpha, v)$ by one hash-table lookup. Their sum answers the query in $O(1)$ expected time. At an attachment leaf, $L_k(v) = 1$, and the downstream term is the size of the attached next-scale subtree minus one, so both representations give the same answer. The stored sizes occupy $O(1)$ words because Lemma H.1 bounds the conceptual size by $(n\ell)^{O(1)}$.

Complexity. $\mathcal{O}^{\mathrm{size}}$ can be constructed in $O(n\ell + |\Sigma|)$ expected time and stored using $O(n\ell + |\Sigma|)$ words. Given a representation of (α, v), the oracle answers the corresponding subtree-size query in $O(1)$ expected time. □

H The Oracle Monotone Search Tree

Bringing together Appendices E–G, the oracle monotone search tree is constructed as follows. First, the private dyadic candidate sets $\{\mathcal{C}_k\}_{k \in \mathcal{H}}$ are constructed. Then, for input alphabets with non-constant, the dyadic candidate sets are encoded to a constant size alphabet to create $\{\mathsf{Enc}(\mathcal{C}_k)\}_{k \in \mathcal{H}}$. Subsequently, for each $k \in \mathcal{H}$ a component tree T_k is constructed from $\mathsf{Enc}(\mathcal{C}_k)$ using a character aligned r-space sparse suffix tree. To build the tree, each candidate in $\mathsf{Enc}(\mathcal{C}_k)$ (conceptually) prefixes a (virtual) copy of T_k. Then, whenever a scale-k prefixed copy reaches a candidate in $\mathsf{Enc}(\mathcal{C}_{2k})$, continue from that node using the corresponding scale-$2k$ component.

The accompanying frequency oracle is built from $\mathsf{Enc}(\mathcal{D})$ using a character aligned r-spaced sparse suffix tree augmented with counters at branching nodes. Finally, the subtree size oracle is constructed

by computing local component sizes and propagating cross-scale contributions from larger to smaller scales.

This process is formalized in Algorithm 4.

Algorithm 4: CONSTRUCTORACLEMONOTONETREE

Input: A dataset $\mathcal{D}$, privacy parameter $\varepsilon_{\mathrm{tree}}$, and failure probability β_{tree}
Output: A private succinct representation of the conceptual tree $\mathcal{T}$ and a subtree-size oracle $\mathcal{O}^{\mathrm{size}}$; an internal exact-frequency oracle $\mathcal{O}^{\mathrm{freq}}$

Privately construct a candidate set $\mathcal{C}_k \subseteq \Sigma^k$ for every $k \in \mathcal{H}$;
Encode the candidates over a constant-size alphabet and, for every $k \in \mathcal{H}$, build one shared component tree T_k from their character-aligned suffixes;
Conceptually prefix T_k by every candidate in $\mathsf{Enc}(\mathcal{C}_k)$;
Whenever a scale-k prefixed copy reaches a candidate in $\mathsf{Enc}(\mathcal{C}_{2k})$, continue from that node using the corresponding scale-$2k$ component;
Build a character-aligned sparse suffix tree from $\mathsf{Enc}(\mathcal{D})$ to implement $\mathcal{O}^{\mathrm{freq}}$;
Compute local component sizes and propagate cross-scale contributions from larger to smaller scales to implement $\mathcal{O}^{\mathrm{size}}$;
return *the private representation of* $\mathcal{T}$*,* $\mathcal{O}^{\mathrm{size}}$*, and the internal oracle* $\mathcal{O}^{\mathrm{freq}}$;

H.1 PROOF OF LEMMA 4.2

We first establish the size (in terms of number of nodes), memory, and navigation guarantees required by Lemma 4.2. The conceptual size of $\mathcal{T}$ controls the error of the private top-down search, whereas the size of its representation controls the actual space usage. We show that $\mathcal{T}$ has polynomial conceptual size, admits a linear-space representation constructible in near-linear time, and supports constant-time navigation.

Lemma H.1 (Size, representation, and navigation)**.** *Assume that* $n, \ell \geq 2$*,* $\varepsilon_{\mathrm{tree}} \in O(\log(n\ell))$*, and* $|\Sigma| \leq (n\ell)^{O(1)}$*. The conceptual tree* $\mathcal{T}$ *has* $(n\ell)^{O(1)}$ *nodes. Given the dyadic candidate sets, a representation of* $\mathcal{T}$ *can be constructed in* $O(n\ell + |\Sigma|)$ *expected time and stored using* $O(n\ell + |\Sigma|)$ *words. Given the representation of a conceptual node, all its children can be obtained in* $O(1)$ *expected time.*

Proof of Lemma H.1. The representation stores the encoded candidate sets, the initial trie of $\mathsf{Enc}(\mathcal{C}_1)$, one shared compact representation of each component tree T_k, and one constant-size attachment record for each candidate in $\mathsf{Enc}(\mathcal{C}_{2k})$, over all $k, 2k \in \mathcal{H}$. We analyze these components separately.

Shared Component Trees. We use the following simplified form of Lemma B.9, which states the encoding assumptions explicitly and is proved in Appendix B.4.

Lemma H.2 (Encoded sparse suffix tree for a string collection)**.** *Let* $X_1, \ldots, X_m$ *be nonempty strings over* Σ*. Assume that every symbol in* Σ *and every integer in* $[m]$ *fits in one word. The path-compressed trie of*

$$\left\{\mathsf{Enc}\big(X_i[p \ldots |X_i|]\big) : i \in [m],\ p \in [|X_i|]\right\}$$

can be constructed in $O\big(\sum_{i=1}^{m} |X_i|\big)$ *expected time and stored using* $O\big(\sum_{i=1}^{m} |X_i|\big)$ *words.*

Fix $k \in \mathcal{H}$, and apply Lemma H.2 to the collection $\mathcal{C}_k$. This collection contains $m = |\mathcal{C}_k| \leq B = \frac{\varepsilon_{\mathrm{tree}} n}{\log^2(n\ell)} = (n\ell)^{O(1)}$ strings, so every integer in $[m]$ fits in one word. The path-compressed trie returned by the lemma contains precisely the strings $\mathsf{Enc}(Q)$, where Q is a suffix of some $P \in \mathcal{C}_k$. Hence, it is a compact representation of T_k.

Throughout, T_k denotes the uncompressed conceptual component tree. Its compact representation stores only the explicit nodes and compact edges. A node of T_k is represented either by an explicit node or by a compact edge together with an offset on that edge.

The total input length at scale k is $k|\mathcal{C}_k|$. Thus, the compact representation of T_k can be constructed in $O(k|\mathcal{C}_k|)$ expected time and stored using $O(k|\mathcal{C}_k|)$ words. During the construction, we also store, for every $\gamma \in \mathsf{Enc}(\mathcal{C}_k)$, a pointer to the explicit leaf in the compact representation of T_k that represents γ. These pointers use one word per candidate and can be recorded within the preceding time and space bounds.

Using $|\mathcal{C}_k| \le B$ and summing over the dyadic scales, the total construction time and space are both

$$O\left(B \sum_{k \in \mathcal{H}} k\right) = O(B\ell) = O(n\ell).$$

Attachment Records. Fix $k \in \mathcal{H}$ such that $2k \in \mathcal{H}$. For every $\beta \in \mathsf{Enc}(\mathcal{C}_{2k})$, write $\beta = \alpha \circ \gamma$, $\alpha, \gamma \in \mathsf{Enc}(\mathcal{C}_k)$. The pointer stored for γ gives the unique node v_γ of T_k satisfying $\mathtt{str}(v_\gamma) = \gamma$. We store a constant-size attachment record mapping (α, v_γ) to the equivalent next-scale representation (β, ρ_{2k}).

The attachment records require

$$O\left(\sum_{k \in \mathcal{H}: 2k \in \mathcal{H}} |\mathcal{C}_{2k}|\right) = O(B \log \ell)$$

words. They can be constructed in $O(B \log \ell)$ expected time using a hash table indexed by (α, v_γ). These bounds are subsumed by the bounds for the shared component trees.

Conceptual Size. We next bound the conceptual size of $\mathcal{T}$. Each component tree T_k is generated from at most $B = \varepsilon_{\mathrm{tree}} n / \log^2(n\ell) \in (n\ell)^{O(1)}$ encoded strings, each of length at most $\ell r \in O(\ell \log|\Sigma|) \subseteq (n\ell)^{O(1)}$. Hence, each T_k has $(n\ell)^{O(1)}$ conceptual nodes. Each scale contains at most $B \in (n\ell)^{O(1)}$ component-tree instances, and there are $O(\log \ell)$ dyadic scales. The initial trie also has polynomial size, while cross-scale merges can only decrease the number of distinct nodes. Thus, $\mathcal{T}$ has $(n\ell)^{O(1)}$ conceptual nodes.

Navigation. A node of T_k is represented either by an explicit node of its compact representation or by a compact edge together with an offset on that edge. If the representation lies inside a compact edge, the node has a unique child in T_k, obtained by incrementing the offset. If the representation is an explicit node, the children in T_k are obtained from its outgoing compact edges. Since the encoded alphabet $\{0, 1, \$\}$ has constant size, these operations take $O(1)$ time.

Consider a node represented by (α, v) in $T_k^{(\alpha)}$. A hash-table lookup determines whether (α, v) is a cross-scale attachment point. If no attachment record is present, its children are obtained from the compact representation of T_k. Otherwise, the attachment record gives the equivalent next-scale representation (β, ρ_{2k}). Since every attachment point is a leaf of its scale-k component tree, all children of the merged node belong to the attached instance $T_{2k}^{(\beta)}$ and are obtained from the root of T_{2k}.

For nodes in the initial trie, we use representations (ϵ, v), where v is the corresponding initial-trie node. Their children are obtained directly from the initial-trie representation. If v is labeled by a candidate $\alpha \in \mathsf{Enc}(\mathcal{C}_1)$, the node is merged with the root of $T_1^{(\alpha)}$, and its children are obtained from that root.

Thus, all children of a conceptual node can be obtained using a constant number of compact-tree operations and hash-table lookups. Since $\mathcal{T}$ has constant branching factor, each navigation query takes $O(1)$ expected time.

Overall Complexity. The encoded candidate sets use $O(rB\ell)$ words. The compact representations of the shared component trees use $O(B\ell)$ words, and the attachment records use $O(B \log \ell)$ words. Since $rB\ell = O(n\ell)$, the complete representation uses $O(n\ell + |\Sigma|)$ words. The representation can be constructed in $O(n\ell + |\Sigma|)$ expected time. This proves Lemma H.1. □

Proof of Lemma 4.2. Run the candidate construction of Lemma E.2 with parameters $\varepsilon_{\mathrm{tree}}$ and β_{tree}, and abort if that construction aborts. Let $\mathcal{E}_{\mathrm{cand}}$ be the simultaneous success event in its analysis. This

event has probability at least $1 - \beta_{\mathrm{tree}}$; on it, the construction does not abort, every candidate set contains all sufficiently frequent strings of its length, and

$$|\mathcal{C}_k| \le B \doteq \frac{\varepsilon_{\mathrm{tree}} n}{\log^2(n\ell)}, \qquad \mathcal{C}_{2k} \subseteq \mathcal{C}_k \circ \mathcal{C}_k.$$

Assemble $\mathcal{T}$ as in Algorithm 3.

Privacy. The encoded candidates, component-tree representations, attachment records, and subtree-size oracle are functions only of the private candidate sets and public parameters. Their construction is therefore post-processing of the $\varepsilon_{\mathrm{tree}}$-DP candidate construction. The exact-frequency oracle is built separately from the input dataset and remains internal to the algorithm, as specified in Algorithm 4; neither its data structure nor its exact answers are released by the tree construction. Its use in the private search is accounted for in the proof of Theorem 4.1.

Structure and navigation. The initial trie contains a node for every candidate in $\mathsf{Enc}(\mathcal{C}_1)$. Inductively, if every scale-k candidate has its prefixed component-tree instance, then every $\beta \in \mathsf{Enc}(\mathcal{C}_{2k})$ can be written as $\beta = \alpha \circ \gamma$, with $\alpha, \gamma \in \mathsf{Enc}(\mathcal{C}_k)$. The string γ is represented in T_k, so the instance prefixed by α contains the node spelling β. The next-scale instance can therefore be attached at this node. The attachment is at a leaf: γ has the maximum component depth kr, where $r = \lceil \log|\Sigma| \rceil + 1$. Thus the assembly preserves the trie property and does not create additional nodes spelling the same string.

The encoded alphabet $\{0, 1, \$\}$ gives constant branching factor. Every node belongs either to the initial trie or to a scale-k instance whose prefix and extension each have length at most kr, so the depth is at most $2\ell r$. As usual, a unary alphabet may be padded with an unused symbol, and logarithmic alphabet factors are read as $\log \max\{2, |\Sigma|\}$. Lemma H.1 supplies polynomial conceptual size, a succinct representation, and constant expected time for obtaining all children of a represented node. This establishes Items 1 and 2.

Frequency labels and coverage. Assign every node x the label $f_{\mathcal{T}}(x) = f^{\mathrm{enc}}_{\mathcal{D}}(\mathtt{str}(x))$. Lemma G.1 gives nonnegative monotone labels, agreement with the original substring frequencies at codeword boundaries, and total frequency mass at most $n\ell$ at every depth. These are Items 3 and 4.

Condition on $\mathcal{E}_{\mathrm{cand}}$, and let P be a substring of length at most ℓ whose frequency is at least the coverage threshold $\tau_{\top}$ from Lemma E.2. If $|P| = 1$, then $P \in \mathcal{C}_1$, so its encoding occurs in the initial trie. Otherwise choose the dyadic scale k such that $k < |P| \le 2k$, and write $P = X \circ Y$ with $|X| = k$. By Lemma E.1, $X \in \mathcal{C}_k$ and Y is a suffix of a candidate in $\mathcal{C}_k$. Hence T_k contains the path spelling $\mathsf{Enc}(Y)$, and the instance prefixed by $\mathsf{Enc}(X)$ contains a node spelling $\mathsf{Enc}(P)$. This proves Item 5 with

$$\tau_{\top} \in O\left(\frac{\ell \log(n\ell)}{\varepsilon_{\mathrm{tree}}} \log \frac{n\ell}{\beta_{\mathrm{tree}}}\right).$$

Oracles and resources. Lemmas G.1 and G.4 provide the two oracles in Item 6. During a top-down traversal, the representation is augmented with the current frequency-oracle locus, which requires a constant number of words and is maintained when moving to a child. Subtree sizes depend only on the candidate sets. The initial-trie nodes are handled by the bottom-up computation described in Appendix G.2.

Candidate generation takes $O(n\ell \log \ell + |\Sigma|)$ expected time and $O(n\ell + |\Sigma|)$ words by Lemma E.2. The representation and both oracles add $O(n\ell + |\Sigma|)$ expected time and words by Lemmas H.1, G.1, and G.4. In particular, storing the encoded candidates does not introduce an extra logarithmic factor, since

$$r \sum_{k \in \mathcal{H}} k|\mathcal{C}_k| = O(rB\ell) = O(n\ell)$$

under the assumptions on $\varepsilon_{\mathrm{tree}}$ and $|\Sigma|$. Summing the construction costs gives the stated bounds. All coverage and non-abortion guarantees hold on the single event $\mathcal{E}_{\mathrm{cand}}$, so their total failure probability is at most β_{tree}. □