Distributed Computing Made Secure: A New Cycle Cover Theorem*

Merav Parter Eylon Yogev†

Abstract

In the area of distributed graph algorithms a number of network's entities with local views solve some computational task by exchanging messages with their neighbors. Quite unfortunately, an inherent property of most existing distributed algorithms is that throughout the course of their execution, the nodes get to learn not only their own output but rather learn quite a lot on the inputs or outputs of many other entities. This leakage of information might be a major obstacle in settings where the output (or input) of network's individual is a private information (e.g., distributed networks of selfish agents, decentralized digital currency such as Bitcoin, voting systems).

While being quite an unfamiliar notion in the classical distributed setting, the notion of secure multi-party computation (MPC) is one of the main themes in the Cryptographic community. Yet despite all extensive work in the area, no existing algorithm fits the framework of classical distributed models in which there are no assumptions on the graph topologies and only messages of bounded size are sent on the edges in each round.

In this paper, we introduce a new framework for *secure distributed graph algorithms* and provide the first *general compiler* that takes any "natural" non-secure distributed algorithm that runs in r rounds, and turns it into a secure algorithm that runs in $\widetilde{O}(r \cdot D \cdot \mathsf{poly}(\Delta))$ rounds where Δ is the maximum degree in the graph and D is its diameter. We also show that this is nearly (existentially) optimal for any round-by-round compiler for bounded degree graphs.

The main technical part of our compiler is based on a new cycle cover theorem: We show that the edges of every bridgeless graph G of diameter D can be covered by a collection of cycles such that each cycle is of length $\widetilde{O}(D)$ and each edge of the graph G appears in $\widetilde{O}(1)$ many cycles.

*Department of Computer Science and Applied Mathematics, Weizmann Institute of Science, Rehovot 76100, Israel. Emails: {merav.parter,eylon.yogev}@weizmann.ac.il.

†Supported in part by grants from the Israel Science Foundation grant no. 950/16.

Contents

1 Introduction **1**
- 1.1 Our Approach and Results . . . 2
  - 1.1.1 From Security Requirements to Graph Structures . . . 2
  - 1.1.2 Secure Simulation . . . 4

2 Our Techniques **5**
- 2.1 Low Congestion Cycle Covers . . . 5
- 2.2 From Low Congestion Covers to Secure Simulation . . . 8

3 Preliminaries and Model **10**
- 3.1 Distributed Algorithms . . . 10
- 3.2 Cryptography with Perfect Privacy . . . 12

4 Low-Congestion Covers **13**
- 4.1 Cycle Cover . . . 13
  - 4.1.1 Covering Non-Tree Edges . . . 13
  - 4.1.2 Covering Tree Edges . . . 19
- 4.2 Private Neighborhood Trees . . . 27

5 Secure Simulation via Low-Congestion Covers **29**
- 5.1 Our Framework . . . 30
- 5.2 Secure Simulation of a Single Round . . . 31
- 5.3 The Final Secure Algorithm . . . 33

6 Distributed Computation of Low Congestion Covers **35**
- 6.1 Cycle Cover . . . 35
  - 6.1.1 Distributed Algorithm for Covering Non-Tree Edges . . . 36
  - 6.1.2 Analysis of Algorithm DistNonTreeCover . . . 41
  - 6.1.3 Distributed Algorithm for Covering Tree Edges . . . 45
- 6.2 Additional Low Congestion Covers . . . 47
- 6.3 Pre-processing for Improved Cover Structures . . . 48

7 Discussion and Future Work **48**

A Balanced Partitioning of a Tree **53**

B Missing Details for Algorithm DistCycleCover **53**

1 Introduction

In *distributed graph algorithms* (or network algorithms) a number of individual entities are connected via a potentially large network. Starting with the breakthrough by Awerbuch et al. [AGLP89], and the seminal work of Linial [Lin92], Peleg [Pel00] and Naor and Stockmeyer [NS95], the area of distributed graph algorithm is growing rapidly. Recently, it receives considerably more theoretical and practical attention motivated by the spread of multi-core computers, cloud computing, and distributed databases. We consider the standard synchronous message passing model where in each round $O(\log n)$ bits can be transmitted over every edge where n is the number of entities (the CONGEST model).

The common principle underlying all distributed algorithms (regardless of the model specification) is that the input of the algorithm is given in a *distributed format*, and consequently the goal of each vertex is to compute its *own* part of the output, e.g., whether it is a member of a computed maximal independent set, its own color in a valid coloring of the graph, its incident edges in the minimum spanning tree, or its chosen edge for a maximal matching solution. In most distributed algorithms, throughout execution, vertices learn much more than merely their own output but rather collect additional information on the input or output of (potentially) many other vertices in the network. This seems inherent in many distributed algorithms, as the output of one node is used in the computation of another. For instance, most randomized coloring (or MIS) algorithms [Lub86, BE13, BEPS16, HSS16, Gha16, CPS17] are based on the vertices exchanging their current color with their neighbors in order to decide whether they are legally colored.

In cases where the data is sensitive or private, these algorithms may raise security concerns. To exemplify this point, consider a voting task for a distributed network, where the goal is to elect the candidate with most votes in the network. This is a rather simple distributed task: construct a BFS tree and let the nodes send the votes from the leaves to the root where each intermediate node sends to its parent in the tree, the sum of all votes for each candidate. While the output goal has been achieved, privacy has been compromised as intermediate nodes learn more information regarding the votes of their subtrees. As privacy in election processes is a fundamental property of any "democratic system", it is desirable to design a secure distributed voting algorithm, in which information of other nodes' votes do not get revealed throughout the course of execution. Additional motivation for secure distributed computation are settings that involve private medical data, networks of selfish agents with private utility functions or even decentralized digital currency such as the Bitcoin.

Whereas much effort in recent years has been devoted to improving the round complexities of various distributed algorithms, e.g., coloring, MIS, Lovász Local lemma [Suo13, BEK14, FHK16, BEPS16, Gha16, CPS17, CHL+17] and many more, the challenge of making these algorithms secure – was left behind. To this point, we have no distributed algorithms for general graphs in the standard CONGEST model that are *secure* and *efficient* compared to their *non-secure* counterparts.

While being a rather virgin objective in the distributed graph algorithm setting, the notion of secure multi-party computation (MPC) is one of the main themes in the Cryptographic community. The goal of an MPC protocol is to allow parties to jointly compute a function of their inputs without revealing any about their input except the output of the function. There has been tremendous work on MPC protocols, from general feasibility results [Yao82, GMW87, BGW88, CCD88] to efficient protocols for specific functionalities [BNP08, BLO16]. Despite the large progress, almost all prior protocols require a private channel between every two nodes in the network in order to secure

compute the desired function. This, of course, defeats the whole purpose of distributed computing. Only a handful of works have considered a general graph interaction patterns [HLP11, HIJ+16, GGG+14, BGI+14], or locality of MPC protocols [BGT13, CCG+14, BIPW17]. Unfortunately, they all have many major drawbacks: they rely on heavy computational assumptions (e.g., obfuscation), they assume a trusted setup phase (which is often not reusable), they require many rounds of interaction, they do not obey bandwidth limitations and they assume specific interaction patterns (e.g., star topology).

Despite some common interests, up to this point, both areas of distributed graph algorithms and secure multi-party computation have been developed in almost total isolation, each in its own community with different requirements and goals in mind. The distributed area puts most emphasis on locality, while the study of MPC focuses mostly on security. In this paper, we aim towards bridging this gap and combine the goods from both worlds, by tackling the following question:

> *How to design distributed algorithms that are both efficient (in terms of round and bandwidth complexity) and secure (where nothing is learned but the desired output)?*

One tedious way to attack this challenge is to go through the most popular distributed problems (e.g., coloring, MIS, matching) and design a secure distributed algorithm for each of them one by one. Much more desirable, however, is to have a general recipe for adding security to existing algorithms, while incurring a small overhead in the round complexity.

Towards this end, we introduce the first *general compiler* that can take any (possibly insecure) distributed algorithm to one that has perfect security (a notion that will be explained next). The compiled algorithm respects the same bandwidth limitations, relies on no setup phase nor on any computational assumption and works for (almost) any graph, while paying an overhead in the number of rounds, to the extent that it is almost existentially needed.

This quite general and powerful framework is made possible due to fascinating connections between "secure cryptographic definitions" and natural combinatorial graph properties. Most notably is the *cycle cover* of a graph. While cycle covers have been studied in the literature, e.g., the well-known double cycle cover conjecture by Szekeresand and Seymour [JT92]; the Chinese postman problem by Edmond [EJ73], none of the known results satisfy our requires. Instead, we prove a new theorem regarding cycle covers with low congestion which we foresee being of independent interest and exploited in future work.

1.1 Our Approach and Results

1.1.1 From Security Requirements to Graph Structures

We demonstrate our approach with a simple example. Let G be an n-vertex graph. Suppose that each vertex u has an input x_u and that each pair of neighbors u, v in the graph (with inputs x_u, x_v) wishes to compute a function $f(x_u, x_v)$ securely, i.e., each party should learn the output $f(x_u, x_v)$ but "nothing more" (the precise notion of security will be later elaborated).

Kushilevitz [Kus89] showed that almost all (non-trivial) functions cannot be computed between u and v with this notion of privacy, and this was generalized for weaker notions of (information theoretic) privacy as well [FJS14]. To circumvent these barriers, Feige, Kilian and Naor [FKN94] (later generalized by [IK97]) suggested a "minimal model" called PSM[1] where a third party, s, aids

[1]The term PSM stands for Private Simultaneous Messages.

the computation. In the model, u and v share private randomness (not known to s) and each sends a single message to s which depends on its own input and the shared randomness. This allows s to compute the output $f(x, y)$ while learning nothing more.

In this paper we introduce *Distributed* PSM *protocols* which generalize the standard two party PSM protocols to general graphs. To provide secure communication between neighboring nodes u, v, we need to find a third node s in the graph so that both u and v can communicate with s without seeing each other's messages. This requirement translates into covering the edge (u, v) with a cycle containing the edge. The number of rounds of the protocol (e.g., sending messages on the cycle to w) is proportional to the length of the cycle. Furthermore, as we wish to run such a protocol for all edges of the graph simultaneously, we need to cover all edges in the graph while having each individual edge participating in a small number of cycles (this is desirable since the bandwidth is limited).

Low Congestion Covers. These requirements motivate the genuine definition of $(\mathtt{d}, \mathtt{c})$-cover $\mathcal{C}$ which is a collection of cycles of length at most $\mathtt{d}$ such that each edge appears at least once and at most $\mathtt{c}$ many times on each of the cycles (the congestion of the cover). Given a $(\mathtt{d}, \mathtt{c})$-cycle cover, we can have all vertices of the graph compute a function $f(x_u, x_v)$ for every edge in the graph simultaneously in $\widetilde{O}(\mathtt{d} + \mathtt{c})$ rounds[2] (in the CONGEST models). A-priori, it is not clear that cycle covers that enjoy both low congestion and short lengths even exist, nor if it is possible to efficiently find them. Perhaps quite surprisingly, we prove the following theorem regarding cycle cover. Throughout, we will have an n-vertex graph G and use the notation $\widetilde{O}$ is the "Big O" notation that hides $\mathsf{polylog}(n)$ factors.

Theorem 1 (Low Congestion Cycle Cover)**.** *Every 2-edge connected graph with diameter D has a $(\mathtt{d}, \mathtt{c})$-cycle cover where $\mathtt{d} = \widetilde{O}(D)$ and $\mathtt{c} = \widetilde{O}(1)$. That is, the edges of G can be covered by cycles such that each cycle is of length at most $\mathtt{d}$ and each edge participates in at most $\mathtt{c}$ cycles.*

We note that this theorem is existentially tight, in the sense that the $\Theta(D)$ factor is necessary (consider a cycle graph). Although several variants of cycle cover problems have been considered in the literature, none of them fit our objective of having both short cycles and small congestion. For instance the k-cycle cover problem aims to minimize only the congestion, i.e., restricting each edge to appear on at most k cycles while allowing arbitrary long cycles. On the other hand, in the *minimum cycle cover* problem that objective is minimize the total length of all cycles in the cover (i.e., instead of minimizing the length of the longest cycle as in our setting) [Fan97].

In order to use these covers in our compiler, we build them once, in a prepossessing step, and show that the covers of Theorem 1 can be constructed distributively in $\widetilde{O}(n + D \cdot \Delta)$ rounds (see Lemma 7 and corollary 3). Alternatively, in Section 6 we also provide a much more efficient distributed construction on the expense of having somewhat worse bounds in the quality of the output covers.

Theorem 2 (Distributed Low Congestion Cycle Cover)**.** *For every 2-edge connected graph G with diameter D and every $\epsilon \in [0, 1]$, there is a distributed algorithm that constructs a $(\widetilde{O}(4^{1/\epsilon} \cdot D), \widetilde{O}(n^{\epsilon}))$ cycle cover in $\widetilde{O}(4^{1/\epsilon} \cdot D + n^{\epsilon})$ rounds.*

Our low congestion cycle cover turns out to be a building block for constructing another, more complex, covers which we need in our final secure compiler. We call this building block a *private*

[2]The $\widetilde{O}(\cdot)$ notation hides $\mathsf{polylog}(n)$ factors where n is the number of vertices in the graph.

neighborhood trees. Roughly speaking, the private neighborhood tree collection of a biconnected graph $G = (V, E)$ is a collection of n trees, one per node u_i, where each tree $T(u_i) \subseteq G \setminus \{u_i\}$ contains all the neighbors of u_i but do not contain u_i. Intuitively, the private neighborhood trees allow all neighbors $\Gamma(u_i)$ of all nodes u_i to exchange a secret without u_i. Note that these covers exists if and only if the graph is 2-vertex connected[3]. Similarly to low-congestion cycle covers, we define $(\mathtt{d}, \mathtt{c})$-private neighborhood trees in which each tree $T(u_i)$ has depth at most $\mathtt{d}$ and each edge belongs to at most $\mathtt{c}$ many trees. This allows us to use all trees simultaneously in $\widetilde{O}(\mathtt{d}+\mathtt{c})$ rounds.

Theorem 3 (Private Neighborhood Trees)**.** *Every biconnected graph with diameter D and maximum degree Δ, has a $(\mathtt{d}, \mathtt{c})$-private neighborhood trees for $\mathtt{d} = \widetilde{O}(D \cdot \Delta)$ and $c = \widetilde{O}(D)$.*

Finally, this covering structures can be constructed also distributively by extending the constructions of low-congestion cycle covers (see Lemma 6 in Section 6).

1.1.2 Secure Simulation

The Security Notion. With this graph framework in mind, we return to our original goal of constructing secure distributed algorithms. We want to design an algorithm in which every node u learns its desired output but does not learn anything else about the inputs and outputs of the rest of the nodes in the graph. In our framework, the topology of the graph is not considered private and is not protected by our security notion. While there are many possible ways to define this kind of privacy, we use the strongest possible notion of *perfect privacy* which is information theoretic and relies and no computation assumptions. This notion uses the existence of an (unbounded) simulator, with the following intuition: a node learns nothing, except its own output y, from the messages it receives throughout the execution of the algorithm, if a *simulator* can produce the same output while receiving only y and the graph G.

We achieve security in what is known as the "semi-honest" model, where the adversary, acting as one of the nodes in the graph, is not allowed to deviate from the prescribed protocol, but can run arbitrary computation given all the messages it received. Moreover, we assume that the adversary does no collude with other nodes in the graph. It is possible to extend the security notion to include these notions, more on this is discussed at Section 7.

The General Compiler. In a distributed algorithm, every node has a state and in each round, it updates its state by applying a local function f that depends on the messages it has received from its immediate neighbors. Theorems 1 and 3 provide the required graph framework for computing this function f in a secure and efficient manner (using a distributed PSM) for all nodes in the graph *simultaneously*. The communication complexity of the secure protocol depends on the computational complexity of the function f. In almost all distributed algorithms, the local update function f can be computed in polynomial time and as a result the communication overhead of the secure protocol is negligible. We call this family of functions *natural* and for simplicity of presentation we state our results for natural distributed algorithms. Finally, we can state our main result: any natural distributed algorithm can be compiled to an equivalent one (that is one that has the same output for each node) that is *secure*.

Theorem 4 (Secure Simulation)**.** *Let G be a 2-vertex connected n-vertex graph with diameter D and maximal degree Δ. Let $\mathcal{A}$ be a* natural *distributed algorithm that runs on G in r rounds. Then, $\mathcal{A}$*

[3] A graph $G = (V, E)$ is 2-vertex connected if for all $u \in V$ the graph $G' = (V \setminus \{u\}, E)$ is connected.

can be transformed to an equivalent algorithm $\mathcal{A}'$ with perfect privacy which runs in $\widetilde{O}(rD \cdot \mathsf{poly}(\Delta))$ rounds, using $\widetilde{O}(n + D \cdot \Delta)$ rounds of pre-processing.

We note that our compiler works for any distributed algorithm rather than only on natural ones. The number of rounds will be proportional to the space complexity of the algorithm (an explicit statement for any algorithm can be found in Remark 1). Moreover, to avoid the preprocessing step, one can use Theorem 2 with $\epsilon = 1/\sqrt{\log n}$, to get a secure simulation with $\widetilde{O}(r \cdot 2^{\sqrt{\log n}} \cdot D \cdot \mathsf{poly}(\Delta))$ rounds (with no preprocessing). We observe that the barrier of [Kus89] can be extended to a cycle graph, which shows that the linear dependency in D in the round complexity of our compiler is existentially *unavoidable*. Our results are summarized in Figure 1.

Applications for Known Distributed Algorithms. Theorem 4 enables us to compile almost all of the known distributed algorithms to a secure version of them, examples include computing majority of votes as well as many other examples which we briefly mention next. It is worth noting that deterministic algorithms for problems in which the nodes do not have any input *cannot* be made secure by our approach since these algorithms only depend on the graph topology which we do not try to hide. Our compiler is meaningful for algorithms where the nodes have input or for randomized algorithms which define a distribution over the output of the nodes. For instance, whereas the deterministic coloring algorithms cannot be made secure, the randomized coloring algorithms (see e.g., [BE13]) which sample a random legal coloring of the graph can be made secure. Specifically, we get a distributed algorithm that (legally) colors a graph (or computes a legal configuration, in general), while the information that each node learns at the end is as if a centralized entity ran the algorithm for the entire network, and revealed each node's output privately (i.e., revealing v the final color of v).

MIS, Coloring, Matching and More. Our approach captures global (e.g., MST) as well as many local problems [NS95]. The MIS algorithm of Luby [Lub86] along with our compiler yields $\widetilde{O}(D \cdot \mathsf{poly}(\Delta))$ secure algorithm according to the notion described above. Slight variations of this algorithm also gives the $O(\log n)$-round $(\Delta+1)$-coloring algorithm (e.g., Algorithm 19 of [BE13]). Combining it with our compiler we get a secure $(\Delta+1)$-coloring[4] algorithm with round complexity of $\widetilde{O}(D \cdot \mathsf{poly}(\Delta))$. Using the Matching algorithm of Israeli and Itai [II86] we get an $\widetilde{O}(D \cdot \mathsf{poly}(\Delta))$ secure maximal matching algorithm. Finally, another example comes from distributed algorithms for the Lovász local lemma (LLL) which receives a lot of attention recently [BFH+16, FG17, CP17] for the class of bounded degree graphs. Using [CPS17], most of these (non-secure) algorithms for defective coloring, frugal coloring, and list vertex-coloring can be made secure within $\widetilde{O}(D)$ rounds.

2 Our Techniques

2.1 Low Congestion Cycle Covers

We give an overview of our low congestion cycle cover of Theorem 1. Let $G = (V, E)$ be a 2-edge connected n-vertex graph with diameter D. We begin by observing that all but $2n$ of the edges in the graph can be covered by edge-disjoint cycles of length at most $\log n$. Since the girth of any graph with at least $2n$ edges is $\log n$, we can repeatedly add a short cycle (up to length $\log n$) to the collection and remove it from the graph. The main challenge is in covering these last $2n$ edges,

[4] We observe that Algorithm 19 of [BE13] can implemented with $O(\log \Delta)$ memory.

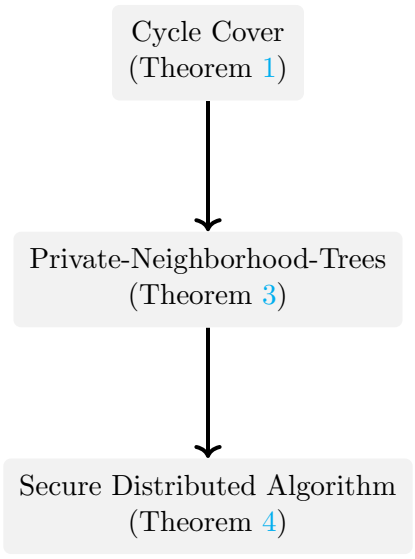

Figure 1: An illustrated summary of our results.

as these edges might be arbitrary with potentially a large diameter.

The remaining $2n$ are covered as follows. We construct a BFS tree T in the graph G. All the remaining uncovered edges in the graph are either tree edges or non-tree edges. Our cover consists of two procedures, where the first constructs a low congestion cycle cover for the *non*-tree edges and the second covers the tree edges.

Covering the Non-Tree Edges. Let E' be the set of uncovered non-tree edges. The main difficulty in covering E' stems from the fact that the diameter[5] of $G \setminus T$ might be large (e.g., $\Omega(n)$). Hence, to cover the edges of E' by *short* cycles, one must use the edges of T. A naïve approach is to cover every edge $e = (u, v)$ in E' by taking its fundamental cycle in T (i.e., using the u-v path in T). Although this yields short cycles, the congestion on the tree edges might become $\Omega(n)$. The key challenge is to use the edges of T (as we indeed have to) in a way that cycles enjoy the small diameter of the graph while not overloading any tree edge more than $\widetilde{O}(1)$ times.

Our approach is based on using the tree T edges only for the purpose of connecting nodes that are somewhat close to each other, and thus the path between them should not occupy too many edges of the tree. To realize this approach, we define a specific way of partitioning the nodes of the tree T to blocks according to E'. In a very rough manner, a block would consist of a set of nodes that have few incident edges in E'. To define these blocks, we number the nodes based on postorder traversal in T and partition them into *blocks* containing nodes with consecutive numbering. The *density* of a block B is the number of edge in E' with one endpoint in B. The blocks are partitioned such that no block has large density (at most some constant), and moreover, the number of blocks is not too large (say, at most $n/8$). The only exception is a block containing a single node, in such a case its density can be unbounded (i.e., even $\Omega(n)$). The end goal is to connect nodes by their tree path only if they reside in the same block.

We then consider the contracted graph obtained by contracting all nodes in a given block into

[5]The graph $G \setminus T$ might be disconnected, when referring to its diameter, we refer to the maximum diameter in each connected component of $G \setminus T$.

one supernode and connecting two supernodes B_1 and B_2, if there is an edge in E' whose one endpoint is in B_1, and the other endpoint is in B_2. This graph is in fact a multigraph, that might contain self-loops or multi-edges. The key idea is that the contracted graph contains only $n' = n/8$ nodes, and hence we can reuse the girth approach from before, and repeatedly find short cycles (length $\log n'$) in it until we are left with at most $2n' = n/4$ edges. These cycles are translated to cycles in the origin graph G by using the tree paths $\pi(u, v, T)$ between nodes u, v belonging to the same supernode (block).

Our key insight is that eventhough paths between two nodes in a block might be long, we show that every tree edge is "used" by at most *two* blocks. That is, for each edge e of the tree, there are at most 2 blocks such that the tree path $\pi(u, v)$ of nodes u, v in the block passes through e. (If a block has only a single node, then it will use no tree edges.) Since the blocks have constant density, we are able to bound the congestion on the edge e. The translation of cycles in the contracted graph to cycles in the original graph yields $O(D \log n)$-length cycles in the original graph where every edge belongs to $O(1)$ cycles.

The above step already covered all but $2n' = n/4$ edges. We continue this process $\log n$ times until all edges of E' are covered, and thus get a $\log n$ factor in the congestion. We note that while producing short cycles of small congestion that cover all the non-tree edges with low congestion, these cycles might be non-simple. To handle that, we add an additional "cleanup" step (procedure SimplifyCycles) which takes the output collection of non-simple cycles and produces a collection of simple ones. In this process, some of the edges in the non-simple cycles might be omitted, however, we prove that only tree edges might get omitted and all non-tree edges remain covered by the simple cycles. This concludes the high level idea of covering the non-tree edges. We note the our blocking definition is quite useful also for distributed implementations. The reason is that although the blocks are not independent, in the sense that the tree path connecting two nodes in a given block pass through other blocks, this independence is very *limited*. The fact that each tree edge is used in the tree paths for only *two* blocks allows us also the work distributively on many blocks simultaneously (see Section 6).

Covering the Tree Edges. Covering the tree edges turns out to be the harder case where new ideas are required. Specifically, whereas in the non-tree edge our goal is to find cycles that use the tree edge as rarely as possible, here we aim to find cycles that cover all edges in the tree, but still avoiding a particular tree edge from participating in too many cycles.

The algorithm for covering the tree edges is recursive, where in each step we split the tree into two edge disjoint subtrees T_1, T_2 that are balanced in terms of number of edges. To perform a recursive step, we would like to break the problem into two independent subproblems, one that covers the edges of T_1 and the other that covers the edges of T_2. However, observe that there might be edges $(u, v) \in T_1$ where the only cycle that covers them[6] passes through T_2 (and vice versa). For every such node $u \in T_1$, let $s(u)$ be the first node in T_2 that appears on the fundamental cycle of the edge (u, v).

To cover these tree edges, we employ two procedures, one on T_1 and the other on T_2 that together form the desired cycles (for an illustration, see Figures 9 and 11). First, we mark all nodes $u \in T_1$ such that their $s(u)$ is in T_2. Then, we use an Algorithm called TreeEdgeDisjointPath (see Lemma 4.3.2 [Pel00]) which solves the following problem: given a rooted tree T and a set of $2k$ marked nodes $M \subseteq V(T)$ for $k \leq n/2$, find a matching of these vertices $\langle u_i, u_j \rangle$ into pairs such that the tree paths $\pi(u_i, u_j, T)$ connecting the matched pairs are *edge-disjoint*.

[6] Recall that the graph G is two edge connected.

We employ Algorithm TreeEdgeDisjointPath on T_1 with the marked nodes as described above. Then for every pair $u_i, u_j \in T_1$ that got matched by Algorithm TreeEdgeDisjointPath, we add a virtual edge between $s(u_i)$ and $s(u_j)$ in T_2. Since this virtual edge is a non-tree edge with both endpoints in T_2, we have translated the dependency between T_1 and T_2 to covering a *non-tree* edge. At that point, we can simply use Algorithm NonTreeCover on the tree T_2 and the non-virtual edges. This computes a cycle collection which covers all virtual edges $(s(u_i), s(u_j))$. In the final step, we replace each virtual edge $(s(u_i), s(u_j))$ with the edge disjoint tree path $\langle u_i, u_j \rangle$ and the paths between u_i and $s(u_i)$ (as well as the path connecting u_j and $s(u_j)$).

This above description is simplified and avoids many details and complications that we had to address in the full algorithm. For instance, in our algorithm, a given tree edge might be responsible for the covering of up to $\Theta(D)$ many tree edges. This prevents us from using the edge disjoint paths of Algorithm TreeEdgeDisjointPath in a naïve manner. In particular, our algorithm has to avoid the multiple appearance of a given tree edge on the same cycle as in such a case, when making the cycle simple that tree edge might get omitted and will no longer be covered. See Section 4.1 for the precise details of the proof, and see Figure 2 for a summary of our algorithm.

Algorithm CycleCover($G = (V, E)$)

1. While there is a cycle of uncovered edges of length at most $\log n$, add it to $\mathcal{C}$.
2. Construct a BFS tree T of G.
3. Let $\widetilde{E} \subseteq E$ be the subset of all uncovered non-tree edges.
4. Repeat $\log n$ times:
   (a) Partition the nodes of T with block density $\mathtt{b}$ with respect to $\widetilde{E}$.
   (b) While there are t edges $(u_1, v_1), \ldots, (u_t, v_t) \in \widetilde{E}$ for $t \leq \log n$ such that for all $i \in [t-1]$, v_i and u_{i+1} are in the same block and v_t and u_1 are in the same block (with respect to the partitioning $\mathcal{B}$): Add the cycle $u_1 \to v_1 \to \pi(v_1, u_2) \to v_2 \to \pi(v_2, u_3) \to v_3 \to \ldots \to v_t \to u_1$ to $\mathcal{C}$.
5. $\mathcal{C} \leftarrow \mathcal{C} \cup$ TreeEdgeCover(T) (see Figure 12).
6. Output SimplifyCycles($\mathcal{C}$).

Figure 2: Procedure for covering non-tree edges.

2.2 From Low Congestion Covers to Secure Simulation

Consider an r-round distributed algorithm $\mathcal{A}$. In a broad view, $\mathcal{A}$ can be considered as a collection of r functions $f_1, \ldots, f_r$. At round i, a node u holds a state σ_i and needs to update its state according to a function f_i that depends on σ_i and the state of its immediate neighbors. We assume that the final state σ_r is the final output of the algorithm for node u.

Our goal is to simulate this process of computing $\sigma_1, \sigma_2, \ldots, \sigma_r$, however, in an oblivious way without knowing any σ_i except the last one σ_r which contains the final output. Thus, we will have

the node u hold an "encrypted" state, $\widehat{\sigma}_i$, instead of the actual state σ_i. This encryption uses a a random mask R such that $\widehat{\sigma}_i \oplus R = \sigma_i$. The key R will be chosen by an arbitrary neighbor v.

The neighbors of u, in turn, hold an encrypted version of messages they would send u in order for him to compute the next state. The key for this encryption is chosen by u itself. Thus, the information required in order to compute the function f_i is now spread out among u and its neighbors. Define a related function f'_i to be a function that gets an encrypted state of u and its neighbors and the corresponding keys. Then the function decrypts the states, computes f_i and finally re-encrypts the new state using a new encryption key. Our goal is to enable u to compute the function f'_i that depends on its neighbors while learning but the output. Actually, we need all nodes $u \in V$ to be able to compute f'_i *simultaneously*.

To achieve this, we use a PSM protocol where u is the server and its neighbors are the parties. The first step of a PSM protocol, is for all parties to share private common randomness (not known to u). This is exactly the reason for introducing the notion of private neighborhood trees. Using these trees, each neighborhood can communicate privately, and moreover, this can be done for all nodes simultaneously (with independent randomness for each neighborhood), with low congestion on the edges. The number of rounds is proportional to the depth of the private trees.

Our secure compiler works round by round, where all nodes in the graph apply the PSM protocol for every round of the original algorithm $\mathcal{A}$. After securely simulating all the rounds of $\mathcal{A}$, each node holds an encrypted version of the last state, which contains nothing but the desired output. The neighbors of each node u send it the encryption key (keys used solely for the last encryption), this allows u to decrypt and obtain the final output of the algorithm. As a result, each node u sees only perfectly encrypted states along the whole duration of the algorithm, where it proceeds from one state to the other using the PSM protocol. The security of the PSM protocol ensures that it learns nothing but the next encrypted state. A summary of the algorithm for a single node u is given in Figure 3.

The description of the simulation of algorithm $\mathcal{A}$ with respect to u.

1. For each round $i = 1 \dots r$ do:
   (a) u holds the encrypted state $\widehat{\sigma}_i$.
   (b) A neighbor v of u samples new encryption keys.
   (c) Run a distributed PSM protocol for with server u to compute the next state of u:
      i. u sends its state $\widehat{\sigma}_i$ to a neighbor $v' \neq v$.
      ii. Neighbors use the **private neighborhood trees** to exchange a private random string.
      iii. u learns its new encrypted state $\widehat{\sigma}_{i+1}$.
2. v sends sends the final encryption key to u.
3. Using this key, u computes its final output σ_r.

Figure 3: A schematic overview of the simulated algorithm.

3 Preliminaries and Model

Unless stated otherwise, the logarithms in this paper are base 2. For a distribution $\mathcal{D}$ we denote by $x \leftarrow \mathcal{D}$ an element chosen from $\mathcal{D}$ uniformly at random. For an integer $n \in \mathbb{N}$ we denote by $[n]$ the set $\{1, \ldots, n\}$. We denote by U_n the uniform distribution over n-bit strings. For two distributions (or random variables) X, Y we write $X \equiv Y$ if they are identical distributions. That is, for any x it holds that $\Pr[X = x] = \Pr[Y = x]$.

Graph Notations. For a tree $T \subseteq G$, let $T(z)$ be the subtree of T rooted at z, and let $\pi(u, v, T)$ be the tree path between u and v, when T is clear from the context, we may omit it and simply write $\pi(u, v)$. The fundamental cycle $C_{e,T}$ of an edge $e = (u, v) \notin T$ is the cycle formed by taking e and the tree path between u and v in T_0, i.e., $C_{e,T} = e \circ \pi(u, v, T)$. For $u, v \in G$, let $\text{dist}(u, v, G)$ be the length (in edges) of the shortest $u - v$ path in G. For every integer $i \geq 1$, let $\Gamma_i(u, G) = \{v \mid \text{dist}_G(u, v) \leq i\}$. When $i = 1$, we simply write $\Gamma(u, G)$. Let $\deg(u, G) = |\Gamma(u, G)|$ be the degree of u in G. For a subset of edges $E' \subseteq E(G)$, let $\deg(u, E') = |\{v : (u, v) \in E'\}|$ be the number of edges incident to u in E'. For a subset of nodes U, let $\deg(U, E') = \sum_{u \in U} \deg(u, E')$. For a u_1-u_2 path P_1 and an u_2-u_3 path, the path $P_1 \circ P_2$ is the concatenation of the two paths. For a subset of vertices $S_i \subseteq V(G)$, let $G[S_i]$ be the induced subgraph on S_i.

Fact 1. *[Moore Bound, [Bol04]] Every n-vertex graph $G = (V, E)$ with at least $2n^{1+1/k}$ edges has a cycle of length at most $2k$.*

3.1 Distributed Algorithms

The Communication Model. We use a standard message passing model, the CONGEST model [Pel00], where the execution proceeds in synchronous rounds and in each round, each node can send a message of size $O(\log n)$ to each of its neighbors. In this model, local computation is done for free at each node and the primary complexity measure is the number of communication rounds. Each node holds a processor with a unique and arbitrary ID of $O(\log n)$ bits. Throughout, we make an extensive use of the following useful tool, which is based on the random delay approach of [LMR94].

Theorem 5 ([Gha15, Theorem 1.3])**.** *Let G be a graph and let $A_1, \ldots, A_m$ be m distributed algorithms in the CONGEST model, where each algorithm takes at most $\mathtt{d}$ rounds, and where for each edge of G, at most $\mathtt{c}$ messages need to go through it, in total over all these algorithms. Then, there is a randomized distributed algorithm (using only private randomness) that, with high probability, produces a schedule that runs all the algorithms in $O(\mathtt{c} + \mathtt{d} \cdot \log n)$ rounds, after $O(\mathtt{d} \log^2 n)$ rounds of pre-computation.*

A Distributed Algorithm. Consider an n-vertex graph G with maximal degree Δ. We model a distributed algorithm A that works in r rounds as describing r functions $f_1, \ldots, f_r$ as follows. Let u be a node in the graph with input x_u and neighbors $v_1, \ldots, v_\Delta$. At any round i, the memory of a node u consists of a state, denoted by σ_i and Δ messages $m_{v_1 \to u} \ldots, m_{v_\Delta \to u}$ that were received in the previous round.

Initially, we set σ_0 to contained only the input x_u of u and its ID and initialize all messages to $\perp$. At round i the node u updates its state to σ_{i+1} according to its previous state σ_i and the message from the previous round, and prepares Δ messages to send $m_{u \to v_1}, \ldots, m_{u \to v_\Delta}$. To ease notation (and without loss of generality) we assume that each state contains the ID of the node u,

thus, we can focus on a single update function f_i for every round that works for all nodes. The function f_i gets the state σ_i and messages $m_{v_1 \to u} \ldots, m_{v_\Delta \to u}$, and randomness s_i and outputs the next state and outgoing message:

$$(\sigma_i, m_{u \to v_1}, \ldots, m_{u \to v_\Delta}) \leftarrow f(\sigma_{i-1}, m_{v_1 \to u}, \ldots, m_{v_\Delta \to u}, s_i).$$

At the end of the r rounds, each node u has a state σ_r and a final output of the algorithm. Without loss of generality, we assume that σ_r is the final output of the algorithm (we can always modify f_r accordingly).

Natural Distributed Algorithms. We define a family of distributed algorithms which we call *natural*, which captures almost all known distributed algorithms. A natural distributed algorithm has two restrictions for any round i: (1) the size the state is bounded by $|\sigma_i| \leq \Delta \cdot \mathsf{polylog}(n)$, and (2) the function f_i is computable in polynomial time. The input for f_i is the state σ_i and at most Δ message each of length $\log n$. Thus, the input length m for f_i is bounded by $m \leq \Delta \cdot \mathsf{polylog}(n)$, and the running time should be polynomial in this input length.

We introduce this family of algorithms mainly for simplifying the presentation of our main result. For these algorithms, our main statement can be described with minimal overhead. However, our results are general and work for any algorithm, with appropriate dependency on the size of the state and the running time the function f_i (i.e., the internal computation time at each node u in round i).

Notations. We introduce some notations: For an algorithm $\mathcal{A}$, graph G, input $X = \{x_v\}_{v \in G}$ we denote by $\mathcal{A}_u(G, X)$ the random variable of the output of node u while performing algorithm $\mathcal{A}$ on the graph G with inputs X (recall that $\mathcal{A}$ might be randomized and thus the output is a random variable and not a value). Denote by $\mathcal{A}(G, X) = \{\mathcal{A}_u(G, X)\}_{u \in G}$ the collection of outputs (in some canonical ordering). Let $\mathsf{View}_u^{\mathcal{A}}(G, X)$ be a random variable of the viewpoint of u in the running of the algorithm $\mathcal{A}$. This includes messages sent to u, its memory and random coins during all rounds of the algorithm.

Secure Distributed Computation. Let $\mathcal{A}$ be a distributed algorithm. Informally, we say that $\mathcal{A}'$ computes $\mathcal{A}$ (or simulates $\mathcal{A}$) in a secure manner of $\mathcal{A}$ if when running the algorithm $\mathcal{A}'$ every node u learns the final output σ_r of $\mathcal{A}$ but "nothing more". This notion is captured by the existence of a simulator and is defined below.

Definition 1 (Perfect Privacy)**.** *Let $\mathcal{A}$ be a distributed (possibly randomized) algorithm, that works in r rounds. We say that an algorithm $\mathcal{A}'$ computes $\mathcal{A}$ with perfect privacy if for every graph G, every $u \in G$ and it holds that:*

1. ***Correctness:*** *For every input $X = \{x_v\}_{v \in V}$: $\mathcal{A}(G, X) \equiv \mathcal{A}'(G, X)$.*
2. ***Perfect Privacy:*** *There exists a randomized algorithm (simulator) Sim such that for every input $X = \{x_v\}_{v \in V}$ it holds that*

$$\mathsf{View}_u^{\mathcal{A}'}(G, X) \equiv \mathsf{Sim}(G, x_u, \mathcal{A}_u(G, X)).$$

This security definition is known as the "semi-honest" model, where the adversary, acting a one of the nodes in the graph, is not allowed to deviate from the prescribed protocol, but can run arbitrary computation given all the messages it received. Moreover, we assume that the adversary does no collude with other nodes in the graph. It is possible to extend the security notion to include these notions, more on this is discussed at Section 7.

3.2 Cryptography with Perfect Privacy

One of the main cryptographic tools we use is a specific protocol for secure multiparty computation that has perfect privacy. Feige Kilian and Naor [FKN94] suggested a model where two players having inputs x and y wish to compute a function $f(x, y)$ in a secure manner. They achieve this by each sending a single message to a third party that is able to compute the output of the function f from these messages, but learn nothing else about the inputs x and y. For the protocol to work, the two parties need to share private randomness that is not known to the third party. This model was later generalized to multi-players and is called the Private Simultaneous Messages Model [IK97], which we formally describe next.

Definition 2 (The PSM model)**.** *Let* $f\colon (\{0,1\}^m)^k \to \{0,1\}^m$ *be a* k *variant function. A* PSM *protocol for* f *consists of a pair of algorithms* $(\mathsf{PSM.Enc}, \mathsf{PSM.Dec})$ *where* $\mathsf{PSM.Enc}\colon \{0,1\}^m \times \{0,1\}^r \to \{0,1\}^t$ *and* $\mathsf{PSM.Dec}\colon (\{0,1\}^t)^k \to \{0,1\}^m$ *such that*

- *For any* $X = (x_1, \ldots, x_k)$ *it holds that:*

$$\Pr_{R\in\{0,1\}^r}[\mathsf{PSM.Dec}(\mathsf{PSM.Enc}(x_1, R), \ldots, \mathsf{PSM.Enc}(x_k, R)) = f(x_1, \ldots, x_k)] = 1.$$

- *There exists a randomized algorithm (simulator)* Sim *such that for* $X = x_1, \ldots, x_k$ *and for* R *sampled from* $\{0,1\}^r$*, it holds that*

$$\{\mathsf{PSM.Enc}(x_i, R)\}_{i\in[k]} \equiv \mathsf{Sim}(f(x_1, \ldots, x_k)).$$

The communication complexity of the PSM protocol is the encoding length t *and the randomness complexity of the protocol is defined to be* $|R| = r$*.*

Theorem 6 (Follows from [IK97])**.** *For every function* $f : (\{0,1\}^m)^k \to \{0,1\}^\ell$ *that is computable by an* $s = s(m, k)$*-space TM there is an efficient perfectly secure* PSM *protocol whose communication complexity and randomness complexity are* $O(km\ell \cdot 2^{2s})$*.*

We describe two additional tools that we will use, secret sharing and one-time-pad encryption.

Definition 3 (Secret Sharing)**.** *Let* $x \in \{0,1\}^n$ *be a message. We say* x *is* secret shared *to* k *shares by choosing* k *random strings* $x^1, \ldots, x^k \in \{0,1\}^n$ *conditioned on* $x = \bigoplus_{j=1}^k x^j$*. Each* x^j *is called a share, and notice that the joint distribution of any* $k-1$ *shares is uniform over* $(\{0,1\}^n)^{k-1}$*.*

Definition 4 (One-Time-Pad Encryption)**.** *Let* $x \in \{0,1\}^n$ *be a message. A one-time pad is an extremely simple encryption scheme that has information theoretic security. For a random key* $K \in \{0,1\}^n$ *the "encryption" of* x *according to* K *is* $\widehat{x} = x \oplus K$*. It is easy to see that the encrypted message* $\widehat{x}$ *(without the key) is distributed as a uniform random string. To decrypt* $\widehat{x}$ *using the key* K *we simply compute* $x = \widehat{x} \oplus K$*. The key* K *might be references as the encryption key or decryption key.*

Paper Organization. In Section 4 we describe the centralized constructions of our low-congestion covers. We start by showing the construction of cycle covers (in Section 4.1). We then use the cycle cover construction to compute private neighborhood trees in Section 4.2. Section 5 describes the secure simulation which generalizes PSM to general graphs. Finally, Section 6 considers the distributed construction of our low congestion covers.

4 Low-Congestion Covers

4.1 Cycle Cover

We give the formal definition of a cycle cover and prove our main theorem regarding low-congestion cycle covers. Intuitively, a cycle cover is a collection is cycles in the graph such that each edge is covered by at least one cycle from the collection. We care about two main parameters regarding the cycle cover that we wish to minimize: (1) cycle length: the maximal length of a cycle and (2) edge congestion: the maximal number of cycles an edge participates in.

Definition 5 (Low-Congestion Cycle Cover)**.** *For a given graph $G = (V, E)$, a $(\mathtt{d}, \mathtt{c})$ low-congestion cycle cover $\mathcal{C}$ of G is a collection of cycles that cover all edges of G such that each cycle $C \in \mathcal{C}$ is of length at most $O(\mathtt{d})$ and each edge appears in at most $O(\mathtt{c})$ cycles in $\mathcal{C}$. That is, for every $e \in E$ it holds that $1 \leq |\{C \in \mathcal{C} : e \in C\}| \leq O(\mathtt{c})$.*

We also consider partial covers, that cover only a subset of edges E'. We say that a cycle cover $\mathcal{C}$ is a $(\mathtt{d}, \mathtt{c})$ cycle cover for $E' \subseteq E$, if all cycles are of length at most D, each edge of E' appears in at least one of the cycles of $\mathcal{C}$, and no edge in $E(G)$ appears in more than c cycles in $\mathcal{C}$. That is, in this restricted definition, the covering is with respect to the subset of edges E', however, the congestion limitation is with respect to all graph edges.

The main contribution of this section is an existential result regarding cycle covers with low congestion. Namely, we show that *any graph* that is 2-edge connected has a cycle cover where each cycle is at most the diameter of the graph (up to $\log n$ factors) and each edge is covered by $O(\log n)$ cycles. Moreover, the proof is actually constructive, and yields a polynomial time algorithm that computes such a cycle cover.

Theorem 1. *For every n-vertex graph G with diameter D that is 2-edge connected, there exists a $(\mathtt{d}, \mathtt{c})$-cycle cover with $\mathtt{d} = O(D \log n)$ and $\mathtt{c} = O(\log^3 n)$.*

The construction of a $(\mathtt{d}, \mathtt{c})$-cycle cover $\mathcal{C}$ starts by constructing a BFS tree T. The algorithm has two sub-procedures: the first computes a cycle collection $\mathcal{C}_1$ for covering the *non-tree* edges $E_1 = E(G) \setminus E(T)$, the second computes a cycle collection $\mathcal{C}_2$ for covering the *tree* edges $E_2 = E(T)$. We describe each cover separately. The pseudo-code for the algorithm is given in Figure 4. The algorithm uses two procedures, NonTreeCover and TreeCover which are given in Section 4.1.1 and Section 4.1.2 respectively.

4.1.1 Covering Non-Tree Edges

Covering the non-tree edge mainly uses the fact that while the graph many edges, then the girth is small. Specifically, using Fact 1, with $k = \log n$ we get that the girth of a graph with at least $2n$ edges is at most $2 \log n$. Hence, as long as that the graph has at least $2n$ edges, a cycle of length $2 \log n$ can be found. We get that all but $2n$ edges in G are covered by edge-disjoint cycles of length $2 \log n$.

In this subsection, we show that the set of edges E_1, i.e., the set of non-tree edges can be covered by a $(D \log n, O(1))$-cycle cover denoted $\mathcal{C}_1$. Actually, what we show is slightly more general: if the tree is of depth $D(T)$ the length of the cycles is at most $O(D(T) \log n)$. Lemma 1 will be useful for covering the tree-edges as well and is used again in see next subsection (Section 4.1.2).

Algorithm CycleCover$(G = (V, E))$

1. Construct a BFS tree T of G (with respect to edge set E).
2. Let $E_1 = E(G) \setminus E(T)$ be all non-tree edges, and let $E_2 = E(T)$ be all tree edges.
3. $\mathcal{C}_1 \leftarrow$ NonTreeCover(T, E_1).
4. $\mathcal{C}_2 \leftarrow$ TreeCover(T, E_2)
5. Output $\mathcal{C}_1 \cup \mathcal{C}_2$.

Figure 4: Centralized algorithm for finding a cycle cover of a graph G.

Lemma 1. *Let $G = (V, E)$ be a n-vertex graph, let $T \subseteq G$ be a tree of depth $D(T)$. Then, there exists a $(D(T) \log n, \log n)$-cycle cover $\mathcal{C}_1$ for the edges of $E(G) \setminus E(T)$.*

An additional useful property of the cover $\mathcal{C}_1$ is that despite the fact that the length of the cycles in $\mathcal{C}_1$ is $O(D \log n)$, each cycle is used to cover $O(\log n)$ edges.

Lemma 2. *Each cycle in $\mathcal{C}_1$ is used to cover $O(\log n)$ edges in $E(G) \setminus E(T)$.*

The rest of this subsection is devoted to the proof of Lemma 1. A key component in the proof is a partitioning of the nodes of the tree T into *blocks*. The partitioning is based on a numbering of the nodes from 1 to n and grouping nodes with consecutive numbers into blocks under certain restrictions. We define a numbering of the nodes

$$N : V(T) \to [|V(T)|]$$

by traversing the nodes of the tree in post order. That is, we let $N(u) = i$ if u is the i^{th} node traversed. Using this mapping, we proceed to defining a partitioning of the nodes into blocks and show some of their useful properties.

For a block B of nodes and a subset of non-tree edges $E' \subseteq E_1$, the notation $\deg(B, E')$ is the number of edges in E' that have an endpoint in the set B. We call this the *density* of block B with respect to E'. For a subset of edges E', and a density bound $\mathtt{b}$ (which will be set to a constant), an $(E', \mathtt{b})$-partitioning $\mathcal{B}$ is a partitioning of the nodes of the graph into blocks that satisfies the following properties:

1. Every block consists of a consecutive subset of nodes (w.r.t. their $N(\cdot)$ numbering).
2. If a block B has density $\deg(B, E') > \mathtt{b}$ then B consists of a single node.
3. The total number of blocks is at most $4|E'|/\mathtt{b}$.

Claim 1. *For any $\mathtt{b}$ and E', there exists an $(E', \mathtt{b})$-partitioning partitioning of the nodes of T satisfying the above properties.*

Proof. This partitioning can be constructed by a greedy algorithm that traverses nodes of T in increasing order of their numbering $N(\cdot)$ and groups them into blocks while the density of the

Algorithm Partition(T, E')

1. Let $\mathcal{B}$ be an empty partition, and let B be an empty block.
2. Traverse the nodes of T in post-order, and for each node u do:
   (a) If $\deg(B \cup \{u\}, E') \leq \mathtt{b}$ add u to B.
   (b) Otherwise, add the block B to $\mathcal{B}$ and initialize a new block $B = \{u\}$.
3. Output $\mathcal{B}$.

Figure 5: Partitioning procedure.

block does not exceed $\mathtt{b}$ (see Figure 5 for the precise procedure). Indeed, properties 1 and 2 are satisfied directly by the construction. For property 3, let t be the number of blocks B with $\deg(B, E') \leq \mathtt{b}/2$. By the construction, we know that for any such block B the block B' that comes after B satisfies $\deg(B, E') + \deg(B', E') > \mathtt{b}$. Let $B_1, \ldots, B_\ell$ be the final partitioning. Then, we have t pairs of blocks that have density at least $\mathtt{b}$ and the rest of the $(\ell - t/2)$ blocks that have density at least $\mathtt{b}/2$. Formally, we have

$$\sum_{i=1}^{\ell} \deg(B_i, E') > t\mathtt{b} + (\ell - t/2)\mathtt{b}/2 = \ell\mathtt{b}/2.$$

On the other hand, since it is a partitioning of E' we have that $\sum_{i=1}^{\ell} \deg(B_i, E') = 2|E'|$. Thus, we get that $\ell\mathtt{b}/2 \leq 2|E'|$ and therefore $\ell \leq 4|E'|/\mathtt{b}$ as required. □

Our algorithm for covering the edges of $E_1 = E(G) \setminus E(T)$ makes use of this block partitioning with $\mathtt{b} = 16$. For any two nodes $u, v \in V(T)$, we use the notation $\pi(u, v, T)$ (or simply $\pi(u, v)$) to denote the unique simple path in the tree T from u to v. The algorithm begins with an empty collection $\mathcal{C}$ and then performs $\log n$ iterations where each iteration works as follows: Let $E' \subseteq E_1$ be the set of uncovered edges (initially $E' = E_1$). Then, we partition the nodes of T with respect to E' and density parameter $\mathtt{b}$. Finally, we search for cycles of length at most $\log n$ between the blocks. If such a cycle exists, we map it to a cycle in G by connecting nodes u, v within a block by the path $\pi(u, v)$ in the tree T. This way a cycle of length $\log n$ between the blocks translates to a cycle of length $D(T) \log n$ in the original graph G. Denote the resulting collection by $\mathcal{C}$.

We note that the cycles $\mathcal{C}$ might not be *simple*. This might happen if and only if the tree paths $\pi(v_i, u_{i+1})$ and $\pi(v_j, u_{j+1})$ intersect for some $j \in [t]$. Notice that the if an edge appears more than once in a cycle, then it must be a tree edge. Thus, we can transform any non-simple cycle C into a collection of simple cycles that cover all edges that appeared only once in C (the formal procedure is given at Figure 7). Since these cycle are constructed to cover only non-tree edges, we get that this transformation did not hurt the cover of E_1. The formal description of the algorithm is given in Figure 6.

We move to the analysis of the algorithm, and show that it yields the desired cycle cover. That is, we show three things: that every cycle has length at most $O(D(T) \log n)$, that each edge is covered by at most $O(\log n)$ cycles, and that each edge has at least one cycle covering it.

Algorithm $\mathsf{NonTreeEdgeCover}(T, E_1)$

1. Initialize a cover $\mathcal{C}$ as an empty set.
2. Repeat $\log |E_1|$ times:
   (a) Let $E' \subseteq E_1$ be the subset of all uncovered edges.
   (b) Construct an (E', b)-partitioning $\mathcal{B}$ of the nodes of T.
   (c) While there are t edges $(u_1, v_1), \ldots, (u_t, v_t) \in E'$ for $t \leq \log n$ such that for all $i \in [t-1]$, v_i and u_{i+1} are in the same block and v_t and u_1 are in the same block (with respect to the partitioning $\mathcal{B}$): Add the cycle $u_1 \to v_1 \to \pi(v_1, u_2) \to v_2 \to \pi(v_2, u_3) \to v_3 \to \ldots \to v_t \to u_1$ to $\mathcal{C}$.
3. Compute $\mathcal{C}' \leftarrow \mathsf{SimplifyCycles}(\mathcal{C})$ and output $\mathcal{C}'$.

Figure 6: Procedure for covering non-tree edges.

Algorithm $\mathsf{SimplifyCycles}(\mathcal{C})$

1. While there is a cycle $C \in \mathcal{C}$ with a vertex $w \in C$ that appears more than once:
   (a) Remove C from $\mathcal{C}$.
   (b) Let $C = v_1 \to \cdots \to v_k$ and define $v_{k+i} = v_i$.
   (c) Let $i_1, \ldots, i_\ell$ be such that $v_{i_j} = w$ for all $j \in [\ell]$, and let $i_{\ell+1} = i_1$.
   (d) For all $j \in [\ell]$ let $C_j = v_{i_j} \to v_{i_j+1} \cdots \to v_{i_{j+1}}$, and if $|C_j| \geq 3$, add C_j to $\mathcal{C}$.
2. Output $\mathcal{C}$.

Figure 7: Procedure making all cycles in $\mathcal{C}$ simple.

Cycle Length. The bound of the cycle length follows directly from the construction. The cycles added to the collection are of the form $u_1 \to v_1 \to \pi(v_1, u_2) \to v_2 \to \pi(v_2, u_3) \to v_3 \to \ldots \to v_t \to u_1$, where each $\pi(v_i, u_{i+1})$ are paths in the tree T and thus are of length at most $2D(T)$. Notice that the simplification process of the cycles can only make the cycles shorter. Since $t \leq \log n$ we get that the cycle lengths are bounded by $O(D(T) \log n)$.

Congestion. To bound the congestion of the cycle cover we exploit the structure of the partitioning, and the fact that each block in the partition has a low density. We begin by showing that by the post-order numbering, all nodes in a given subtree have a continuous range of numbers. For every $z \in V(T)$, let $\min_N(z)$ be the minimal number of a node in the subtree of T rooted by z. That is, $\min_N(z) = \min_{u \in T_z} N(u)$ and similarly let $\max_N(z) = \max_{u \in T_z} N(u)$.

Claim 2. *For every $z \in V(T)$ and for every $u \in G$ it holds that (1) $\max_N(z) = N(z)$ and (2) $N(u) \in [\min_N(z), \max_N(z)]$ iff $u \in T_z$.*

Proof. The proof is by induction on the depth of T_z. For the base case, we consider the leaf nodes

z, and hence T_z with 0-depth, the claim holds vacuously. Assume that the claim holds for nodes in level $i+1$ and consider now a node z in level i. Let $v_{i,1}, \ldots, v_{i,\ell}$ be the children of z ordered from left to right. By the post-order traversal, the root $v_{i,j}$ is the last vertex visited in $T_{v_{i,j}}$ and hence $N(v_{i,j}) = \max_N(v_{i,j})$. Since the traversal of $T_{v_{i,j}}$ starts right after finishing the traversal of $T_{v_{i,j-1}}$ for every $j \geq 2$, it holds that $\min_N(v_{i,j}) = N(v_{i,j-1}) + 1$. Using the induction assumption for $v_{i,j}$, we get that all the nodes in $T_z \setminus \{z\}$ have numbering in the range $[\min_N(v_{i,1}), \max_N(v_{i,\ell}]$ and any other node not in T_z is not in this range. Finally, $N(z) = N(v_{i,\ell}) + 1$ and so the claim holds. □

The cycles in the cover we compute are composed of paths $\pi(u, v)$ for two nodes u and v in the same block. Thus, to bound the congestion on an edge $e \in T$ we need to bound the number of blocks such that there contains u, v in that block such that $\pi(u, v)$ passes through e. The next claim shows that every edge in the tree is effected by at most 2 blocks.

Claim 3. *Let $e \in T$ be a tree edge and define $\mathcal{B}(e) = \{B \in \mathcal{B} \mid \exists u, v \in B \text{ s.t. } e \in \pi(u, v)\}$. Then, $|\mathcal{B}(e)| \leq 2$ for every $e \in T$.*

Proof. Let $e = (w, z)$ where w is closer to the root in T, and let u, v be two nodes in the same block B such that $e \in \pi(u, v)$. Let ℓ be the LCA of u and v in T (it might be that $\ell \in \{u, v\}$), then the tree path between u and v can be written as $\pi(u, v) = \pi(u, \ell) \circ \pi(\ell, v)$. Without loss of generality, assume that $e \in \pi(\ell, v)$. This implies that $v \in T_z$ but $u \notin T_z$. Hence, the block of u and v intersects the nodes of T_z. Each block consists of a consecutive set of nodes, and by Claim 2 also T_z consists of a consecutive set of nodes with numbering in the range $[\min_N(z), \max_N(z)]$, thus there are at most two such blocks that intersect $e = (w, z)$, i.e., blocks B that contains both a vertex y with $N(y) \in [\min_N(z), \max_N(z)]$ and a vertex y' with $N(y') \notin [\min_N(z), \max_N(z)]$, and the claim follows. □

Finally, we use the above claims to bound the congestion. Consider any tree edge $e = (w, z)$ where w is closer to the root than z. Let T_z be the subtree of T rooted at z. Fix an iteration i of the algorithm. We characterize all cycles in $\mathcal{C}$ that go through this edge.

For any cycle that passes through e there must be a block B and two nodes $u, v \in B$ such that $e \in \pi(u, v)$. By Claim 3, we know that there are that at each iteration of the algorithm, there are at most two such blocks B that can affect the congestion of e. Moreover, we claim that each such block has density at most $\mathtt{b}$. Otherwise it would be a block containing a single node, say u, and thus the path $\pi(u, u) = u$ is empty and cannot contain the edge e. For each edge in E' we construct a single cycle in $\mathcal{C}$, and thus for each one of the two blocks that affect e the number of pairs u, v such that $e \in \pi(u, v)$ is bounded by $\mathtt{b}/2$ (each pair u, v has two edges in the block B and we know that the total number of edges is bounded by $\mathtt{b}$).

To summarize the above, we get that for each iteration, that are at most 2 blocks that can contribute to the congestion of an edge e: one block that intersects T_z but has also nodes smaller than $\min_N(z)$ and one block that intersects T_z but has also nodes larger than $\max_N(z)$. Each of these two blocks can increase the congestion of e by at most $\mathtt{b}/2$. Since there are at most $\log n$ iterations, we can bound the total congestion by $b \log n$. Notice that if an edge appears k times in a cycle, then this congestion bound counts all k appearances. Thus, after the simplification of the cycles, the congestion remains unchanged.

Cover. We show that each edge in E_1 was covered by some cycle and that each cycle is used to cover $O(\log n)$ edges in E_1. We begin by showing cover for the cycles before the simplification procedure, and then show that the cover remains after this procedure. The idea is that at each iteration of the

algorithm, the number of uncovered edges is reduced by half. Therefore, the $\log |E_1| = O(\log n)$ iterations should suffice for covering all edges of E_1. In each iteration we partition the nodes into blocks, and we search for cycles between the blocks. The point is that if the number of edges is large, then we considering the blocks as nodes in a new virtual graph, this graph has a large number of edges and thus must have a short cycle. At each iteration, the blocks become larger which make the virtual have less nodes and the number of edges is larger relative to the small number of nodes.

In what follows, we formalize the intuition given above. Let E'_i be the set E' at the i^{th} iteration of the algorithm. Consider the iteration i with the set of uncovered edge set E'_i. Our goal is to show that $E'_{i+1} \leq 1/2E'_i$. By having $\log |E_1|$ iterations, last set will be empty.

Let $\mathcal{B}_i$ be the partitioning performed at iteration i with respect to the edge set E'_i. Define a super-graph $\widetilde{G}$ in which each block $B_j \in \mathcal{B}_i$ is represented by a node $\widetilde{v}_j$, and there is an edge $(\widetilde{v}_j, \widetilde{v}_{j'})$ in $\widetilde{G}$ if there is an edge in E'_i between some node u in B_j and a node u' in $B_{j'}$, i.e.,

$$(\widetilde{v}_j, \widetilde{v}_{j'}) \in E(\widetilde{G}) \iff E'_i \cap (B_j \times B_{j'}) \neq \emptyset.$$

See Figure 8 for an illustration. The number of nodes in $\widetilde{G}$, which we denote by n_i, is the number of

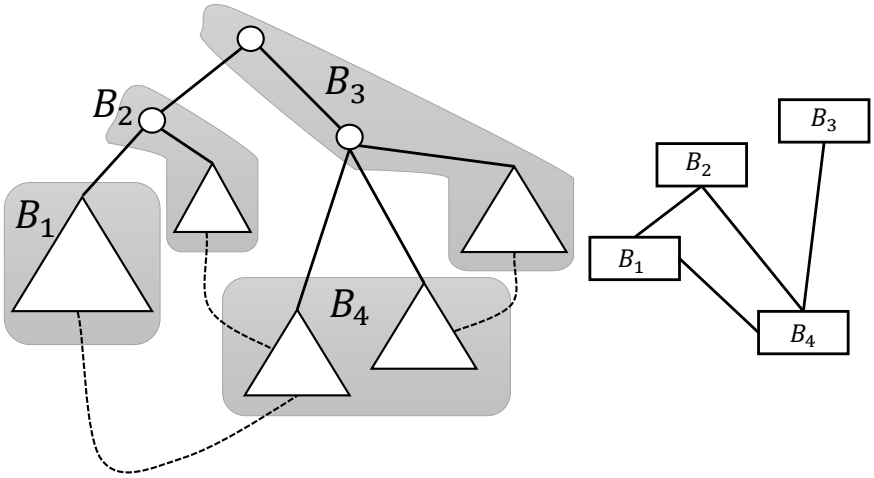

Figure 8: Left: Schematic illustration of the block partitioning in the tree T. Dashed edges are those that remain to be covered after employing Alg. LocalCover, where each two blocks are connected by exactly one edge. Each dashed edge corresponds to super-edges in $\widetilde{G}$. Right: A triangle in the super-graph $\widetilde{G}$.

blocks in the partition and is bounded by $n_i \leq 4|E'_i|/\mathtt{b}$. Let $B(u)$ be the block of the node u. The algorithm finds cycles of the form $u_1 \to v_1 \to \pi(v_1, u_2) \to v_2 \to \pi(v_2, u_3) \to v_3 \to \ldots \to v_t \to u_1$, which is equivalent to finding the cycle $B(u_1), \ldots, B(u_t)$ in the graph $\widetilde{G}$. In general, any cycle of length t in $\widetilde{G}$ is mapped to a cycle in G of length at most $t \cdot D(T)$. Then, the algorithm adds the cycle to $\mathcal{C}$ and removes the edges of the cycle (thus removing them also from $\widetilde{G}$). At the end of iteration i the graph $\widetilde{G}$ has no cycles of length at most $\log n$. At this point, the next set of edges E'_{i+1} is exactly the edges left in $\widetilde{G}$. By Fact 1 (and recalling that $\mathtt{b} = 16$) we get that if $\widetilde{G}$ does not have any cycles of length at most $\log n$ then we get the following bound on the number of edges:

$$E'_{i+1} \leq 2n_i = 8|E'_i|/\mathtt{b} = |E'_i|/2.$$

Thus, all will be covered by a cycle C before the simplification process. We show that the simplification procedure of the cycle maintains the cover requirement. This stems from the fact that any edge that appears more than once in a cycle, might be dropped but is not the edge for which this cycle was constructed to cover. That is, for each non-tree edge, we construct a cycle that covers it. By the construction of the cycles, if a cycle is not simple it must be because of a tree edge. It is left to show that this process only drops edges that appear more than once:

Claim 4. *Let C be a cycle and let $\mathcal{C}' \leftarrow \mathsf{SimplifyCycles}(C)$. Then, for every edge $e \in C$ that appears at most once in C there is a cycle $C' \in \mathcal{C}'$ such that $e \in C'$.*

Proof. The procedure SimplifyCycles works in iterations where in each iteration it chooses a vertex w that appears more than once in C and partitions the cycle C to consecutive parts, $C_1, \ldots, C_\ell$. All edges in C appear in some C_j. However, C_j might not be a proper cycle since it might be the case that $|C_j| \leq 2$. Thus, we show that in an edge $e \in C$ appeared at most once in a cycle C then it will appear in C_j for some j where $|C_j| \geq 3$. We show that this holds for any iteration and thus will hold at the end of the process.

We assume without loss of generality that no vertex has two consecutive appearances. Denote $e = (v_2, v_3)$ and let $C = v_1 \to v_2 \to v_3 \to \ldots v_k$ for $k \geq 3$. Since e does not appear again in C we know that v_1, v_2, v_3 are distinct. Thus, if $k = 3$ then C will not be split again and the claim follows.

Therefore, assume that $k \geq 4$. Since e does not appear again in C we know that v_2, v_3, v_4 are distinct (it might be the case that $v_1 = v_4$). Thus, we know that $|\{v_1, v_2, v_3, v_4\}| \geq 3$. Any subsequence begins and ends at the same vertex and thus the subsequence C_j that contains e must contains all of v_1, v_2, v_3, v_3 and thus $|C_j| \geq 3$, and the claim follows. □

Finally, we turn to prove Lemma 2. The lemma follows by noting that each cycle in $\mathcal{C}_1$ contains at most $O(\log n)$ non-tree edges. To see this, observe that each cycle computed in the contracted block graph has length $O(\log n)$. Translating these cycles into cycles in G introduces only *tree* edges. We therefore have that each cycle is used to cover $O(\log n)$ non-tree edges.

4.1.2 Covering Tree Edges

Finally, we present Algorithm TreeCover that computes a cycle cover for the tree edges. The algorithm is recursive and uses Algorithm NonTreeCover as a black-box. Formally, we show:

Lemma 3. *For every n-vertex graph G and a tree $T \subseteq G$ of depth D, there exists a $(D \log n, \log^3 n)$ cycle cover $\mathcal{C}_3$ for the edges of T.*

We begin with some notation. Throughout, when referring to a tree edge $(u, v) \in T$, the node u is closer to the root of T than v. Let $E(T) = \{e_1, \ldots, e_{n-1}\}$ be an ordering of the edges of T in non-decreasing distance from the root. For every tree edge $e \in T$, define the *swap* edge of e by $e' = \mathsf{Swap}(e)$ to be an arbitrary edge in G that restores the connectivity of $T \setminus \{e\}$. Since the graph G is 2-edge connected such an edge $\mathsf{Swap}(e)$ is guaranteed to exist for every $e \in T$. Let $e = (u, v)$ and $(u', v') = \mathsf{Swap}(e)$, we denote u by $p(v)$ (since u is the parent of v in the tree) and v' by $s(v)$, where $s(v)$ is the endpoint of $\mathsf{Swap}(e)$ that do not belong to $T(u')$ (i.e., the subtree T rooted at u). Define the v-$s(v)$ path

$$P_e = \pi(v, u') \circ \mathsf{Swap}(e).$$

For an illustration see Figure 9.

For the tree T, we construct a subset of tree edges denoted by $I(T)$ that we are able to cover. These edges are *independent* in the sense that their P_e paths are "almost" edge disjoint (as will be shown next). The subset $I(T)$ is constructed by going through the edges of T in non-decreasing distance from the root. At any point, we add e to $I(T)$ only if it is not covered by the $P_{e'}$ paths of the e' edges already added.

Claim 5. *The subset $I(T)$ satisfies the following properties:*

- *For every $e \in E(T)$, there exists $e' \in I(T)$ such that $e \in e' \circ P_{e'}$.*
- *For every $e, e' \in I(T)$ such that $e \neq e'$ it holds that P_e and $P_{e'}$ have no tree edge in common (no edge of T is in both paths).*
- *For every swap edge (z, w), there exists at most two paths $P_e, P_{e'}$ for $e, e' \in I(T)$ such that one passes through (z, w) and the other through (w, z). That is, each swap edge appears at most twice on the P_e paths, once in each direction.*

Proof. The first property follows directly from the construction. Next, we show that they share no tree edge in common. Assume that there is a common edge $(z, w) \in P_e \cap P'_e \cap E(T)$. Then, both e, e' must be on the path from root to z on the tree and hence $e' \in P_e$, leading to contradiction. For the third property, assume towards constriction that both P_e and $P_{e'}$ use the same swap edge in the same direction. Again it implies that both e, e' are on the path from the root to z on T. □

Our cycle cover for the $I(T)$ edges will be shown to cover all the edges of the tree T. This is because the cycle that we construct to cover an edge $e \in I(T)$ necessarily contains P_e.

Algorithm TreeCover uses the following procedure TreeEdgeDisjointPath, usually used in the context of distributed routing.

Key Tool: Route Disjoint Matching. Algorithm TreeEdgeDisjointPath solves the following problem defined by Peleg (see Lemma 4.3.2 [Pel00]): given a rooted tree T and a set of $2k$ marked nodes $M \subseteq V(T)$ for $k \leq n/2$, the goal is to find (by a distributed algorithm) a matching of these vertices $\langle w_i, w_j \rangle$ into pairs such that the tree paths $\pi(w_i, w_j, T)$ connecting the matched pairs are *edge-disjoint*. This matching can be computed distributively in $O(\mathsf{Diam}(T))$ rounds by working from the leaf nodes towards the root. In each round a node u that received information on more $\ell \geq 2$ unmarked nodes in its subtree, match all but at most one into pairs and upcast to its parent the ID of at most one unmarked node in its subtree. It is easy to see that all tree paths between matched nodes are indeed edge disjoint.

We are now ready to explain the cycle cover construction of the tree edges $E(T)$.

Description of Algorithm TreeCover. We restrict attention for covering the edges of $I(T)$. The tree edges $I(T)$ will be covered in a specific manner that covers also the edges of $E(T) \setminus I(T)$. The key idea is to define a collection of (virtual) non-tree edges $\widetilde{E} = \{(v, s(v)) : (p(v), v) \in I(T)\}$ and covering these non-tree edges by enforcing the cycle that covers the non-tree edge $(v, s(v))$ to covers the edges $e = (p(v), v)$ as well as the path P_e. Since every edge $e' \in T$ appears on one of the $e \circ P_e$ paths, this will guarantee that all tree edges are covered.

Algorithm TreeCover is recursive and has $O(\log n)$ levels of recursion. In each independent level of the recursion we need to solve the following sub-problem: Given a tree T', cover by cycles the edges of $I(T')$ along with their P_e paths. The key idea is to subdivide this problem into two

independent and balanced subproblems. To do that, the tree T' gets partitioned[7] into two balanced edge disjoint subtrees T'_1 and T'_2, where $|T'_1|, |T'_2| \leq 2/3 \cdot |T'|$ and $E(T'_1) \cup E(T'_2) = E(T')$. Some of the tree edges in T' are covered by applying a procedure that computes cycles using the edges of T', and the remaining ones will be covered recursively in either T'_1 or T'_2. Specifically, the edges of $I(T')$ are partitioned into 4 types depending on the position of their swap edges. Let

$$E'_{x,y} = \{(u,v) \in E(T'_x) \cap I(T') \mid v \in V(T'_x) \text{ and } s(v) \in V(T'_y) \setminus V(T'_x)\}, \text{ for every } x, y \in \{1,2\}.$$

The algorithm computes a cycle cover $\mathcal{C}_{1,2}$ (resp., $\mathcal{C}_{2,1}$) for covering the edges of $E'_{1,2}, E'_{2,1}$ respectively. The remaining edges E'_{11} and E'_{22} are covered recursively by applying the algorithm on T'_1 and T'_2 respectively. See Fig. 9 for an illustration.

We now describe how to compute the cycle cover $\mathcal{C}_{1,2}$ for the edges of $E'_{1,2}$. The edges $E'_{2,1}$ are covered analogously (i.e., by switching the roles of T'_1 and T'_2). Recall that the tree edges $E'_{1,2}$ are those edges $(p(v), v)$ such $v \in T'_1$ and $s(v) \in T'_2$. The procedure works in $O(\log n)$ phases, each phase i computes three cycle collections $\mathcal{C}'_{i,1}, \mathcal{C}'_{i,2}$ and $\mathcal{C}'_{i,3}$ which together covers at least half of the yet uncovered edges of $E'_{1,2}$ (as will be shown in analysis).

Consider the i^{th} phase where we are given the set of yet uncovered edges $X_i \subseteq E'_{1,2}$. We first mark all the vertices v with $(p(v), v) \in X_i$. Let M_i be this set of marked nodes. For ease of description, assume that M_i is even, otherwise, we omit one of the marked vertices w (from M_i) and take care of its edge $(p(w), w)$ in later phases. We apply Algorithm $\mathsf{TreeEdgeDisjointPath}(T'_1, M_i)$ (see Lemma 4.3.2 [Pel00]) which matches the marked vertices M_i into pairs $\Sigma = \{\langle v_1, v_2\rangle \mid v_1, v_2 \in M_i\}$ such that for each pair $\sigma = \langle v_1, v_2\rangle$ there is a tree path $\pi(\sigma) = \pi(v_1, v_2, T'_1)$ and all the tree paths $\pi(\sigma), \pi(\sigma')$ are edge disjoint for every $\sigma, \sigma' \in \Sigma$.

Let $X''_i = \{e = (p(v), v) \in X_i : \exists v' \text{ and } \langle v, v'\rangle \in \Sigma, \text{ s.t. } e \in \pi(v, v', T'_1)\}$ be the set of edges in X_i that appear on the collection of edge disjoint paths $\{\pi(\sigma), \sigma \in \Sigma\}$. Our goal is to cover all edges in $E''_i = X''_i \cup \{P_e \mid e \in X''_i\}$ by cycles $\mathcal{C}_i$. To make sure that all edges E''_i are covered, we have to be careful that each such edge appears on a given cycle exactly once. Towards this end, we define a directed conflict graph G_Σ whose vertex set are the pairs of Σ, and there is an arc $(\sigma', \sigma) \in A(G_\Sigma)$ where $\sigma = \langle v_1, v_2\rangle$, $\sigma = \langle v'_1, v'_2\rangle$, if at least one of the following cases holds: Case (I) $e = (p(v_1), v_1)$ on $\pi(v_1, v_2, T'_1)$ and the path $\pi' = \pi(v'_1, v'_2, T'_1)$ intersects the edges of P_e; Case (II) $e' = (p(v_2), v_2)$ on $\pi(v_1, v_2, T'_1)$ and the path π' intersects the edges of $P_{e'}$. Intuitively, a cycle that contains both π' and P_e is not simple and in particular might not cover all edges on P_e. Since the goal of the pair $\sigma = \langle v_1, v_2\rangle$ is to cover all edges on P_e (for $e \in \pi(v_1, v_2, T'_1)$), the pair σ' "interferes" with σ.

In the analysis section (Claim 6), we show that the outdegree in the graph G_Σ is bounded by 1 and hence we can color G_Σ with 3 colors. This allows us to partition Σ into three color classes Σ_1, Σ_2 and Σ_3. Each color class Σ_j is an independent set in G_Σ and thus it is "safe" to cover all these pairs by cycles together. We then compute a cycle cover $\mathcal{C}_{i,j}$, for each $j \in \{1,2,3\}$. The collection of all these cycles will be shown to cover the edges E''_i.

To compute $\mathcal{C}_{i,j}$ for $j = \{1,2,3\}$, for each matched pair $\langle v_1, v_2\rangle \in \Sigma_j$, we add to T'_2 a virtual edge $\widehat{e}$ between $s(v_1)$ and $s(v_2)$. Let

$$\widehat{E}_{i,j} = \{(s(v_1), s(v_2)) \mid \langle v_1, v_2\rangle \in \Sigma_j\}.$$

We cover these virtual non-tree edges by cycles using Algorithm $\mathsf{NonTreeCover}$ on the tree T'_2 with the non-tree edges $\widehat{E}_{i,j}$. Let $\mathcal{C}''_{i,j}$ be the output $O(D \log n, \log n)$ cycle cover of Algorithm

[7] This partitioning procedure is described in Appendix A. We note that this partitioning maintains the layering structure of T'.

NonTreeCover$(T_2', E'_{i,j})$. The output cycles of $\mathcal{C}''_{i,j}$ are not yet cycles in G as they consists of two types of virtual edges: the edges in $\widehat{E}_{i,j}$ and the edges $\widetilde{E} = \{(v, s(v)) \mid (p(v), v) \in I(T)\}$. First, we translate each cycle $C'' \in \mathcal{C}''_{i,j}$ into a cycle C' in $G \cup \widetilde{E}$ by replacing each of the virtual edges $\widehat{e} = (s(v_1), s(v_2)) \in \widehat{E}_{i,j}$ in C'' with the path $P(\widehat{e}) = (s(v_1), v_1) \circ \pi(v_1, v_2, T_1') \circ (v_2, s(v_2))$. Then, we replace each virtual edge $(v, s(v)) \in \widetilde{E}$ in C' by the v-$s(v)$ path P_e for $e = (p(v), v)$. This results in cycles $\mathcal{C}_{i,j}$ in G.

Finally, let $\mathcal{C}_i = \mathcal{C}_{i,1} \cup \mathcal{C}_{i,2} \cup \mathcal{C}_{i,3}$ and define $X_{i+1} = X_i \setminus X''_i$ to be the set of edges $e \in X_i$ that are not covered by the paths of Σ. If in the last phase $\ell = O(\log n)$, the set of marked nodes M_ℓ is odd, we omit one of the marked nodes $w \in M_\ell$, and cover its tree edge $e = (p(w), w)$ by taking the fundamental cycle of the swap edge Swap(e) into the cycle collection. The final cycle collection for $E'_{1,2}$ is given by $\mathcal{C}_{1,2} = \bigcup_{i=1}^{\ell} \mathcal{C}_i$. The same is done for the edges $E'_{2,1}$. This completes the description of the algorithm. The final collection of cycles is denoted by $\mathcal{C}_3$. See Figure 12 for the full description of the algorithm. See Figures 9 to 11 and for illustration.

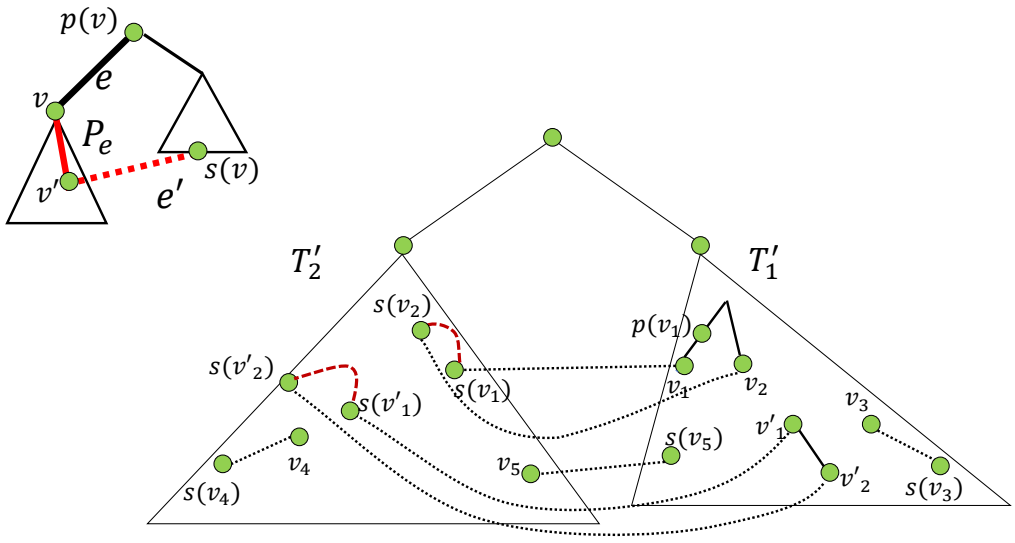

Figure 9: Left: Illustration the swap edge $e' =$ Swap(e) and the path P_e for an edge $e \in T$. For each tree edge $e = (u, v) \in T$, we add the auxiliary edge $(v, s(v))$. Right: The tree T' is partitioned into two balanced trees T'_1 and T'_2. The root vertex in this example belongs to both trees. The edges $\widetilde{E}$ are partitioned into four sets: $E'_{1,1}$ (e.g., the edge $(p(v_3), v_3)$), $E'_{2,2}$ (e.g., the edge $(p(v_4), v_4)$), $E'_{1,2}$ (e.g., the edge $(p(v_1), v_1)$), $E'_{2,1}$ (e.g., the edge $(p(v_5), v_5)$). The algorithm covers the edges of $E'_{1,2}$ by using Algorithm TreeEdgeDisjointPath to compute a matching and edge disjoint paths in T'_1. See the tree paths between v_1 and v_2 and v'_1 and v'_2. Based on this matching, we add virtual edges between vertices of T'_2, for example the edges $(s(v_1), s(v_2))$ and $(s(v'_1), s(v'_2))$ shown in dashed. The algorithm then applies Algorithm NonTreeCover to cover these non-tree edges in T'_2.

We analyze the TreeCover algorithm and show that it finds short cycles, with low congestion and that every edge of T is covered.

Short Cycles. By construction, each cycle that we compute using Algorithm NonTreeCover con-

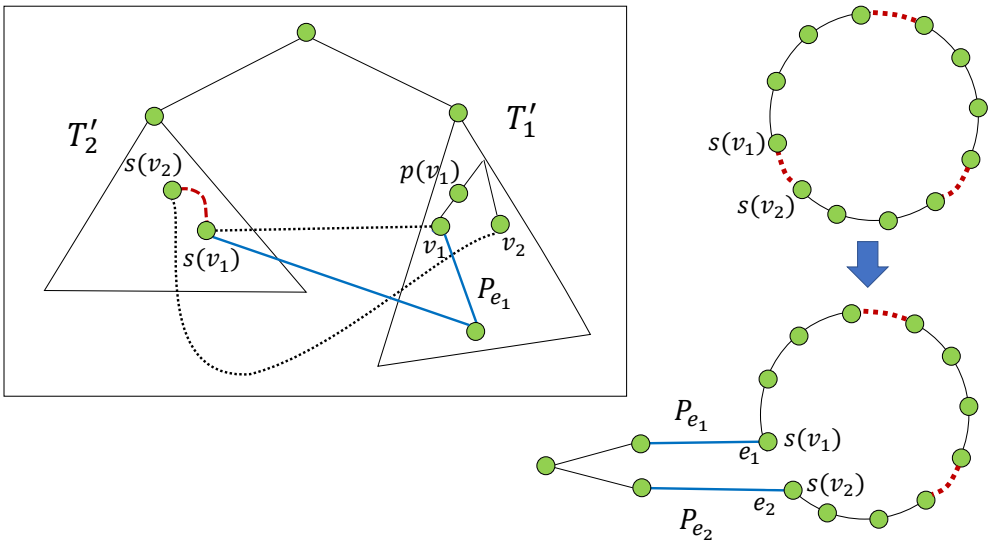

Figure 10: Illustration of replacing a single virtual edge $(s(v_1), s(v_2))$ in a cycle $C'' \in \mathcal{C}''_{i,j}$ by an $s(v_1)$-$s(v_2)$ path in G.

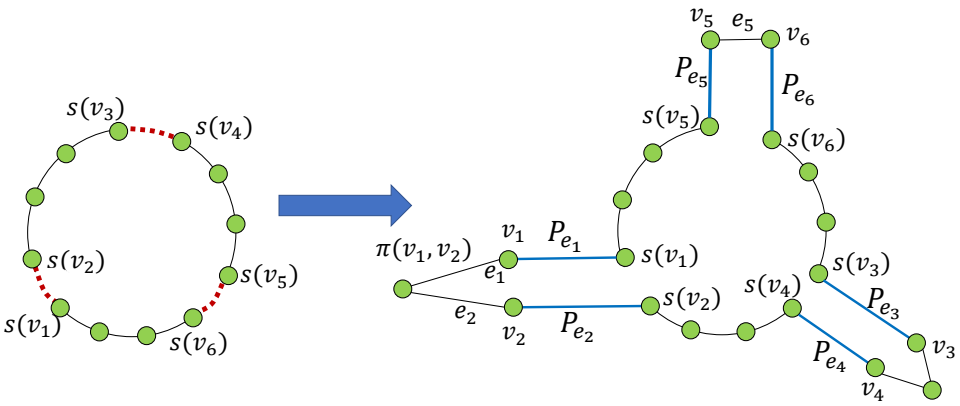

Figure 11: Translating virtual cycles into cycles in G. Each cycle contains $O(\log n)$ virtual edges which are are replaced by (almost) edge disjoint paths in G. Note that each edge on $e_i \circ P_{e_i}$ appears exactly once since the P_e paths are tree-edge disjoint and the pairs $\sigma = \langle u_j, v_j \rangle$ in $\Sigma_{i,j}$ do not interfere with each other.

sists of at most $O(\log n)$ non-tree edges $\widehat{E}_{i,j}$. The algorithm replaces each non-tree edge $\widehat{e} = (v_1, v_2)$ by an v_1-v_2 path in G of length $O(D)$. This is done in two steps. First, $\widehat{e} = (v_1, v_2)$ is replaced by

Algorithm $\mathsf{TreeCover}(T')$

1. If $|T'| = 1$ then output empty collection.
2. Let $\mathcal{C}$ be an empty collection.
3. Partition T' into balanced $T'_1 \cup T'_2$.
4. Let E' be an empty set.
5. For every $(u, v) \in T$ let $(u', v') = \mathsf{Swap}((u, v))$ and add a virtual edge (v, v') to E'.
6. For $i = 1, ..., O(\log n)$:
   (a) Let M_i be all active nodes $v \in V(T'_1)$ s.t. $\mathsf{Swap}(v) \in V(T_2)\}$.
   (b) Apply $\mathsf{TreeEdgeDisjointPath}(T'_1, M_i)$ and let $\Sigma = \{\langle v_1, v_2\rangle\}$ be the collection of matched pairs.
   (c) Partition Σ into 3 *independent* sets Σ_1, Σ_2 and Σ_3.
   (d) For every $j \in \{1, 2, 3\}$ compute a cycle cover $\mathcal{C}_{i,j}$ as follows:
      i. For every pair $\langle v_1, v_2\rangle$ in Σ_j add a virtual edge $(s(v_1), s(v_2))$ to $\widehat{E}_{i,j}$.
      ii. $\mathcal{C}''_{i,j} \leftarrow \mathcal{C}''_{i,j} \cup \mathsf{NonTreeCover}(T_2, \widehat{E}_{i,j})$.
      iii. Translate $\mathcal{C}''_{i,j}$ to cycles $\mathcal{C}_{i,j}$ in G.
   (e) Let $\mathcal{C}_i = \mathcal{C}_{i,1} \cup \mathcal{C}_{i,2} \cup \mathcal{C}_{i,3}$.
7. $\mathcal{C}_1 = \bigcup_i \mathcal{C}_i$.
8. Repeat where T'_1 and T'_2 are switched.
9. Add to $\mathcal{C}$ the output of $\mathsf{TreeCover}(T'_1) \cup \mathsf{TreeCover}(T'_2)$.
10. Output $\mathsf{SimplifyCycles}(\mathcal{C} \cup \mathcal{C}_1)$.

Figure 12: Procedure for covering tree edges.

a path $P_{\widehat{e}} = (v_1, s(v_1)) \circ \pi(v_1, v_2, T) \circ (v_2, s(v_2))$ in $G \cup \widetilde{E}$. Then, each $(v, s(v))$ edge is replaced by the path $P_{(p(v),v)}$ in G, which is also of length $O(D)$. Hence, overall the translated path v_1-v_2 path in G has length $O(D)$. Since there are $O(\log n)$ virtual edges that are replaced on a given cycle, the cycles of G has length $O(D \log n)$.

Cover. We start with some auxiliary property used in our algorithm.

Claim 6. *Consider the graph G_Σ constructed when considering the edges in $E'_{1,2}$. The outdegree of each pair $\sigma' = \langle v'_1, v'_2\rangle \in \Sigma$ in G_Σ is at most 1. Therefore, G_Σ can be colored in 3 colors.*

Proof. Let $\sigma = \langle v_1, v_2\rangle$ be such that σ' interferes with σ (i.e., $(\sigma', \sigma) \in A(G_\Sigma)$). Without loss of generality, let $e = (p(v_1), v_1)$ be such that $e \in \pi(v_1, v_2, T'_1)$ and $\pi(v'_1, v'_2, T'_1)$ intersects the edges of P_e.

We first claim that this implies that e appears above the LCA of v'_1 and v'_2 in T'_1, and hence by the properties of our partitioning, also in T. Assume towards contradiction otherwise, since $e \circ P_e$ is a path on T (where e is closer to the root) and since P_e intersects $\pi(v'_1, v'_2, T'_1)$, it implies that $e \in \pi(v'_1, v'_2, T'_1)$. Since the vertex v_1 is marked, we get a contradiction that v'_1 got matched with v'_2 as the algorithm would have matched v_1 with one of them. In particular, we would get that the paths $\pi(\sigma)$ and $\pi(\sigma')$ are *not* edge disjoint, as both contain e. Hence, we prove that e is above the LCA of v'_1 and v'_2.

Next, assume towards contradiction that there is another pair $\sigma'' = \langle v''_1, v''_2 \rangle \in \Sigma$ such that σ' interferes with σ''. Without loss of generality, let v''_1 be such that $e'' = (p(v''_1), v''_1)$ in on $\pi(v''_1, v''_2, T'_1)$ and $P_{e''}$ intersects with $\pi(v'_1, v'_2, T'_1)$. This implies that e'' is also above the LCA of v'_1 and v'_2 in T'_1. Since one of the edges of $P_{e''}$ is on $\pi(v'_1, v'_2, T'_1)$ it must be that either e'' on P_e or vice verca, in contradiction that $e, e'' \in I(T)$. □

We now claim that each edge $e \in T$ is covered. By the definition of $I(T) \subseteq E(T)$, it is sufficient to show that:

Claim 7. *For every edge $e \in I(T)$, there exists a cycle $C \in \mathcal{C}_3$ such that $e \circ P_e \subseteq C$.*

We now consider a specific tree edge $e = (p(v), v)$. First, note that since $(v, s(v))$ is a non-tree edge, there must be some recursive call with the tree T' such that $v \in T'_1$ and $s(v) \in T'_2$ where T'_1 and T'_2 are the balanced partitioning of T'. At that point, $(v, s(v))$ is an edge in $E'_{1,2}$. We show that in the $\ell = O(\log n)$ phases of the algorithm for covering the $E'_{1,2}$, there is a phase in which $e = (p(v), v)$ is covered.

Claim 8. *(I) For every $e = (p(v), v) \in E'_{1,2}$ except at most one edge e^*, there is a phase i_e where Algorithm* TreeEdgeDisjointPath *matched v with some v' such that $e \in \pi(v, v')$.*
(II) Each edge $e \neq e^*$ is covered by the cycles computed in phase i_e.

Proof. Consider phase i where we need to cover the edges of X_i. Recall, that the algorithm marks the set of nodes v with $(p(v), v) \in X_i$, resulting in the set M_i. Let Σ be the output pairs of Algorithm TreeEdgeDisjointPath(T'_1, M_i). We first show that at least half of the edges in X_i are covered by the paths of Σ.

If M_i is odd, we omit one of the marked nodes and then apply Algorithm TreeEdgeDisjointPath to match the pairs in the even-sized set M_i. The key observation is that for every matched pair $\langle v_1, v_2 \rangle$, it holds that either $(p(v_1), v_1)$ or $(p(v_2), v_2)$ is on $\pi(v_1, v_2, T'_1)$ (or both). Hence, at least half of the edges of X_i are on the edge disjoint paths $\pi(v_1, v_2, T'_1)$.

We therefore get that after $\ell = c \log n$ phases, we are left with $|M_\ell| = O(1)$ at that point if $|M_\ell|$ is odd, we omit one vertex v^* such that $e^* = (p(v^*), v^*)$. Claim (I) follows.

We now consider (II) , let $e = (p(v), v)$ and consider phase $i = i_e$ in which $e \in \pi(v, v', T'_1)$ where v' is the matched pair of v. We show that all the edges of $e \circ P_e$ are covered by the cycles $\mathcal{C}_i$ computed in that phase. By definition, $\langle v, v' \rangle$ belongs to Σ. By Claim 6, G_Σ can be colored by 3 colors, let $\Sigma_j \subseteq \Sigma$ be the color class that contains $\langle v, v' \rangle$.

We will show that there exists a cycle C in $\mathcal{C}_{i,j}$ that covers each edge $e'' \in e \circ P_e$ exactly once. Recall that the algorithm applies Algorithm NonTreeCover which computes a cycle cover $\mathcal{C}''_{i,j}$ to cover all the virtual edges $\widehat{E}_{i,j}$ in T'_2. Also, $(s(v), s(v')) \in \widehat{E}_{i,j}$.

Let C'' be the (simple) cycle in $\mathcal{C}''_{i,j}$ that covers the virtual edge $(s(v), s(v'))$. In this cycle C'' we have two types of edges: edges in T'_2 and virtual edges $(s(v_1), s(v_2))$. First, we transform

C'' into a cycle C' in which each virtual edge $\widehat{e} = (s(v_1), s(v_2))$ is replaced by a path $P(\widehat{e}) = (s(v_1), v_1) \circ \pi(v_1, v_2, T'_1) \circ (v_2, s(v_2))$. Next, we transform C' into $C \subseteq G$ by replacing each edge $(v_1, s(v_1)) \in \widetilde{E}$ in C'' by the v_1-$s(v_1)$ path P_{e_1} for $e_1 = (p(v_1), v_1)$.

We now claim that the final cycle $C \subseteq G$, contains each of the edges $e \circ P_e$ exactly once, hence even if C is not simple, making it simple still guarantees that $e \circ P_e$ remain covered. Since T'_1 and T'_2 are edge disjoint, we need to restrict attention only two types of T'_1 paths that got inserted to C: (I) the edge disjoint paths $\Pi_{i,j} = \{\pi(v_1, v_2, T'_1) \mid \langle v_1, v_2 \rangle \in \Sigma_j\}$ and (II) the v'-$s(v')$ paths $P_{e'}$ for every edge $e' = (p(v'), v')$ (appears on C').

We first claim that there is exactly one path $\pi(v_1, v_2, T'_1) \in \Pi_{i,j}$ that contains the edge $e = (p(v), v)$. By the selection of phase i, $e \in \pi(v, v', T'_1)$ where v' is the pair of v. Since all paths $\Pi_{i,j}$ are edge disjoint, no other path contains e. Next, we claim that there is *no* path $\pi \in \Pi_{i,j}$ that passes through an edge $e' \in P_e$. Since $e = (p(v), v) \in \pi(v, v', T'_1)$ and all edges on P_e are *below* e on T[8], the path $\pi(v, v', T'_1)$ does not contain any $e' \in P_e$. In addition, since all pairs in Σ_j are independent in G_Σ, there is no path in $\pi(\sigma') \in \Pi_{i,j}$ that intersects P_e (as in such a case, σ interferes with $\langle v, v' \rangle$). We get that e appears exactly once on $\Pi_{i,j}$ and no edge from P_e appears on $\Pi_{i,j}$. Finally, we consider the second type of paths in T'_1, namely, the $P_{e'}$ paths. By construction, every $e' \in X_i$ is in $I(T)$ and hence that $P_{e'}$ and P_e share no tree edge. We get that when replacing the edge $(v, s(v))$ with P_e all edges $e' \in P_e$ appears and non of the tree edges on P_e co-appear on some other $P_{e''}$. All together, each edge on $e \circ P_e$ appears on the cycle C exactly once. This completes the cover property. □

Since the edge e^* is covered by taking the fundamental cycle of its swap edge, we get that all edges of $E'_{1,2}$ are covered. Since each edge $(v, s(v))$ belongs to one of these $E'_{1,2}$ sets, the cover property is satisfied.

Congestion. A very convenient property of our partitioning of T' into two trees T'_1 and T'_2 is that this partitioning is closed for LCAs. In particular, for $j \in \{1, 2\}$ then if $u, v \in T'_j$, the LCA of u, v in T' is also in T'_j. Note that this is in contrast to blocks of Section 4.1.1 that are not closed to LCAs.

We begin by proving by induction on $i = \{1, \ldots, O(\log n)\}$ that all the trees T', T''... considered in the same recursion level i are edge disjoint. In the first level, the claim holds vacuously as there is only the initial tree T. Assume it holds up to level i and consider level $i + 1$. As each tree T_j in level $i - 1$ is partitioned into two edge disjoint trees in level $i + 1$, the claim holds.

Note that each edge $e = (v, s(v))$ is considered exactly once, i.e., in one recursion call on $T' = T'_1 \cup T'_2$ where without loss of generality, $v \in T'_1$ and $s(v) \in T'_2 \setminus T'_1$. By Claim 8, there is at most one edge $e^* \in E'_{1,2}$, which we cover by taking the fundamental cycle of $\mathsf{Swap}(e^*)$ in T.

We first show that the congestion in the collection of all the cycles added in this way is bounded by $O(\log n)$. To see this, we consider one level i of the recursion and show that each edge appears on at most 2 of the fundamental cycles $\mathcal{F}_i$ added in that level. Consider an edge e^* that is covered in this way in level i of the recursion. That is the fundamental cycle of $\mathsf{Swap}(e^*)$ given by $\pi(v^*, s(v^*)) \cup P_{e^*}$ was added to $\mathcal{F}_i$. Let T' be such that $T' = T'_1 \cup T'_2$ and $e^* = (p(v^*), v^*)$ is such that $v^* \in T'_1$ and $s(v^*) \in T'_2$. Since both v and $s(v)$ are in T', the tree path $\pi(v^*, s(v^*)) \subseteq T'$. As all other trees $T'' \neq T'$ in level i of the recursion are edge disjoint, they do not have any edge in common with $\pi(v^*, s(v^*))$. For the tree T', there are at most two fundamental cycles that we add. One for covering an edge in $E'_{1,2}$ and one for covering an edge in $E'_{2,1}$. Since $e^* \in I(T)$, and each

[8]Since our partitioning into T'_1, T'_2 maintains the layering structure of T, it also holds that P_e is below e on T'_1.

edge appears on at most two paths $P_e, P_{e'}$ for $e, e' \in I(T)$, overall each edge appears at most twice on each of the cycles in $\mathcal{F}_i$ (once in each direction of the edge) and over all the $O(\log n)$ of the recursion, the congestion due to these cycles is $O(\log n)$.

It remains to bound the congestion of all cycles obtained by translating the cycles computed using Algorithm TreeCover. We do that by showing that the cycle collection $\mathcal{C}_i$ computed in phase i to cover the edges of $E'_{1,2}$ is an $O(D \log n, \log n)$ cover. Since there are $O(\log n)$ phases and $O(\log n)$ levels of recursion, overall it gives an $O(D \log n, \log^3 n)$ cover.

Since all trees considered in a given recursion level are edge disjoint, we consider one of them: T'. We now focus on phase i of Algorithm TreeCover(T'). In particular, we consider the output cycles $\mathcal{C}''_{i,j}$ for $j \in \{1,2,3\}$ computed by Algorithm NonTreeCover for the edges $\widehat{E}_{1,2}$ and T'_2. Each edge $e \in T'_2$ appears on $O(\log n)$ cycles of $\mathcal{C}''_{i,j}$. Each virtual edge $\widehat{e} = (s(v_1), s(v_2))$ is replaced by an $s(v_1)$-$s(v_2)$ path $P(\widehat{e}) = (s(v_1), v_1) \circ \pi(v_1, v_2, T'_1) \circ (s(v_1), v_1)$ in $G \cup \widetilde{E}$. Let $\mathcal{C}'_{i,j}$ be the cycles in $G \cup \widetilde{E}$ obtained from $\mathcal{C}''_{i,j}$ by replacing the edges of $\widehat{e} \in \widehat{E}_{1,2}$ with the paths $P(\widehat{e})$ in $G \cup \widetilde{E}$. Note that every two paths $P(\widehat{e})$ and $P(\widehat{e}')$ are edge disjoint for every $\widehat{e}, \widehat{e}' \in \widehat{E}_{1,2}$. The edges $(s(v_1), v_1)$ of $\widetilde{E}$ gets used only in tree T' in that recursion level. Hence, each edge $(v_1, s(v_1))$ appears on $O(\log n)$ cycles C' in $G \cup \widetilde{E}$. Since the paths $\pi(v_1, v_2, T'_1)$ are edge disjoint, each edge $e' \in \pi(v_1, v_2, T'_1)$ appears on at most $O(\log n)$ cycles C' in $G \cup \widetilde{E}$ (i.e., on the cycles translated from $C'' \in \mathcal{C}''_{i,j}$ that contains the edge $\widehat{e} = (s(v_1), s(v_2))$). Up to this point we get that each virtual edge $(v, s(v)) \in \widetilde{E}$ appears on $O(\log n)$ cycles of $\mathcal{C}'_{i,j}$. Finally, when replacing $(v, s(v))$ with the paths $P_{(p(v),v)}$, the congestion in G is increased by factor of at most 2 as every two path P_e and $P_{e'}$ for $e, e' \in I(T)$, are nearly edge disjoint (each edge (z, w) appears on at most twice of these paths, one time in each direction). We get that the cycle collection $\mathcal{C}_i$ is an $O(D \log n, \log n)$ cover, as desired.

4.2 Private Neighborhood Trees

We introduce the notion of *Private Neighborhood Trees* in which the graph G is decomposed into overlapping trees $T(u_1), \ldots, T(u_n)$ such that each tree $T(u_i)$ contains the neighbors of u_i in G but *does not* contain u_i. Hence, $T(u_i)$ provides the neighbors of u_i a way to communicate privately without their root u_i. The goal is to compute a collection of trees (or clusters) with small overlap and small diameter. Thus, we are interested in the existence of a *low-congestion private neighborhood trees*.

Definition 6 (Private Neighborhood Trees)**.** *Let $G = (V = \{u_1, \ldots, u_n\}, E)$ be an biconnected graph. The private neighborhood trees $\mathcal{N}$ of G is a collection of n subtrees $T(u_1), \ldots, T(u_n)$ in G such that for every $i \in \{1, \ldots, n\}$ it holds that $\Gamma(u_i) \setminus \{u_i\} \subseteq T(u_i)$, but $u_i \notin T(u_i)$. An (d, c) private neighborhood trees $\mathcal{N}$ satisfies:*

1. $\mathsf{Diam}(T(u_i)) = O(\mathsf{d})$ *for every* $i \in \{1, \ldots, n\}$,
2. *Every edge $e \in E$ appears in at most $O(\mathsf{c})$ trees.*

Note that since the graph is biconnected, all nodes of $\Gamma(u)$ are indeed connected in $G \setminus \{u\}$ for every node u. The main challenge is in showing that all n trees can be both of small diameter and with small overlap.

Theorem 3 (Private Trees)**.** *For every biconnected graph G with maximum degree Δ and diameter D, there exists a (d, c) private trees with $\mathsf{d} = O(D \cdot \Delta \cdot \log n)$ and $\mathsf{c} = O(D \cdot \log \Delta \cdot \log^3 n)$.*

In fact we show that given a construction of $(\mathtt{d}, \mathtt{c})$ cycle cover $\mathcal{C}$, we can construct an $(\mathtt{d} \cdot \Delta, \mathtt{c} \cdot D \cdot \log \Delta)$ private neighborhood trees $\mathcal{N}$. In particular, using our construction of $(D \log n, \log^3 n)$ cycle cover $\mathcal{C}$ yields the desired bound.

The construction of the private neighborhood trees $\mathcal{N}$ consists of $\ell = O(\log \Delta)$ phases. In each phase, we compute an $(D \log n, \log^3 n)$ cycle cover in some auxiliary graph using Theorem 1. We start by having for each node u an empty forest $F_0(u) = (\Gamma(u, G), \emptyset)$ consisting only of u's neighbors. Then in each phase, we add edges to these forests so that the number of connected components (containing the neighbors $\Gamma(u, G)$) is reduced by factor 2. After $O(\log \Delta)$ phases, we will have for every $u \in V$, a tree $T(u)$ in $G \setminus \{u\}$ that spans all neighbors $\Gamma(u, G)$. We need some notation. Let $\mathcal{C}_0$ be a cycle cover of G. For every $i \in \{0, \ldots, \ell\}$, let $CC_i(u)$ be the number of connected components in the forest $F_i(u)$. In particular, $CC_0(u) = \deg(u)$.

In each phase $i \geq 1$, we are given a collection of forests $\mathcal{N}_{i-1} = \{F_{i-1}(u_1), \ldots, F_{i-1}(u_n)\}$ such that (I) $F_{i-1}(u_j) \subseteq G \setminus \{u_j\}$, (II) $\Gamma(u_j) \subseteq V(F_{i-1}(u_j))$ and (III) $F_{i-1}(u_j)$ has $CC_{i-1}(u_j) \leq \deg(u_j)/2^{i-1}$ connected components. The goal of phase i is to add edges to each $F_{i-1}(u_j)$ in order to reduce the number of connected components by factor 2. The algorithm uses the current collection of forests $\mathcal{N}_{i-1}$ to define an auxiliary graph $\widetilde{G}_i$ which contains the edges of G and some additional virtual nodes and edges. For every $u \in V$, we add to $\widetilde{G}_i$ a set of $k = CC_{i-1}(u)$ virtual nodes $\widetilde{u}_1, \ldots, \widetilde{u}_k$. We connect u to each of its virtual copies $\widetilde{u}_j$ and in addition connect the j^{th} virtual copy $\widetilde{u}_j$ is connected to all the neighbors of u (in $\Gamma(u, G)$) that belong to the j^{th} connected component in the forest $F_{i-1}(u) \in \mathcal{N}_{i-1}$. This is done for every $u \in G$. The final auxiliary graph $\widetilde{G}_i$ has $O(m)$ nodes, $O(m)$ edges and diameter at most $2D$. Note that each edge of $\widetilde{G}_i$ has one real endpoint (i.e., node of G) and one virtual endpoint. Next, the algorithm uses Algorithm CycleCover to compute an $(D \log n, \log^3 n)$ cycle cover $\widetilde{\mathcal{C}}_i$ for the edges of $\widetilde{G}_i$. To map these virtual cycles to real cycles $\mathcal{C}_i$ in G, we simply replace a virtual node $\widetilde{u}_j$ with the real node u. As the virtual neighbors of u, namely, $\widetilde{u}_j$, are connected to the neighbors of u, this indeed defines legal cycles in G. Define $G_i(u) = F_{i-1}(u) \cup \{C \mid C \in \mathcal{C}_i \text{ and } (u, v) \in C\} \setminus \{u\}$ and let $F_i(u) \subseteq G_i(u)$ be a forest that spans all the neighbors of u. This forest can be computed, for instance, by running a BFS from a neighbor u in each connected component of $G_i(u)$. This completes the description of phase i. The final private tree collection is given by $\mathcal{N} = \{F_\ell(u_1), \ldots, F_\ell(u_n)\}$. We now turn to analyze this construction and prove Theorem 3.

Small Diameter Trees. We begin by showing that the diameter of each tree $T(u_i)$ is bounded by $O(\Delta D \cdot \log n)$. Note that this bound is existentially tight (up to logarithmic factors) as there are graphs G with diameter D and there is a node u with degree Δ such that the diameter of $G \setminus \{u\}$ is $O(\Delta D)$.

Claim 9. *For every $i \in \{0, \ldots, \log \Delta\}$ and for every $u \in V$ the number of connected components satisfies $CC_i(u) \leq \Delta/2^i$.*

Proof. The lemma is shown by induction on i. The case of $i = 0$ holds vacuously. Assume that the claim holds up to $i - 1$ and consider phase i. By construction, for each u, the auxiliary graph $\widetilde{G}_i$ contains $CC_{i-1}(u)$ virtual nodes $\widetilde{u}_j$ that are connected to u.

The cycle cover $\widetilde{\mathcal{C}}_i$ for $\widetilde{G}_i$ covers all these virtual $(u, \widetilde{u}_j)$ by virtual cycles, each such cycle connects two virtual nodes. Since every two virtual nodes of u in $\widetilde{G}_i$ are connected to neighbors of u that belong to different components in $G_{i-1}(u)$, every cycle that connects two virtual neighbors is mapped into a cycle that connects two of u's neighbors that belong to a different connected

component in phase $G_{i-1}(u)$. Hence, the number of connected components in the forest $F_i(u)$ has been decreased by factor at least 2 compared to that of $F_{i-1}(u)$. □

Claim 10. *The diameter of each tree $T(u_i) \in \mathcal{N}$ is $O(\Delta \cdot D \cdot \log n)$.*

Proof. We first claim that the diameter of each component in the forest $F_i(u)$ is bounded by $O(\Delta \cdot D \cdot \log n)$ for every $u \in V$ and every $i \in \{1, \ldots, \ell\}$. To see this, note that the forest $F_i(u)$ is formed by collection of $O(D \log n)$-length cycles that connect u's neighbors. Hence, when removing u, we get paths of length $O(D \log n)$. Consider the process where in each phase i, every two u's-neighbors that are connected by a cycle in $\mathcal{C}_i$ are connected by a single "edge". By the Proof of Claim 9, after $\ell = O(\log \Delta)$ phases, we get a connected tree with $\deg(u)$ nodes, and hence of "diameter" $\deg(u)$. Since each edge corresponds to a path of length $O(D \log n)$ in G, we get that the final diameter of $F_\ell(u)$ is $O(\deg(u) \cdot D \cdot \log n)$. □

Congestion.

Claim 11. *Each edge e appears on $O(D \log^3 n)$ different subgraphs $T(u_i) \in \mathcal{N}$.*

We first show that the cycles $\mathcal{C}_i$ computed in G have congestion $O(\log^3 n)$ for every $i \in \{1, \ldots, \ell\}$. Clearly, the cycles $\widetilde{\mathcal{C}}_i$ computed in $\widetilde{G}_i$ have congestion of $O(\log^3 n)$. Consider the mapping of cycles $\widetilde{\mathcal{C}}_i$ in $\widetilde{G}_i$ to a cycles $\mathcal{C}_i$ in G. Edges of the type $(u, \widetilde{u}_j)$ are replaced by (u, u) and hence there is no real edge in the cycle. Edges of the type $(\widetilde{u}_j, w)$ are replaced by (u, w). Since there is only one virtual node of u that connects to w, and since $(\widetilde{u}_j, w)$ appears in $O(\log^3 n)$ many cycles, also (u, w) appears in $O(\log^3 n)$ many cycles (i.e., this conversion does not increase the congestion).

Note that the cycle C of each edge (u, v) joins the G_i subgraphs of at most D nodes since in our construction a cycle C might cover up to D edges. (Recall that in Algorithm TreeCover a cycle that covers an a tree edge e, also covers up to D edges on the path P_e). In addition, each edge e' appears on different cycles in $\mathcal{C}_i$.

We now claim that each edge e appears on $O(i \log^3 n \cdot D)$ graphs $G_i(u)$. For $i = 1$, this holds as the cycle C of an edge (u, v) joins the subgraphs $G_1(x)$ and $G_1(y)$ for every edge (x, y) that is covered by C. Assume it holds up to $i - 1$ and consider phase i. In phase i, we add to the $G_i(u)$ graphs the edges of $\mathcal{C}_i$. Again, each cycle C' of an edge (u, v) joins D graphs $G_i(x), G_i(y)$ for every (x, y) that is covered by C'. Hence each edge e appears on $O(D \cdot \log^3 n)$ of the subgraphs $G_i(u_j) \setminus G_{i-1}(u_j)$. By induction assumption, each e appears on $(i-1) \log^3 n \cdot D$ graphs $G_{i-1}(u_j)$ and hence overall each edge e appears on $O(i \log^3 n)$ graphs $G_i(u_j)$. Therefore we get that each edge appears on $O(\log \Delta \cdot \log^3 n \cdot D)$ trees in $\mathcal{N}$.

5 Secure Simulation via Low-Congestion Covers

In this section we describe how to transform any distributed algorithm $\mathcal{A}$ to a new algorithm $\mathcal{A}'$ which has the same functionality as $\mathcal{A}$ (i.e., the output for every node u in $\mathcal{A}$ is the same as in $\mathcal{A}'$) but has perfect privacy (as is defined in Definition 1). Towards this end, we assume that the combinatorial structures required are already computed (in a preprocessing stage), namely, a private neighborhood tree in the graph. The output of the preprocessing stage is given in a distributed manner. The (distributed) output of the private neighborhood trees for each node u, is such that each vertex v knows its parent in the private neighborhood tree of u (if such exists).

Theorem 4. *Let G be an n-vertex graph with diameter D and maximal degree Δ. Let $\mathcal{A}$ be a natural distributed algorithm that works on G in r rounds. Then, $\mathcal{A}$ can be transformed to an equivalent algorithm $\mathcal{A}'$ with perfect privacy which runs in $\widetilde{O}(rD \cdot \mathsf{poly}(\Delta))$ rounds (after a preprocessing stage).*

As a preparation for our secure simulation, we provide the following convenient view of distributed algorithm.

5.1 Our Framework

We treat the distributed r-round algorithm $\mathcal{A}$ from the view point of some fixed node u, as a collection of r functions $f_1, \ldots, f_r$ as follows. Let $\Gamma(u) = \{v_1, \ldots, v_k\}$. At any round i, the memory of u consists of a state, denoted by σ_i and Δ messages $m_{v_1 \to u} \ldots, m_{v_\Delta \to u}$ that were received in the previous round (in the degree of the node is less than Δ the rest of the messages are empty). Initially, we set σ_0 to be a fixed string and initialize all messages to NULL. At round i the node u updates its state to σ_{i+1} according to its previous state σ_i and the messages that it got in the previous round. It then prepares k messages to send $m_{u \to v_1}, \ldots, m_{u \to v_\Delta}$. To ease notation (and without loss of generality) we assume that each state contains the ID of the node u. Thus, we can focus on a single update function f_i for every round that works for all nodes. The function f_i gets the state σ_i, the messages $m_{v_1 \to u} \ldots, m_{v_\Delta \to u}$, and the randomness s. The output of f_i is the next state σ_{i+1}, and at most k outgoing messages:

$$(\sigma_i, m_{u \to v_1}, \ldots, m_{u \to v_\Delta}) \leftarrow f_i(\sigma_{i-1}, m_{v_1 \to u}, \ldots, m_{v_\Delta \to u}, s).$$

Our compiler works *round-by-round* where each round i is replaced by a collection of rounds that "securely" compute f_i, in a manner that will be explained next. The complexity of our algorithm depends exponentially on the space complexity of the functions f_i. Thus, we proceed by transforming the original algorithm $\mathcal{A}$ to one in which each f_i can be computed in logarithmic space, while slightly increasing the number of rounds.

Claim 12. *Any natural distributed algorithm $\mathcal{A}$ the runs in r rounds can be transformed to a new algorithm $\widehat{\mathcal{A}}$ with the same output such that $\widehat{\mathcal{A}}$ is computable in logarithmic space using $r' = r \cdot \mathsf{poly}(\Delta + \log n)$ rounds.*

Proof. Let t be the running time of the function f_i. Then, f_i can be computed with a circuit of at most t gates. Note that since $\mathcal{A}$ is natural, it holds that $t \leq \mathsf{poly}(\Delta, \log n)$.

Instead of letting u computing f_i in round i, we replace the ith round by t rounds where each round computes only a single gate of the function f_i. These new rounds will have no communication at all, but are used merely for computing f_i with a *small* amount of memory.

Let $g_1, \ldots, g_t$ be the gates of the function f_i in a computable order where g_t is the output of the function. We define a new state σ'_i of the form $\sigma' = (\sigma_i, g_1, \ldots, g_t)$, where σ_i is the original state, and g_j is the value of the jth gate. Initially, $g_1, \ldots, g_t$ are set to $\perp$. Then, for all $j \in [t]$ we define the function

$$f_i^j(\sigma_i, g_1, \ldots, g_{j-1}, \perp, \ldots, \perp) = (\sigma_i, g_1, \ldots, g_{j-1}, g_j, \perp, \ldots, \perp).$$

In the jth round we compute f_i^j, until the final g_t is computed. Note that f_i^j can be computed with logarithmic space, and since $t \leq \mathsf{poly}(\Delta, \log n)$ we can compute f_i^j with space $O(\log \Delta + \log \log n)$. As a result, the r-round algorithm $\mathcal{A}$ is replaced by an rt-round algorithm $\widehat{\mathcal{A}}$, where $t \leq \mathsf{poly}(\Delta, \log n)$. That is, we have that $r' \leq \mathsf{poly}(\Delta, \log n)$. □

As we will see, our compiler will have an overhead of $\mathsf{poly}(\Delta, \log n)$ in the round complexity and hence the overhead of Claim 12 is insignificant. Thus, we will assume that the distributed algorithm $\mathcal{A}$ satisfies that all its functions f_i are computable in logarithmic space (i.e., we assume that the algorithm is already after the above transformation).

5.2 Secure Simulation of a Single Round

In the algorithm $\mathcal{A}$ each node u computes the function f_i in each round i. In our secure algorithm $\mathcal{A}'$ we want to simulate this computation, however, on *encrypted* data, such that u does not get to learn the true output of f_i in any of the rounds except for the last one. When we say "encrypted" data, we mean a "one-time-pad" (see Definition 4). That is, we merely refer to a process where we the data is masked by XORing it with a random string R. Then, R is called the encryption (and also decryption) key. Using this notion, we define a related function f'_i that, intuitively, simulates f_i on encrypted data, by getting encrypted state and messages as input, decrypting them, then computing f_i and finally encrypting the output with a new key. We simulate every round of the original algorithm $\mathcal{A}$ by a PSM protocol for the function f'_i.

The Secure Function f'_i. The function f'_i gets the following inputs (encrypted elements will be denoted by the $\widehat{\cdot}$ notation):

1. An encrypted state $\widehat{\sigma}_{i-1}$ and encrypted messages $\{\widehat{m}_{v_j \to u}\}_{j=1}^{\Delta}$.
2. The decryption key $R_{\sigma_{i-1}}$ of the state $\widehat{\sigma}_{i-1}$ and the decryption keys $\{R_{v_j \to u}\}_j^{\Delta}$ for the messages $\{\widehat{m}_{v_j \to u}\}_{j=1}^{\Delta}$.
3. Shares for randomness $\{R_s^j\}_{j=1}^{\Delta}$ for the function f_i.
4. Encryption keys for encrypting the new state R_{σ_i} and messages $\{R_{u \to v_j}\}_{j=1}^{\Delta}$.

The function f'_i decrypts the state and messages and runs the function f_i (using randomness $s = \bigoplus R_s^j$) to get the new state σ_i and the outgoing messages $m_{u \to v_1}, \ldots, m_{u \to v_\Delta}$. Then, it encrypts the new state and messages using the encryption keys. In total, the function f'_i has $O(\Delta)$ input bits. The precise description of f'_i is given in Figure 13.

Recall that in the PSM model, we have k parties $p_1, \ldots, p_k$ and a server s, where it was assumed that (PI) all parties have private shared randomness (not known to s) and (PII) that each of party has a private communication channel to the server. Our goal is to compute f'_i securely by implementing a PSM protocol for all nodes in the graph simultaneously. We call this implementation a *distributed* PSM *algorithm*.

The distributed PSM algorithm securely computes f'_i by simulating the PSM protocol for f'_i, treating u as the *server* and its immediate neighborhood as the *parties*. Since each party is an immediate neighbor of the server, (PII) is provided easily. Our main efforts is in providing (PI). Towards that goal, we define the notion of private neighborhood trees which provides us the communication backbone for implementing this distributed PSM protocol in general graph topologies for all nodes simultaneously.

The private neighborhood tree collection consists of n trees, a tree T_u for every u, that spans all the neighbors of u (i.e., the parties) without going through u, i.e., $T_u \subseteq G \setminus \{u\}$. Using this tree, all the parties can compute shared private random bits R which are not known to u. For

The description of the function f'_i.
Input: An encrypted state $\widehat{\sigma}_{i-1}$, encrypted messages $\left\{\widehat{m}_{v_j\to u}\right\}_{j=1}^{\Delta}$, keys for decrypting the input $R_{\sigma_{i-1}}, \left\{R_{v_j\to u}\right\}_j^{\Delta}$, randomness $\left\{R_s^j\right\}_{j=1}^{\Delta}$ and keys for encrypting the output $R_{\sigma_i}, \left\{R_{u\to v_j}\right\}_{j=1}^{\Delta}$.
Run:

1. Compute $\sigma_{i-1} \leftarrow \widehat{\sigma}_{i-1} \oplus R_{\sigma_{i-1}}$ and $s \leftarrow \left(\bigoplus_{j=1}^{\Delta} R_s^j\right)$.
2. For $j = 1 \ldots \Delta$: compute $m_{v_j\to u} \leftarrow \widehat{m}_{v_j\to u} \oplus R_{v_j\to u}$.
3. Run $\sigma_i, m_{u\to v_1}, \ldots, m_{u\to v_\Delta} \leftarrow f(\sigma_{i-1}, m_{v_1\to u}, \ldots, m_{v_\Delta\to u}, s)$.
4. Compute $\widehat{\sigma_i} \leftarrow \sigma_i \oplus R_{\sigma_i}$.
5. For $j = 1 \ldots \Delta$: compute $\widehat{m}_{u\to v_j} \leftarrow m_{u\to v_j} \oplus R_{u\to v_j}$.
6. Output $\widehat{\sigma_i}, \widehat{m}_{u\to v_1}, \ldots, \widehat{m}_{u\to v_\Delta}$.

Figure 13: The function f'_i.

a single node u, this can be done in $O(\mathsf{Diam}(T_u) + |R|)$ rounds, where $\mathsf{Diam}(T_u)$ is the diameter of the tree and R is the number of random bits. Clearly, our objective is to have trees T_u with small diameter. Furthermore, as we wish to implement this kind of communication in all n trees, $T_{u_1}, \ldots, T_{u_n}$ simultaneously, a second objective is to have small overlap between the trees. That is, we would like each edge e to appear only on a small number of trees T_u (as on each of these trees, the edge is required to pass through different random bits). These two objectives are encapsulated in our notion of *private-neighborhood-trees*. The final algorithm $\mathcal{A}'_i(u)$ for securely computing f'_i is described in Figure 14.

The algorithm $\mathcal{A}'_i(u)$ for securely computing f'_i.
Input: Each node $v \in \Gamma(u)$ has input $x_v \in \{0,1\}^m$.

1. Let T_u be the tree spanning $\Gamma(u)$ in $G \setminus \{u\}$ and let w be the root.
2. w chooses a random string R and sends it to $\Gamma(u)$ using the tree T_u.
3. Each node $v \in \Gamma(u)$ computes $M_v = \mathsf{PSM.Enc}(f'_i, x_v, R)$ and sends it to u.
4. u computes $y = \mathsf{PSM.Dec}\left(f'_i, \{M_v\}_{v\in\Gamma(u)}\right)$.

Figure 14: The description of the distributed PSM algorithm of node u for securely computing the function f'_i.

In what follows analyze the security and round complexity of Algorithm $\mathcal{A}'_i$.

Round Complexity. Let $f : \{0,1\}^{m\cdot|\Gamma(u)|} \to \{0,1\}^\ell$ be a function with $|\Gamma(u)| \le \Delta$ inputs, where each input is of length m bits. The communication complexity of the PSM protocol depends on the input and output length of the function and also on the memory required to compute f. Suppose

that f is computable by an s-space TM. Then, by Theorem 6 the communication complexity (and randomness complexity) of the protocol is at most $O(\Delta m \ell \cdot 2^{2s})$.

In the first phase of the protocol, the root w sends a collection of random bits R to $\Gamma(u)$ using the private neighborhood trees, where $|R| = O(\Delta \cdot m \cdot \ell \cdot 2^{2s})$. By Theorem 3, the diameter of the tree is at most $\widetilde{O}(D\Delta)$ and each edge belongs to $\widetilde{O}(D)$ different trees. Therefore, there are total of $\widetilde{O}(D \cdot |R|)$ many bits that need to go through a single edge when sending the information on all trees simultaneously. Using the random delay approach of Theorem 5, this can be done in $\widetilde{O}(D\Delta + D \cdot |R|) = \widetilde{O}(\Delta \cdot D \cdot m \cdot \ell \cdot 2^{2s})$ rounds. This is summarized by the following Lemma:

Lemma 4. *Let $f : (\{0,1\}^m)^\Delta \to \{0,1\}^\ell$ be a function over Δ inputs where each is of length at most m and that is computable by a s-space TM. Then, there is a distributed algorithm $\mathcal{A}'_i(u)$ (in the* CONGEST *model) with perfect privacy where each node u outputs f evaluated on $\Gamma(u)$. The round complexity of $\mathcal{A}'_i(u)$ is $\widetilde{O}(\Delta \cdot D \cdot m \cdot \ell \cdot 2^{2s})$.*

5.3 The Final Secure Algorithm

Using the function f'_i, we define the algorithm $\mathcal{A}'_u$ for computing the next state and messages of the node u. We describe the algorithm for any u in the graph and at the end we show that all the algorithms $\{\mathcal{A}'_u\}_{u \in G}$ can be run simultaneously with low congestion.

The algorithm $\mathcal{A}'_u$ involves running the distributed algorithm $\mathcal{A}'_i(u)$ for each round $i \in \{1, \ldots, r\}$. The secure simulation of round i starts by letting the root of each tree T_u (i.e., the tree connecting the neighbors of u in $G \setminus \{u\}$) sample a key R_{σ_i} for encrypting the new state of u. Moreover, each neighbor v_j of u samples a share of the randomness R^j_s used to evaluate the function f_i, and a key $R_{u \to v_j}$ for encrypting the message sent from u to v_j.

Then they run $\mathcal{A}'_i(u)$ algorithm with u as the server and $\Gamma(u)$ as the parties for computing the function f'_i (see Figure 14). The node u has the encrypted state and message, the neighbors of u have the (encryption and decryption) keys for the current state, the next state and the sent messages, and moreover the randomness for evaluating f'_i. At the end of the protocol, u computes the output of f'_i which is the encrypted output of the function f_i.

After the final round, u holds an encryption of the final state $\widehat{\sigma_r}$ which contains only the output of the original algorithm $\mathcal{A}$. At this point, the neighbors of u send it the decryption key for this last state, u decrypts its state and outputs the decrypted state. Initially, the state σ_0 is a fixed string which is not encrypted, and the all encryption keys for this round are assumed to be 0. This description is summarized in Figure 15. Finally, we show that the protocol is correct and secure.

Correctness. The correctness follows directly from the construction. Consider a node u in the graph. Originally, u computes the sequence of states $\sigma_0, \ldots, \sigma_r$ where σ_r contained the final output of the algorithm. In the compiled algorithm $\mathcal{A}'$, for each round i of $\mathcal{A}$ and every node u the sub-algorithm $\mathcal{A}'_i(u)$ computes $\widehat{\sigma_i}$, where $\widehat{\sigma_i} = \sigma_i \oplus R_{\sigma_i}$ where v_1 holds R_{σ_i}. Thus, after the last round, u has $\widehat{\sigma_r}$ and v_1 has R_{σ_r}. Finally, u computes $\widehat{\sigma_r} \oplus R_{\sigma_r} = \sigma_r$ and outputs σ_r as required.

Round Complexity. We compute the number of rounds of the algorithm for any natural algorithm $\mathcal{A}$. The algorithm consists of $r' = r \cdot \mathsf{poly}(\Delta + \log n)$ iterations. In each iteration, every vertex u implements algorithm $\mathcal{A}'_i$ for the function f'_i (there are other operations in the iteration but they are negligible). We know that f_i can be computed in s-space where $s = O(\log \Delta + \log \log n)$, and thus we can bound the size of each input to f'_i by $\mathsf{poly}(\Delta) \cdot \mathsf{polylog}(n)$. Indeed, the state has this bound by the definition of a natural algorithm, and thus also the encrypted state (which has the

The description of the algorithm $\mathcal{A}'_u$.

1. Let $v_1, v_2, \ldots, v_\Delta$ be some arbitrary ordering on $\Gamma(u)$.
2. For each round $i = 1 \ldots r$ do:
   (a) u sends $\widehat{\sigma}_{i-1}$ to neighbor v_2.
   (b) Each neighbor v_j of u samples R^j_s at random (and stores it).
   (c) v_1 chooses R_{σ_i} at random (and stores it).
   (d) Run the $\mathcal{A}_i(u)$ algorithm for f'_i with server u and parties $\Gamma(u)$ where:
      i. v_1 has an inputs $R_{\sigma_{i-1}}$ and R_{σ_i} and v_2 has input $\widehat{\sigma}_{i-1}$.
      ii. In addition, each neighbor v_j of u has input $R_{u\to v_j}, R^j_s$.
      iii. u learns the final output of the algorithm $(\widehat{\sigma}_i, \widehat{m}_{u\to v_1}, \ldots, \widehat{m}_{u\to v_\Delta})$.
3. v_1 sends R_{σ_r} to u.
4. u computes $\sigma_r = \widehat{\sigma}_r \oplus R_{\sigma_r}$ and outputs σ_r.

Figure 15: The description of the Algorithm $\mathcal{A}'_u$. We assume that in "round 0" all keys are initialized to 0. That is, we let $R_{\sigma_0} = 0$, and initially set $R_{v_j\to u} = 0$ for all $j \in [\Delta]$.

exact same size), the messages and encryption keys for the messages have length at most $\log n$, and the randomness shares are of size at most the running time of f_i which is at most 2^s where s is the space of f_i and thus the bound holds. The output length shares the same bound as well.

Since f_i can be computed in s-space where $s = O(\log \Delta + \log\log n)$, we observe that f'_i can be computed in s-space as well. This includes running f_i is a "lazy" manner. That is, whenever the TM for computing f_i asks to read a the i^{th} bit of the input, we generate the this bit by performing the appropriate XOR operations for the i^{th} bit of the input elements. The memory required for this is only storing indexes of the input which is $\log(\Delta \cdot \mathsf{poly}(\log n))$ bits and thus s bits suffice.

Then, by Lemma 4 we get that algorithm $\mathcal{A}'_i(u)$ for f'_i runs in $\widetilde{O}(D \cdot \mathsf{poly}(\Delta))$ rounds, and the total number of rounds of our algorithm is $\widetilde{O}(rD \cdot \mathsf{poly}(\Delta))$. In particular, if the degree Δ is bounded by $\mathsf{polylog}(n)$ then we get $\widetilde{O}(rD)$ number of rounds.

Remark 1 (Round complexity for non-natural algorithms)**.** *If $\mathcal{A}$ is not a "natural" algorithm then we can bound the number of rounds with dependency on the time complexity of the algorithm. If each function f_i (the local computation of the nodes) can be computed by a circuit of size t then the number of rounds of the compiled algorithm is bounded by $\widetilde{O}(rDt \cdot \mathsf{poly}(\Delta))$.*

Security. We begin by describing the security of a single sub-protocol $\mathcal{A}'_u$ for any node u in the graph. The algorithm $\mathcal{A}'_u$ has many nodes involved, and we begin by showing how to simulate the messages of u. Fix an iteration i, and consider the all the messages sent to u by the PSM protocol in $\mathcal{A}'_i(u)$ denoted by $\{M_v\}_{v\in G}$, and let $\widehat{\sigma}_i, \widehat{m}_{u\to v_1}, \ldots, \widehat{m}_{u\to v_\Delta}$ be the output of the protocol. By the security of the PSM protocol, there is a simulator Sim such that the following two distributions are equal:

$$\{M_v\}_{v\in G} \equiv \mathsf{Sim}(\widehat{\sigma}_i, \widehat{m}_{u\to v_1}, \ldots, \widehat{m}_{u\to v_\Delta}).$$

Since $\widehat{\sigma_i}$ and $\widehat{m}_{u\to v_1},\ldots,\widehat{m}_{u\to v_\Delta}$ are encrypted by keys that are never sent to u we have that from the viewpoint of u the distribution of $\widehat{\sigma_i}$ and of $\widehat{m}_{u\to v_1},\ldots,\widehat{m}_{u\to v_\Delta}$ are uniformly random. Thus, we can run the simulator with a random string R of the same length and have

$$\mathsf{Sim}(\widehat{\sigma_i},\widehat{m}_{u\to v_1},\ldots,\widehat{m}_{u\to v_\Delta}) \equiv \mathsf{Sim}(R).$$

While this concludes the simulator for u, we need to show a simulator for other nodes that participate in the protocol. Consider the neighbors of u. The neighbor v_1 has the encryption key for the state, and v_2 has the encrypted state. Since they never exchange this information, each of them gets a uniformly random string. In addition to their own input, the neighbors have the shared randomness for the PSM protocol. All these elements are uniform random strings which can be simulated by a simulator Sim by sampling a random string of the same length.

To conclude, the privacy of $\mathcal{A}'_i(u)$ follows from the perfect privacy of PSM protocol we use. The PSM security guarantees a perfect simulator for the server's viewpoint, and it is easy to construct a simulator for all other parties in the protocol as they only receive random messages. While the PSM was proven secure in a stand alone setting, in our protocol we have a composition of many instances of the protocol. Fortunately, it was shown in [KLR10] that any protocol that is perfectly secure and has a black-box non-rewinding simulator, is also secure under universal composability, that is, security is guaranteed to hold when many arbitrary protocols are performed concurrently with the secure protocol. We observe that the PSM has a simple simulator that is black-box and non-rewinding, and thus we can apply the result of [KLR10]. This is since the simulator of the PSM protocol is an algorithm that runs the protocol on an arbitrary message that agrees with the output of the function.

6 Distributed Computation of Low Congestion Covers

In this section, we describe the distributed algorithms for constructing the low-congestion covering structures. We start by describing the construction of low-congestion cycle cover.

6.1 Cycle Cover

We describe Algorithm DistCycleCover which constructs a low congestion cycle cover in the distributed setting. The output of the algorithm is defined as follows: the endpoints of every edge $e=(u,v)$ know all the identifiers of all the edges that are covered by the cycles that go through e.

Theorem 2. *For every 2-edge connected n-node graph $G=(V,E)$ with diameter D and maximum degree Δ, Algorithm* DistCycleCover *computes a $(4^{1/\epsilon}\cdot D\cdot\log\Delta, n^{\epsilon}\log^2 n)$ cycle cover $\mathcal{C}$ in $\widetilde{O}(4^{1/\epsilon}\cdot D+n^{\epsilon})$ rounds for every $\epsilon\in(0,1)$.*

We start by claiming that up to logarithmic factors in the round complexity and in the quality of our cycle cover, it is sufficient to restrict attention to graphs with maximum degree 5.

Lemma 5. *Let $\mathcal{A}'$ be an algorithm that computes a (d,c) cycle cover for every N-node graph G' with maximum degree 5 within $f(N,\mathsf{d},\mathsf{c},\mathsf{Diam}(G))$ rounds. Then, there exists an algorithm $\mathcal{A}$ that computes a $(\mathsf{d}\log\Delta,\mathsf{c})$ cycle cover for every n-node graph $G=(V,E)$ with maximum degree Δ, within $f(|E|,\mathsf{d}\log\Delta,\mathsf{c},\mathsf{Diam}(G)\cdot\log\Delta)$ rounds.*

Proof. The 2-edge connected graph G can be transformed to a graph G' with bounded degree 5 in the following manner: For every node u with degree $\ell = \deg(u)$, we add in G', $2\ell - 1$ many copies $\widehat{u}_1, \ldots, \widehat{u}_{2\ell-1}$ that are connected as follows. The node u is the root of a binary tree $\widehat{T}_u$ of depth $\log(\deg(u))$ and with the leaf nodes $\widehat{u}_1, \ldots, \widehat{u}_\ell$ which correspond to the $\Gamma(u, G)$ nodes. That is, each leaf node $\widehat{u}_j$ corresponds to the j^{th} neighbor of u. In addition, all these leaf nodes are connected via a path. The leaf nodes of different trees are connected as follows: the leaf node $\widehat{u}_j$ in $\widehat{T}_u$ that corresponds to the j^{th} neighbor $v \in \Gamma(u, G)$ is connected to the corresponding copy of u in $\widehat{T}_v$, namely, $\widehat{v}_i$. Overall we get a 2-edge connected graph with $\Theta(|E(G)|)$ nodes and $\Theta(|E(G)|)$ edges, has maximum degree 5 and is of diameter $\Theta(\log \Delta \cdot \mathsf{Diam}(G))$. Any round of an algorithm for G' can be simulated by a single round in the graph G. In addition, any cycle computed in G' can be easily translated into a cycle in G without increasing the congestion by replacing each virtual copy $\widehat{u}_j$ with the node u. □

Given Lemma 5, we can assume without loss of generality, that G has maximum degree $O(1)$. As in the centralized construction, we start by constructing a BFS tree T. We first cover all the non-tree edges E' using Algorithm DistNonTreeCover, and then cover the tree edges $E(T)$, using Algorithm DistTreeCover. Since we already assume that we work on bounded degree graphs (by paying an extra factor of $\log \Delta$ in the length of the cycles), there is no need for the preliminary step that computes short edge disjoint cycles of length $O(\log n)$.

6.1.1 Distributed Algorithm for Covering Non-Tree Edges

In this subsection we describe Algorithm DistNonTreeCover that constructs an $(2^{1/\epsilon} \cdot D, n^{2\epsilon})$ cycle cover $\mathcal{C}_1$ for the non tree edges. We next describe two basic procedures used by algorithm and then describe how to used them to compute the cover.

Tool (I): Computing an Hierarchy of Block Partitioning. We are given a BFS tree T, a subset of non-tree edges E' and an input parameter $\epsilon \in [0, 1]$. Computing the cycle cover is based upon constructing a hierarchy of $(E', \mathtt{b}_i)$ block partitioning where $\mathtt{b}_i = m/n^{i\cdot\epsilon}$ for every $i \in \{0, \ldots, \ell\}$ where $\ell = c \cdot \lceil 1/\epsilon \rceil$. As in Section 4.1, every block consists of a subset of nodes with consecutive postorder traversal numbering. To see how we can compute the postorder numbering distributively, see Appendix B. We now define the hierarchical block partitioning and then explain how to compute it in a distributed manner. The 0-level of the hierarchy consists of one block containing all nodes. In the i^{th}-level of the hierarchy, each $(i-1)$-level block is subdivided into n^ϵ sub-blocks, each with bounded density of $\mathtt{b}_i = m/n^{i\cdot\epsilon}$. This subdivision is done greedily, going through the nodes in the block from the lowest numbered node to the highest and grouping them into blocks of capacity $\mathtt{b}_i$. Just like in the centralized setting, each block either consists of a single node with at least $\mathtt{b}_i$ edges in E' or with a block of bounded density $O(\mathtt{b}_i)$.

Turning to the distributed implementation, we consider the i-phase for $i \geq 1$, where we are given the $(i-1)$-level of the partitioning $B_{i-1,1}, \ldots, B_{i-1,\ell}$ such that each node know the ID of the $(i-1)$-level block $B_{i,j}$ to which it belongs and the LCA of its block in T (knowing the LCA is important for communicating in the tree $T_{B_{i-1,j}}$). Equipped with that knowledge, we now describe how to compute the i^{th}-level of the partitioning. We will work on each block separately but simultaneously. That is, in phase i, we run (simultaneously) a collection of ℓ algorithms, one per $(i-1)$-level block $B_{i-1,j}$. From this point on, we focus on one such $(i-1)$-block $B = B_{i-1,j}$ and describe Algorithm PartitionSingleBlock which partitions B into $O(n^\epsilon)$ blocks, each with bounded density $\mathtt{b}_i$.

The output of this algorithm is that each node u knows the block IDs and the LCAs of all its blocks in each of $O(1/\epsilon)$ levels of the hierarchy. Throughout, the density of a subset $S \subseteq V$ is the number of edges in E' that have at least one endpoint in S. We say that a block in level i is *full* if its density is at least $\mathtt{b}_i$. We need some notation. Almost all our computation for a block B are restricted to the subtree in T that connects all B nodes, namely, the subtree $T_B = \bigcup_{u \in B} \pi(u, \ell, T)$ where ℓ is the LCA of B in T. Note that $V(T_B)$ is not necessarily B, that is, T_B might contain nodes that belong to different blocks than B. For $u \in B$, $b_i(u)$ is the ID of the i-level block of u. For $u \in T_B$ (which might not necessarily be in B), let $\max_B(u)$ be the node of maximum ID in $T_B(u) \cap B$ (if $u \in B$, then $\max_B(u) = u$). Finally, let $b_i^*(u)$ be the i-level block ID of $\max_B(u)$. We work from the leaf nodes of T_B up the root (i.e., the LCA of B-nodes). Letting D_B be the diameter of the tree T_B, at step $j \geq 0$, we assume that all nodes u in layers $D_B - j$ of T_B hold the following information:

- the block ID $b_i^*(u)$,
- the LCA of the block $b_i^*(u)$,
- the density of the block $b_i^*(u)$,
- the number of i-level blocks in $T_B(u) \cap B$[9].

These values are trivially computed for all leaf nodes in level D_B of T_B. We now show how to compute this information in level $D_B - (j+1)$. All nodes in layer $D - j$ send their block information to their parents in T_B. Consider now a parent u (in layer $D_B - (j+1)$) and its children in $T_B \cap B$: $v_1, \ldots, v_k$, ordered from left to right. The node u looks for a maximal sequence of children (from left to right) each with *only one* i-level block in their subtree $T_B(v_p) \cap B$. For any such maximal sequence, it merges these i-level blocks greedily, by filling up blocks as long as the density of the block is below $\mathtt{b}_i$. Once all mergings of i-level blocks of u's children is done, u can obtain all desired information. If $u \in B$, and the last i-level block of its rightmost child is not completely full (i.e., its density is below $\mathtt{b}_i$), then u joins that block, and otherwise, it opens a new block and the ID of the block is simply the ID of u.

Hence, it is easy to see that u has all the required information as it can easily get the density of the block to which it belongs and compute the number of block in $T_B(u) \cap B$. This completes the description of phase i. After D_B rounds of working bottom up on T_B, we start working top-down. Here, there is no need to work layer by layer, but rather each node u that just merged the blocks of its children into a full block (with density at least $\mathtt{b}_i$), sends all the information about that block (ID, LCA which is node u) to all its relevant children (whose block got merged) and this information propagates down their tree $T_B(v_p)$. Since we only merge subtrees that contain a single i-level block, there is no congestion in propagating the information of the full block down the tree. This completes the description of Algorithm PartitionSingleBlock.

We analyze the partitioning algorithm and show:

Claim 13. *Algorithm* PartitionSingleBlock *partitions an* $(i-1)$*-block* $B_{i-1,j}$ *into* $O(n^\epsilon)$ i*-level blocks.*

Proof. Let $B' = B_{i-1,j}$ and $T' = T_{B'}$ be the subtree of T connecting all nodes of B'. Let D' be the depth of T'. For node $u \in T'$, let $d(u, B') = \sum_{v \in (T'(u) \cap B')} \deg(v, E')$.

[9] In fact it is sufficient to know if there is exactly one block in $T_B(u)$ or more.

The blocks are defined bottom up on T'. We will claim by induction on the number of round $r \geq 0$, that for each node $u \in V(T')$ in layer $D' - r$ (in T') the following holds: If $T'(u) \cap B'$ (the subtree of u in T') has more than one block of level i , then for any i-level block in $T'(u)$ with density less than $\mathtt{b}_i$, either the i-level block before it or after it in $T'(u) \cap B'$ has density at least $\mathtt{b}_i$. As a result, we will get that that the number of i-level blocks in $T(u) \cap B'$ is at most $d(u, B')/\mathtt{b}_i$. Applying this to the root of T' (the LCA of the nodes of B'), we get that there are at most $\deg(B', E')/\mathtt{b}_i$. Since B' is a block in level $i-1$, we get that $\deg(B', E') \leq m/n^{\epsilon \cdot (i-1)}$ and hence, there are at most n^{ϵ} i-level blocks.

For $r = 0$, the claim holds vacuously. Assume that it holds up to round r and consider round $r+1$. Let u be a node in layer $D' - r + 1$ and let $v_1, \ldots, v_\ell$ be its children that belongs to B' in layer $D - r$. First, consider all v_j with at least two i-level blocks and such that their last block has density less than $\mathtt{b}_i$. By induction, we know that each such low-density block has a block of density at least $\mathtt{b}_i$ before it. All v_j with a single i-level block in $T'(v_j) \cap B'$ are merged until there is at most one low-density block. We note that if in $T'(v_j)$, there are at least two i-level blocks, it implies that either the block right before the block of v_j has density of at least $\mathtt{b}$, or that $v_j \in B'$ and $\deg(v_j, E') \geq \mathtt{b}$). Finally, if the i-level block of v_ℓ has low-density, $u \in B'$ and $\deg(u, E') \leq \mathtt{b}_i$, then the number of low-density blocks is not increased and hence the claim holds by induction assumption. □

We turn to bound the round complexity of the hierarchical partitioning.

Claim 14. *The hierarchical block partitioning can be computed in $O(D)$ rounds.*

The algorithm for computing the hierarchical block partitioning has $O(1/\epsilon)$ steps, where in each step i, we run Algorithm PartitionSingleBlock for *each* $(i-1)$-level block $B_{i-1,j}$ simultaneously. It is easy to a single application of Algorithm PartitionSingleBlock for one block $B_{i-1,j}$ in level $i-1$ takes $O(D)$ rounds. The next lemma proves Claim 14 by showing that since Algorithm PartitionSingleBlock only communicates on the tree edges $T_{B_{i-1,j}}$, we can run this algorithm for all the blocks in level $i-1$ simultaneously (at the cost of increasing the number of rounds by factor 2). The next lemma is quite general and will be useful later, since most of the computation that we do for blocks B uses only the tree edges T_B.

Claim 15. *Consider the collection of blocks in the i level of the partitioning: $B_1, \ldots, B_k$ and $\mathcal{A}_1, \ldots, \mathcal{A}_k$ be a set of algorithms, each takes at most R rounds, and sends messages only on the edges of T_{B_j} respectively. Then, all algorithms $\mathcal{A}_1, \ldots, \mathcal{A}_k$ for each of the blocks can run simultaneously in $2R$ rounds, in the* CONGEST *model.*

Proof. By Claim 3, each edge $e = (x, y)$ can appear in at most 2 different blocks B_i, B_j. Hence, the congestion when working on all blocks simultaneously is increased by factor of at most 2 when compared to working on a single block. □

Tool (II): Edge Disjoint Matching inside a Block. Our next tool uses again Algorithm TreeEdgeDisjointPath from [Pel00] (see Lemma 4.3.2 within). Recall that $T_{B_i} = \bigcup_{u \in B_i} \pi(\ell_i, u)$ is the subtree of T that connects all nodes of B_i where ℓ is the LCA of the nodes in B_i. In the *Block Matching Problem* we are given a block B_i in the partitioning, a subset of marked nodes $M_i \subseteq B_i$ and a spanning tree T. The goal is to match nodes in M_i (but at most one) into pairs, such that the (unique) tree path in T_{B_i} between each pair are all *edge disjoint*. The distributed output of the

problem is that each edge on the path $P_{x,y}$ between a matching pair $\langle x, y\rangle$ for $x, y \in M_i$ knows the endpoints of the path, namely, x and y, and in addition, each node on the path knows the next hops towards x and y respectively.

Algorithm DistBlockMatching works in $O(\mathsf{Depth}(T))$ rounds. In the first phase which works from the leafs up, it computes the matching and in the second phase, the edges of the paths between the matched nodes are getting marked. We start by explaining the first phase. We keep the invariant that in round $\mathsf{Depth}(T) - i$, every node u at layer i in T_{B_i}, keeps at most one ID of a node in M_i which is not yet matched. In every round $i \geq 0$, each node v sends the ID of at most one unmatched node in $T_{B_i}(v)$ to its parent u. A node u in layer $\mathsf{Depth}(T) - i + 1$ that received more than one ID of marked nodes in M_i, arbitrary match all but at most one of the received nodes into pairs. In D rounds, all (but at most one) nodes of M_i get matched.

In the second phase, for every matched pair $\langle x, y\rangle$, we mark the edges on its path $P_{x,y}$. Let v be the node that matched x and y in the first phase. We now let v send this matching outcome back to its children from which it got these IDs (i.e., it sends the pair $\langle x, y\rangle$ to its children that send it x or y). A node receiving this pair continues to send it to its children who sent it x or y. This message backtracks until it gets to x and y. It is easy to see that all this informed edges (that got an $\langle x, y\rangle$ message) are precisely the edges on the $x - y$ path in T. Moreover, all these computed paths are edge disjoint and the messages containing the matching result traverses *only* the edges belonging to the path between the matched pair. If the number of nodes in M_i is not even, then the last unmatched node is matched to an arbitrary (already matched) node. This increases the congestion on the path edges by at most 1. By using Claim 15 we have:

Claim 16. *(I) The Block Matching problem can be solved in $O(\mathsf{Depth}(T))$ rounds.*
(II) The Block Matching problem can be solved for all blocks *simultaneously in $O(\mathsf{Depth}(T))$ rounds.*
(III) On each edge $e \in G$, $O(1)$ messages are sent on e in total (throughout the entire execution).

Algorithm DistNonTreeCover**.** As in our centralized construction, we first consider the non-tree edges $E' = E(G) \setminus E(T)$ and construct a cycle cover $\mathcal{C}_1$ for these edges.

We begin with an high level description. The algorithm has $\ell = c \cdot \lceil 1/\epsilon \rceil$ phases. During each phase, some virtual edges $\widetilde{E}$ are added to the set of edges E' that we wish to cover. For clarity, in our description we treat the virtual edges as standard edges in G. Informally speaking, whenever the algorithm adds a virtual edge between two nodes u and v, it implies that the algorithm has already computed an u-v path, and the virtual edge (u, v) indicates the need for computing another u-v path so that we will endup with a cycle. In other words, adding a virtual edge means that we deffer the closure of the cycle to the future iterations. These cycles get completed only at a point where the congestion (or overlap) between cycles can be bounded.

In phase i, we are given the set of non-tree edges E' in G and a subset $\widetilde{E}_i$ of virtual edges that should be covered by cycles. The algorithm assumes at that phase that all the edges in $E' \cup \widetilde{E}_i$ between nodes belonging to *different* j-level blocks for some $j \leq i - 1$ are already taken care of (in a way that becomes clear later). It then restricts attention to cover the edges (u, v) in $E' \cup \widetilde{E}_i$ between nodes u and v that belong to the same block in all first $(i - 1)$-levels of the hierarchy, but belong to different blocks in level i. To handle these edges we will have $n^{2\epsilon}$ applications of Algorithm DistBlockMatching for each pair of i level blocks B_x and B_y which belong to the same block in level $i-1$. The subset of marked nodes in each application of Algorithm DistBlockMatching depends on the edges in $E' \cup \widetilde{E}_i$ that connect these pair of blocks. Based on the output matching of Algorithm DistBlockMatching, the algorithm defines a new set of virtual edges which are internal

to the i-level blocks. After $\ell = c \cdot \lceil 1/\epsilon \rceil$ phases, the remaining edges in $E' \cup \widetilde{E}_\ell$ are between nodes that belong to the same ℓ-level block. As will be shown in the analysis section, at that point the total number of edges $E' \cup \widetilde{E}_\ell$ that have both endpoint in the same ℓ-level block is $O(1)$. Hence, the algorithm covers these remaining edges by taking their fundamental cycles in T .

We explain the procedure in more details. The algorithm starts by computing $\ell = c \cdot \lceil 1/\epsilon \rceil$ levels of block hierarchies (with respect to the input E' of non-tree edges). By letting all nodes exchange their information on their ℓ blocks with their neighbors, all nodes know which of the E'-edges should be considered in a given phase i. (The knowledge about the virtual edges is acquired at the point of their creation). Initially, let $\widetilde{E}_1 = \emptyset$ be the (empty) set of virtual edges of phase 1.

We now focus on phase $i \geq 1$ and on one particular $(i-1)$-level block B that is partitioned (in level i of the hierarchy) into $k = \lceil n^\epsilon \rceil$ blocks $\mathcal{B} = \{B_1, \ldots, B_k\}$. We assume that we have in addition to E' a collection of virtual edges $\widetilde{E}_i$ and that each of the endpoints of a virtual edge in $\widetilde{E}_i$ knows about its second endpoint. Let $\mathcal{E}_i = E' \cup \widetilde{E}_i$. The goal of each block $B_x \in \mathcal{B}$ at that phase is to take care of all its edges in $E' \cup \widetilde{E}_i$ whose second endpoint is in B_y for every $B_y \in \mathcal{B} \setminus B_x$. To take care of these k different types of edges, we run (in B_x) k applications of Algorithm DistBlockMatching simultaneously.

Consider a pair of blocks $B_x, B_y \in \mathcal{B}$, and let $E_{x,y}$ be the edges in $\mathcal{E}_i$ with one endpoint in B_x and one endpoint in B_y. We divide $E_{x,y}$ into two subsets. Let $V_{x,y} \subseteq B_x$ be the nodes in B_x that have at least two incident edges in $E_{x,y}$ and let $E^1_{x,y}$ be the subset of edges in $E_{x,y}$ incident to the nodes of $V_{x,y}$. The set of these $E^1_{x,y}$ edges are handled locally with a single round of communication. We let each node $u \in V_{x,y}$, compute an arbitrary matching between its $E_{x,y}$-neighbors in B_y. We then connect each of the matched pairs by a virtual edge to be added to $\widetilde{E}_{i+1}$. To do that, we simply let each $u \in V_{x,y}$ send to each of its $E_{x,y}$-neighbors in B_y the ID of their matched node (indicating to them their new virtual edge to this node).

From now on, we restrict attention to the set of $E^2_{x,y} = E_{x,y} \setminus E^1_{x,y}$ edges, in which each node in B_x has exactly one incident edge in $E_{x,y}$. Here we cannot decide locally which virtual edges (between B_y nodes) should be added. Alternatively, we define a subset of marked nodes $M_{x,y} \subseteq B_x$ as the nodes in B_x that have exactly one incident edge in $E_{x,y}$.

Case (I): $|M_{x,y}| = 1$**.** The algorithm covers the single edge between B_x and B_y in $E_{x,y}$ by taking its fundamental cycle in T and add it to $\mathcal{C}_1$.

Case (II): $|M_{x,y}| \geq 2$**.** First, if $|M_{x,y}|$ is odd, the root of T_{B_x} picks one edge (u, v) in $E_{x,y}$ incident to node in $M_{x,y}$, add the fundamental cycle of that edge to the cycle cover $\mathcal{C}_1$ (i.e., by marking the cycle edges) and omit u from $M_{x,y}$. From now on, assume that $|M_{x,y}|$ is *even*. The algorithm then applies Algorithm DistBlockMatching on B_x with the marked nodes $M_{x,y}$. This algorithm matches the nodes of $M_{x,y}$ into pairs and mark the edge disjoint paths between these paired nodes on T_{B_x}. This pairing of nodes in $M_{x,y}$ is now used to define the set of virtual edges between nodes of B_y. In particular, let $\langle a, b \rangle$ be a matched pair in $M_{x,y}$ and let $(a, a'), (b, b') \in E_{x,y}$ where $a', b' \in B_y$. The algorithm will connect a' and b' in B_y by a virtual edge to be added to $\widetilde{E}_{i+1}$. Recall that Algorithm DistBlockMatching computes an edge disjoint path $\pi(a, b, T_{B_x})$. If $(a, a'), (b, b')$ are true edges in G it is easy to see how to inform a' about its virtual endpoint b' and vice versa. In such a case, we also send the information of the edges (a, a') and (b, b') on the path $W_{a',b'} = (a', a) \circ \pi(a, b, T_{B_x}) \circ \pi(b, b')$. This way the edges on the path $P_{a',b'}$ of every virtual edge (a', b') added know the G-edges that this path (which will become later a cycle) covers. In case where one of the edges $(a, a'), (b, b')$ is virtual, say the edge (a, a'), the algorithm sends this information using the a-a' walk $W_{a,a'}$ computed in previous phases. These walks are shown to have

length at most $O(2^{1/\epsilon} \cdot D)$ and overlap of $O(n^{\epsilon})$ and hence this information passing on these walks can be done efficiently of all virtual edges simultaneously using the random delay approach. These details are further explained in Appendix B. For an illustration see Figure 17.

Overall, for each $B_x \in \mathcal{B}$, we run simultaneously[10] k applications of Algorithm DistBlockMatching with different subsets of marked nodes $M_{x,y}$ for every $B_y \in \mathcal{B}$. At the end of all these applications, there are new virtual edges connecting nodes that belong to the same block B_y. For each such virtual edge, we have computed a path in G between its endpoints. This completes the description of phase $i \leq \ell$.

In the last phase ℓ, we consider all edges $e = (u, v)$ in $E' \cup \widetilde{E}_\ell$ such that u and v belong to the same ℓ-level block. We connect these virtual edge endpoints by taking their paths in T. Since each virtual edge $(a, b) \in E' \cup \widetilde{E}_\ell$ already has a walk $W_{u,v}$ which is shown to be of length $O(2^{1/\epsilon} D)$, we get a cycle $P_{u,v} \circ \pi(u, v, T)$ which is added to the cycle collection. This cycle is not yet simple but by letting all edges on the cycle learn all the edges of the cycle (using the random delay approach), the cycle can be made simple locally (by internal computation at each node of the cycle) using the centralized algorithm of Section 4.1. In Claim 21 we show that making the cycle simple does not violate the covering property. Further implementation Details are provided in Appendix B. This completes the description of the main algorithm for covering non-tree edges. Let $\mathcal{C}_1$ be the output cycle collection.

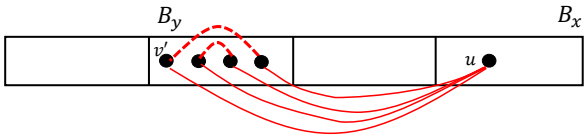

Figure 16: Dealing with nodes with more than one endpoint in a different block. The node u locally matched its neighbors in B_y which defines a new set of virtual edges to be covered.

6.1.2 Analysis of Algorithm DistNonTreeCover

Congestion and Dilation Analysis. We start by showing that each edge appears on $O(1/\epsilon \cdot n^{2\epsilon})$ cycles. Recall that $b_i(u)$ is the i-level block of u and that $\ell = c \cdot \lceil 1/\epsilon \rceil$. For every $i = \{1, \ldots, \ell\}$ and $u \in V$, let $\mathcal{E}_i(u) = \{(u, v) \in \mathcal{E}_i \mid b_{i-1}(u) = b_{i-1}(v)\}$ be the set of all u edges in $\mathcal{E}_i$ that are internal to its $(i-1)$-level block. Note that Algorithm DistNonTreeCover, in phase i, considers only edges belonging to $\bigcup_u \mathcal{E}_i(u)$.

Claim 17. *The number of edges in $\mathcal{E}_\ell$ that have both endpoints in a given ℓ-level block B is $O(1)$.*

[10] When saying *simultaneously* we mean using the random delay approach.

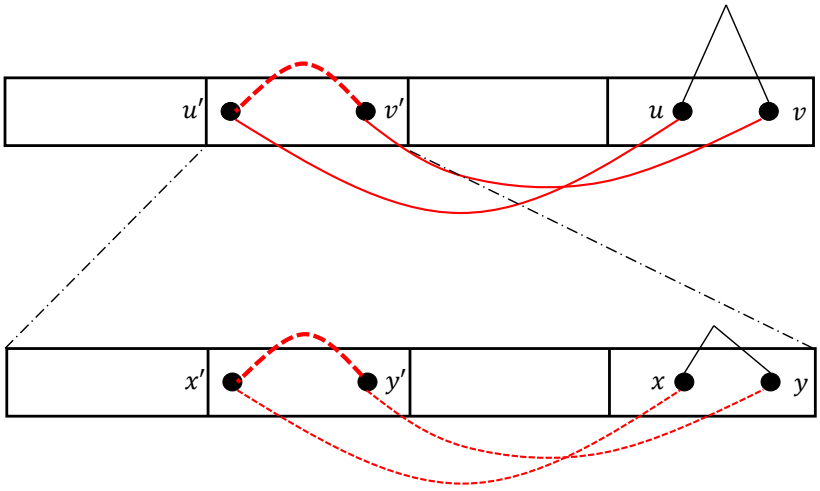

Figure 17: Illustration of the hierarchical block partitioning and the addition of virtual edges. Top: In phase 1, we have n^ϵ blocks. Zoom into two blocks B_1 and B_2. The pair $\langle u, v\rangle$ got matched in B_1 and hence the virtual edge is added in B_2. Bottom: In phase 2, we consider only internal edges in 2-level blocks. Zoom into B'_1 and B'_2 which were part of the same block in level 1. The matched pair $\langle x, y\rangle$ has two virtual edges to x' and y' respectively and define a new virtual edge between x' and y'. This define a path $P_{x',y'}$ obtained by concatenating the path $P_{x,x'}$, $P_{y,y'}$ and $\pi(x, y, T_{B'_1})$ computed in the tree of B'_1.

Proof. For every $i = \{1, \ldots, \ell\}$ and $u \in V$, let $\deg_i(u) = |\mathcal{E}_i(u)|$. We now claim that $\deg_i(u) \geq \deg_{i+1}(u)$ for every $i = \{1, \ldots, \ell\}$. To see this, observe that each edge $(u, v) \in \mathcal{E}_i(u) \setminus \mathcal{E}_{i+1}(u)$ (i.e., in this case, $b_{i-1}(u) = b_{i-1}(v)$ but $b_i(u) \neq b_i(v)$) leads to the introduction of at most one virtual edge $(u, v') \in \mathcal{E}_{i+1}(u) \setminus \mathcal{E}_i(u)$ where $b_i(u) = b_i(v')$. In addition, all the virtual edges added to $\mathcal{E}_{i+1}(u) \setminus \mathcal{E}_i(u)$ are due to edges in $\mathcal{E}_i(u) \setminus \mathcal{E}_{i+1}(u)$. Combining these two observations, we get that $\deg_{i+1}(u) \leq \deg_i(u)$.

We therefore have that for every $u \in V$, $\deg_1(u) \geq \deg_\ell(u)$. By definition

$$\mathcal{E}_1(u) = \{(u, v) \mid (u, v) \in E'\}$$

and hence $\deg_1(u) = \deg(u, E')$. Let B' be an $(\ell - 1)$ level block such that $B \subseteq B'$. By the construction of the $(\ell - 1)$ level block B', we get that $\sum_{u \in B'} \deg_\ell(u) \leq \mathtt{b}_{\ell-1} = O(1)$. In particular, since $B \subseteq B'$, it also holds that $\sum_{u \in B} \deg_\ell(u) = O(1)$, since $\deg_\ell(u)$ is the number of u edges in $\mathcal{E}_\ell$ whose second endpoint is in the same $(\ell - 1)$-level block, it also holds that number of u edges that have both endpoints in its ℓ-level block is bounded by $O(1)$. □

Let $\mathcal{C}'_i$ be the collection of fundamental cycles added at phase i. Recall that we cover an edge $e \in \mathcal{E}_i$ by its fundamental cycle in T if the edge e is the only edge in that set the connects two brother blocks B_x and B_y.

Claim 18. *Every edge appears on at most $O(n^{2\epsilon})$ cycles in $\mathcal{C}'_i$.*

Proof. Let $B_1, \ldots, B_k$ be the collection of all $(i - 1)$-level blocks. We now consider a particular block B_j which is partitioned into $B_{j,1}, \ldots, B_{j,\ell}$ blocks in level i where $\ell = O(n^\epsilon)$. For such B_j we

add at most $O(\ell^2)$ fundamental cycles to $\mathcal{C}'_i$ (one per pair of blocks in $B_{j,1}, \ldots, B_{j,\ell}$). Note that all these cycles are edges in T_{B_j}. By Claim 3, every edge e appears on at the trees of at most two $(i-1)$-blocks, hence overall each edge appears in $O(n^{2\epsilon})$ cycles. □

We now turn to consider cycles that are computed using the addition of the virtual edges.

Claim 19. *For each virtual edge (u, v) added in phase i, the algorithm computes a u-v walk $W_{u,v}$ of length at most $2^i \cdot D$. In addition, this walk "covers" $O(2^i)$ edges in E'.*

Proof. Both claims are shown by induction on i. For $i = 1$, when adding a virtual edge (u, v), we have two edges (u, u') and (v, v') in E' such that u, v (resp., u', v') are in the same 1-level block. Using Algorithm DistBlockMatching, we have marked the tree path between u' and v' and hence we have a path $W_{u,v} = (u, u') \circ \pi(u', v', T) \circ (v', v)$. Clearly, $|W_{u,v}| \leq 2D + 2$. This path covers exactly two edges (u, u') and (v, v') which proves the claims for the induction base.

Assume that the claims hold up to phase $i-1$ and consider phase i. Here, when adding a virtual edge (u, v) in phase i, same story holds as in the induction phase only that the two edges (u, u') and (v, v') might be virtual edges that were added in previous phases. The u-v walk is given by $W_{u,v} = W_{u,u'} \circ \pi(u', v', T) \circ W_{v',v}$ where $W_{u,u'}, W_{v',v}$ is either an edge in G or a walk computed in previous phases. Using the induction assumption, we get that the length of the walk $W_{u,v}$ at most $2 \cdot 2^{i-1} \cdot D + 2D \leq 2^i \cdot D$. In addition, if (u, u') is a virtual edge, then by induction assumption, $W_{u,u'}$ covers at most 2^{i-1} edges in E' and same goes to $W_{v,v'}$. If (u, u') is an edge in E' than it covers only itself (same for (v, v')). Overall the walk $W_{u,v}$ covers 2^i edges. The claim follows. □

Let $\mathcal{W}_i = \{W_{u,v} \mid (u, v) \in \mathcal{E}_{i+1} \setminus \mathcal{E}_i\}$ be the collection of all u-v walks between all virtual edges (u, v) added in phase i. Let $\mathcal{W} = \bigcup_{i=1}^{\ell} \mathcal{W}_i$ be the collection of all these walks.

Claim 20. *For every i, each edge $e \in G$ appears at most $O(i \cdot n^\epsilon)$ in total on all walks of $\mathcal{W}_i$ (this takes into account the multiple appearance of the edge on the same walk). Hence, overall, each edge e appears $O((1/\epsilon)^2 \cdot n^\epsilon)$ time on all walks of $\mathcal{W}$.*

Proof. Recall that we add a virtual edge (u, v) in phase $i \geq 1$ for $u, v \in B_y$, in case where there are edges $(u, u'), (v, v')$ in $\mathcal{E}_i$ such that u', v' are in B_x and B_x and B_y are brothers (i.e., belong to the same block in level $i-1$). Note that for $i \geq 2$, the edges $(u, u'), (v, v')$ might be virtual. In any case, we have have u-v walk $W_{u,v} = W_{u,u'} \circ \pi(u', v', T) \circ W_{v',v}$ where in phase 1, $W_{u,u'}, W_{v,v'}$ are simply the edges (u, u') and (v, v').

We prove the claim by induction on i. For the base of the induction, consider $i = 1$. In this case, all walks $W_{u,v}$ for the virtual edges added at phase 1 are in fact paths. Thus, we only need to bound the number of an edge e on different such paths. Since we have n^ϵ applications of Algorithm PartitionSingleBlock in each 1-level block, and using Claim 15, we get that each edge e appears on $O(n^\epsilon)$ different paths in $\mathcal{W}_1$.

Assume that the claim holds up to phase $i-1$ and consider phase i. Here, we can apply the induction assumption for $W_{u,u'}$ and $W_{v',v}$ (it is sufficient to consider the case where $(u, u'), (v, v')$ are virtual edges added in previous phases, as this is the interesting case). The walk $W_{u,v}$ has two types of segments: (I) walks connecting virtual edges that were added in phase $j \leq i-1$, namely, $W_{u,u'}, W_{v,v'}$ and (II) tree segment $\pi(u, v)$. By the same reasoning as in the base of the induction, each edge e can appear on at most $O(n^\epsilon)$ tree paths $\pi(u, v)$ for every virtual edge (u, v) added in phase i. By the induction assumption, each edge e appears at most $(i-1)n^\epsilon$ times on the collection of walks in $\mathcal{W}_j$ for $j \leq i-1$. Hence, overall an edge e appears $i \cdot n^\epsilon$ many time on the walks of $\mathcal{W}_i$. The claim holds. □

We therefore have:

Corollary 1. *Every edge appears on $O(n^{2\epsilon})$ cycles in the final cycle collection. Each cycle has length $O(2^{1/\epsilon} \cdot D)$ and covers $O(2^{1/\epsilon})$ many non-tree edges.*

Covering Analysis of Algorithm DistNonTreeCover**.** Note that the cycles computed are not necessarily simple. We next claim that they can be made simple without loosing the covering property. Recall that in each phase i, for every virtual edge (u, v) added in phase i we compute a walk $W_{u,v}$ in G.

Claim 21. *For every virtual edge (u, v), any edge that appears more than once on the walk $W_{u,v}$ is a T-edge.*

Proof. The proof is shown by induction on the phase number i. For $i = 1$, there exists two edges (u, u') and (v, v') in E' such that u, v (resp., u', v') are in the same 1-level block and u', v' have been paired in their block by Algorithm DistBlockMatching. In this case, $W_{u,v} = (u, u') \circ \pi(u', v', T) \circ (v', v)$ which is in fact a path!.

Assume that the claim holds up to phase $i-1$ and consider phase i. For any virtual edge (u, v) added in phase i, there exists two edges (u, u') and (v, v') in $\mathcal{E}_i$ such that u, v (resp., u', v') are in the same i-level block and u', v' have been paired in their block by Algorithm DistBlockMatching. The only difference to the induction base is that now the edges (u, u') and (v, v') might be virtual edges added in some phase $j \leq i - 1$. In either case $W_{u,u'}$ is either an edge (u, u') are of walk for which the induction assumption can be applied. The walk $W_{u,v} = W_{u,u'} \circ \pi(u', v', T) \circ W_{v',v}$ adds a tree segment $\pi(u', v', T)$ to the existing walks $W_{u,u'}, W_{v',v}$. Hence, it can only increase the appearance of the tree edges. Combining with the induction assumption, the claim holds. □

Claim 22. *Every edge $e' \in E'$ is covered by the cycles of Algorithm* DistNonTreeCover*.*

Proof. By Claim 21, it remains to show that for every edge $e' \in E'$, there is a walk $W_{u,v}$ such that e' appears on it. Let $e' = (x, y)$ and define i to be the first index such that x and y belong to different i-level block. The edge e' is then considered in phase i of the algorithm. If the algorithm did not cover e' by taking its fundamental cycle, it implies that i is not the last phase and necessarily some virtual edge was added due to $e' = (w, z)$. By construction, e' connects two nodes in the same i-level block and hence will be handled in phase $j \geq i + 1$ and in addition, (x, y) appears on the walk $W_{w,z}$.

We can continue with the argument with the virtual edge (w, z) which is handled only at later phase j, claiming that the edge (x, y) appears when another virtual edge (w', z') is added when considering (w, z) and hence (x, y) appears on the walk $W_{w',z'}$. This continues until we get to a point where the fundamental cycle of some virtual edge (a, b) such that $(x, y) \in W_{a,b}$ is added and at that point, Claim 21 guarantees that when making the cycle $W_{a,b} \circ \pi(a, b, T)$ simple the edge (x, y) is covered. Recall that indeed in the last phase ℓ we handle all remaining edges (also the virtual onces) by taking their fundamental cycles. The claim follows. □

Round Complexity of Algorithm DistNonTreeCover**.**

Claim 23. *Algorithm* DistTreeCover *has round complexity $\widetilde{O}(1/\epsilon \cdot (2^{1/\epsilon} D + n^{2\epsilon}))$*

Proof. By Claim 14, the hierarchical block partitioning can be computed in $O(d)$ rounds. The algorithm has $O(1/\epsilon)$ phases. We show that each phase can be implemented can in $O(2^{1/\epsilon}D + n^{\epsilon})$ rounds. We have $O(n^{\epsilon})$ applications of Algorithm DistBlockMatching in each block B_x. Using the random delay approach and Claim 16 all the $O(n^{\epsilon})$ applications in a given block B_x can be done in $\widetilde{O}(n^{\epsilon} + D)$ rounds. Using Claim 15, this can be done for all blocks in $\widetilde{O}(n^{\epsilon} + D)$ rounds as well. We next bound the number of rounds required to form the virtual edges (a', b') and exchange information about the G edges that are covered by the $W_{a',b'}$ path. By Claims 19 and 20, using the random delay approach it can be done in $\widetilde{O}(n^{2\epsilon} + 2^{1/\epsilon}D)$ rounds. Finally, once the cycles are computed, using the random delay approach again, all edges can learn all the cycles that go through it in $\widetilde{O}(n^{2\epsilon} + 2^{1/\epsilon}D)$ round and locally make them simple. □

6.1.3 Distributed Algorithm for Covering Tree Edges

We turn to consider the remaining tree edges $E(T)$. Algorithm DistTreeCover essentially mimics the centralized construction of Section 4.1. Recall that $p(v)$ is the parent of v in the BFS tree T. A non-tree edge $e' = (u', v')$ is a swap edge for the tree edge $e = (p(v), v)$ if $e \in \pi(u', v')$, let $s(v) = v'$ by the endpoint of e' that is not in $T(v)$. By using the algorithm of Section 4.1 in [GP16], we can make every node v know $s(v)$ in $O(D)$ rounds.

A key part in the algorithm of Section 4.1 is the definition of the path $P_e = \pi(v, u') \circ (u', s(v))$ for every tree edge $e = (p(v), v)$. By computing swap edges using Section 4.1 in [GP16] all the edges of each P_e get marked.

Computing the set $I(T) \subseteq E(T)$. We next describe how to compute a maximal collection of tree edges $I = \{e_i\}$ whose paths P_{e_i} are edge disjoint and in addition for each edge $e_j \in E(T) \setminus I$ there exists an edge $e_i \in T'$ such that $e_j \in P_{e_i}$. To achieve this, we start working on the root towards the leaf. In every round $i \in \{1, \ldots, D\}$, we consider only *active* edges in layer i in T. Initially, all edges are active. An edge becomes inactive in a given round if it receives an inactivation message in any previous round. Each active edge in layer i, say e_j, initiates an inactivation message on its path P_{e_j}. An inactivation message of an edge e_j propagates on the path P_{e_j} round by round, making all the corresponding edges on it to become inactive.

Note that the paths P_{e_j} and $P_{e_{j'}}$ for two edges e_j and $e_{j'}$ in the same layer of the BFS tree, are edge disjoint and hence inactivation messages from different edges on the same layer do not interfere each other. We get that an edge in layer i active in round i only if it did not receive any prior inactivation message from any of its BFS ancestors. In addition, any edge that receives an inactivation message necessarily appears on a path of an active edge. It is easy to see that within D rounds, all active edges I on T satisfy the desired properties (i.e., their P_{e_i} paths cover the remaining T edges and these paths are edge disjoint).

Distributed Implementation of Algorithm TreeCover. First, we mark all the edges on the P_e paths for every $e \in I(T)$. As every node v with $e = (p(v), v)$ know its swap edge, it can send information along P_e and mark the edges on the path. Since each edge appears on the most two P_e paths, this can be done simultaneously for all $e \in I(T)$.

From this point on we follow the steps of Algorithm TreeCover. The partitioning of Appendix A can be done in $O(D)$ rounds as it only required nodes to count the number of nodes in their subtree. We define the ID of each tree T'_1, T'_2 to be the maximum edge ID in the tree (as the trees are edge disjoint, this is indeed an identifier for the tree). By passing information on the P_e paths, each node v can learn the tree ID of its swap endpoint $s(v)$. This allows to partition the edges of T'

into $E'_{x,y}$ for $x, y \in \{1, 2\}$. Consider now the i^{th} phase in the computation of cycle cover $\mathcal{C}_{1,2}$ for the edges $E'_{1,2}$.

Applying Algorithm TreeEdgeDisjointPath can be done in $O(D)$ round. At the end, each node v_j knows its matched pair v'_j and the edges on the tree path $\pi(v_j, v'_j, T'_1)$ are marked. Let Σ be the matched pairs. We now the virtual conflict graph G_Σ. Each pair $\langle v_j, v'_j \rangle \in \Sigma$ is simulated by the node of higher ID, say, v_j. We say that v_j is the *leader* of the pair $\langle v_j, v'_j \rangle \in \Sigma$. Next, each node v that got matched with v' activates the edges on its path $P_e \cap E(T'_1)$ for $e = (p(v), v)$. Since the π edges of the matched pairs are marked as well, every edge $e' \in \pi(v_k, v'_k, T'_1)$ that belongs to an active path P_e sends the ID of the edge e to the leader of the pair $\langle v_k, v'_k \rangle$. By Claim 6, every pair σ' interferes with at most one other pair and hence there is no congestion and a single message is sent along the edge-disjoint paths $\pi(v_j, v'_j, T'_1)$ for every $\langle v_j, v'_j \rangle \in \Sigma$. Overall, we get the the construction of the virtual graph can be done in $O(D)$ rounds.

We next claim that all leaders of two neighboring pairs $\sigma, \sigma' \in G_\Sigma$ can exchange $O(\log n)$ bits of information using $O(D)$ rounds. Hence, any r-round algorithm for the graph G_Σ can be simulated in T'_1 in $O(r \cdot D)$ rounds. To see this, consider two neighbors $\sigma = \langle x, y \rangle, \sigma' = \langle x', y' \rangle$ where σ' interferes σ. Without loss of generality, assume that the leader x' of σ' wants to send a message to the leader x of σ. First, x' sends the message on the path $\pi(x', y', T'_1)$. The edge $e' \in \pi(x', y', T'_1) \cap P_e$ for $e = (p(x), x)$ that receives this message sends it to the leader x along the path P_e. Since we only send messages along edge disjoint paths, there is no congestion and can be done in $O(D)$ rounds.

Since the graph G_Σ has arboricity $O(1)$, it can be colored with $O(1)$ colors and $O(\log n)$ rounds using the algorithm of [BE10]. By the above, simulating this algorithm in G takes $O(D \log n)$ rounds. We then consider each color class at a time where at step j we consider $\Sigma_{i,j}$. For every $\sigma = \langle x, y \rangle$, x sends the ID of $s(y)$ to $s(x)$ along the P_e path for $e = (p(x), x)$. In the same manner, y sends the ID of $s(x)$ to $s(y)$. This allows each node in T'_2 know its virtual edge. At that point we run Algorithm DistNonTreeCover to cover the virtual edges. Each virtual edge is later replaced with a true path in G in a straightforward manner.

Analysis of Algorithm DistTreeCover**.**

Claim 24. *Algorithm* DistTreeCover *computes a* $(2^{1/\epsilon}D, n^\epsilon \log^2 n)$ *cycle cover* $\mathcal{C}_2$ *for the tree edges* $E(T)$ *and has round complexity of* $O(2^{1/\epsilon}D \cdot n^\epsilon \cdot \log^2 n)$.

Proof. The correctness follows the same line of arguments as in the centralized construction (see the Analysis of Section 4.1.2), only the here we use Algorithm DistNonTreeCover. Each cycle computed by Algorithm DistNonTreeCover has length $O(2^{1/\epsilon}D)$ and the cycle covers $O(2^{1/\epsilon})$ non-tree edges. In our case, each non-tree edge is virtual and replaced by a path of length $O(D)$ hence the final cycle has still length $O(2^{1/\epsilon}D)$. With respect to congestion, we have $O(\log n)$ levels of recursion and in each level when working on the subtree T' we have $O(\log n)$ applications of Algorithm DistNonTreeCover which computes cycles with congestion $O(n^\epsilon)$. The total congestion is then bounded by $O(n^\epsilon \cdot \log^2 n)$.

We proceed with round complexity. The algorithm has $O(\log n)$ levels of recursion. In each level we work on edge disjoint trees simultaneously. Consider a tree T'. The partitioning into T'_1, T'_2 takes $O(D)$ rounds. We now have $O(\log n)$ phases. We show that each phase takes $O(2^{1/\epsilon}D \cdot n^\epsilon)$ rounds, which is the round complexity of Algorithm DistNonTreeCover. In particular, In phase i we have the following procedures. Applying Algorithm TreeEdgeDisjointPath in T'_1, T'_2 takes $O(D)$ rounds. The computation of the conflict graph G_Σ takes $O(D)$ rounds as well and coloring it using the coloring algorithm for low-arboricity graphs of [BE10] takes $O(D \log n)$ rounds. Then we apply

Algorithm DistNonTreeCover which takes $\widetilde{O}(2^{1/\epsilon} \cdot D + n^{2\epsilon})$ rounds. Translating the cycles into cycles in G takes $\widetilde{O}(2^{1/\epsilon} \cdot D + n^{2\epsilon})$ rounds. Overall, we have $\widetilde{O}(2^{1/\epsilon} \cdot D + n^{2\epsilon})$ rounds. □

Theorem 2 follows by combining Lemma 5 and Claims 23 and 24.

6.2 Additional Low Congestion Covers

Private Neighborhood Trees. We now show how to use the distributed construction of cycle covers to construct private neighborhood trees. The distribute output format of private neighborhood trees $\mathcal{N}$ is that each node u knows its parent in the spanning tree $T(v) \in \mathcal{N}$ for every $v \in V$.

Lemma 6. *There exists an $\widetilde{O}(4^{1/\epsilon} \cdot \Delta \cdot D + n^{\epsilon} \cdot D)$-round algorithm that computes an $(4^{1/\epsilon} \cdot D \cdot \Delta \cdot \log \Delta, n^{\epsilon} \cdot D \cdot \log^2 n \cdot \log \Delta)$ private neighborhood trees.*

Proof. We start by running Algorithm DistCycleCover which computes a $(\mathtt{d}, \mathtt{c})$ cycle cover $\mathcal{C}$ for $\mathtt{d} = O(4^{1/\epsilon} \cdot D)$ and $\mathtt{c} = O(n^{\epsilon} \cdot \log^2 n)$. By using $\widetilde{O}(\mathtt{d} + \mathtt{c})$ rounds, we can make each edge (u, v) know the edges of all the cycles it belongs to in $\mathcal{C}$. We mimic the centralized reduction to cycle cover. In this reduction, we have $O(\log \Delta)$ applications of Algorithm DistCycleCover on some virtual graph. Since a node v knows the cycles of its edges, it knows which virtual edges it should add in phase i. Simulating the virtual graph can be done with no extra congestion in G. In each phase i, we compute $(\mathtt{d}, \mathtt{c})$ cycle cover $\mathcal{C}_i$ in G obtained by computing the cycle cover in a virtual graph. By the same argument as in Claim 11, translating these cycles to cycles in G does not increase the congestion. Using $\widetilde{O}(\mathtt{d} + \mathtt{c})$ rounds, each edge e can learn all the edges on the cycles that pass through it appears in $\mathcal{C}_i$. At the last phase $\ell = O(\log \Delta)$, the graph $G_\ell(u_j)$ consists of $O(\log \Delta \cdot \Delta)$ cycles. In particular,

$$G_\ell(u_j) = \bigcup_{i=1}^{\ell} \{C \in \mathcal{C}_i \mid (u_j, v) \in C, v \in \Gamma(u_j, v)\}.$$

By the same argument of Claim 11, each edge e appears on $O(\log \Delta \cdot \log^2 n \cdot n^{\epsilon} \cdot D)$ different subgraphs $G_\ell(u_j)$ for $u_j \in V$. The diameter of each subgraph can $G_\ell(u_j)$ can be clearly bounded by the number of nodes it contained which is $O(4^{1/\epsilon} \cdot \log \Delta \cdot \Delta \cdot D)$. Since each edge e knows all cycles it appears on[11], it also knows all the graphs $G_\ell(u_j)$ to which it belongs. Computing a spanning tree in $G_\ell(u_i) \setminus \{u_i\}$ can be done in $\widetilde{O}(4^{1/\epsilon} \Delta \cdot D)$ rounds. Using the standard random delay approach, and using the fact that each edge appears on $O(\log \Delta \cdot \log^2 n \cdot n^{\epsilon} \cdot D)$, all the spanning trees in $G_\ell(u_j) \setminus \{u_j\}$ can be construct simultaneously in $\widetilde{O}(4^{1/\epsilon} \cdot \Delta \cdot D + n^{\epsilon} \cdot D)$ rounds. □

By combining our general compiler Theorem 4 with our distributed low-congestion cover of Lemma 6, we get:

Corollary 2 (Distributed PSM with no pre-processing)**.** *Let G be a 2-vertex connected n-vertex graph with diameter D and maximal degree Δ. Let $\mathcal{A}$ be a* natural *distributed algorithm that runs on G in r rounds. Then, $\mathcal{A}$ can be transformed to an equivalent algorithm $\mathcal{A}'$ with perfect privacy which runs in $\widetilde{O}(r \cdot 2^{\sqrt{\log n}} \cdot D \cdot \mathsf{poly}(\Delta))$*

[11] We say that an edge (u, v) knows a piece of information, if at least one of the edge endpoints know that.

6.3 Pre-processing for Improved Cover Structures

We next show how the covers of Theorems 1 and 3 with existentially optimal bounds can be constructed using $\widetilde{O}(n + D \cdot \Delta)$ rounds in the distributed setting. This pre-processing step should be done only once and any general compiler that will run in the network in the future will be able to use it (i.e., these structures should be computed once for a given network).

Lemma 7. *For every 2-edge connected n-vertex graph a $(D \log n, \log^3 n)$ cycle cover can be computed distributively in $\widetilde{O}(n)$ rounds of pre-processing.*

Proof. Compute a BFS tree T and consider the set of non-tree edges E'. Let $E_0 = E'$. As long that number of edges E_i to be covered in E' is at least $O(\log^c n \cdot n)$, we do as follows in phase i.

Let $\Delta_i = |E_i|/n$. We partition the edges of E_i into $\ell_i = \Delta_i/(c \cdot \log n)$ edge-disjoint subgraphs by letting each edge in E_i pick a number in $[1, \ell_i]$ uniformly at random. We have that w.h.p. each subgraph $E_{i,j}$ contains $\Theta(n \log n)$ edges of E_i.

At the point, we work on each subgraph $E_{i,j}$ independently. We compute a BFS tree $T_{i,j}$ in each $E_{i,j}$ (using only communication on $E_{i,j}$ edges). We then collect all edges of $E_{i,j}$ to the root by pipelining these edges on $T_{i,j}$. At that point, each root of $T_{i,j}$ can partition all but $2n$ edges of $E_{i,j}$ into edge disjoint cycles of length $O(\log n)$. The root also pass these cycle information to the relevant edges using the communication on $T_{i,j}$. Note that since the $E_{i,j}$ subgraphs are disjoint, this can be done simultaneously for all subgraphs $E_{i,j}$. At the end of that phase, we are left with $2n \cdot \ell_i = O(|E_i|/\log n)$ uncovered edges E_{i+1} to be handled in the next phase. Overall, after $O(\log n/\log\log n)$ phases, we are left with $O(n \log n)$ uncovered edges. At the point, we can pipeline these edges to the root of the BFS tree, along with the $n-1$ edges of the BFS tree and let the root compute it locally as explained in Section 4.1. The lemma follows. □

Combining this with Lemma 6 we get:

Corollary 3. *For every biconnected n-vertex graph $G = (V, E)$ with diameter D and maximum degree Δ, an $O(D \cdot \Delta \cdot \log n, D \cdot \log^3 n \cdot \log \Delta)$ private neighborhood tree collection $\mathcal{N}$ can be constructed using $\widetilde{O}(n + D \cdot \Delta)$ rounds.*

7 Discussion and Future Work

In this paper we introduce a new framework for secure distributed graph algorithms. We present the construction of a general compiler that can turn any natural non-secure distributed algorithm to a secure one, while increasing the round complexity by factor $\widetilde{O}(D \cdot \Delta)$. There are many remaining interesting research directions.

First, we note that our protocols are secure against an adversary that controls a *single* node in the graph. To see this, consider an adversary that gets both $\widehat{\sigma}_i$ and R_{σ_i}, decrypts the state and learn additional information.

An additional research direction involves the distribution computation of $(\widetilde{O}(D), \widetilde{O}(1))$ cycle covers in sublinear number of rounds. Our current efficient distributed algorithm obtains $(\widetilde{O}(4^{1/\epsilon} \cdot D), \widetilde{O}(n^{\epsilon}))$ cycle covers in $O(4^{1/\epsilon} \cdot D + n^{\epsilon})$ rounds. We note that computing $(\widetilde{O}(D), \widetilde{O}(1))$ cycle covers boils into the following problem:

> *Given a graph $G = (V, E)$ compute (in the* CONGEST *model) a maximal collection of edge disjoint $O(\log n)$-length cycles.*

Note that in the LOCAL model, this problem is easy, and the main challenges comes from the bandwidth limitations of the CONGEST model.

Acknowledgments

We thank Benny Applebaum, Uri Feige, Moni Naor and David Peleg for fruitful discussions concerning the nature of distributed algorithms, cycle covers and secure protocols.

References

[ABCP96] Baruch Awerbuch, Bonnie Berger, Lenore Cowen, and David Peleg. Fast distributed network decompositions and covers. *Journal of Parallel and Distributed Computing*, 39(2):105–114, 1996.

[ABCP98] Baruch Awerbuch, Bonnie Berger, Lenore Cowen, and David Peleg. Near-linear time construction of sparse neighborhood covers. *SIAM Journal on Computing*, 28(1):263–277, 1998.

[AGLP89] Baruch Awerbuch, Andrew V. Goldberg, Michael Luby, and Serge A. Plotkin. Network decomposition and locality in distributed computation. In *30th Annual Symposium on Foundations of Computer Science, Research Triangle Park, North Carolina, USA, 30 October - 1 November 1989*, pages 364–369, 1989.

[BE10] Leonid Barenboim and Michael Elkin. Sublogarithmic distributed mis algorithm for sparse graphs using nash-williams decomposition. *Distributed Computing*, 22(5-6):363–379, 2010.

[BE13] Leonid Barenboim and Michael Elkin. Distributed graph coloring: Fundamentals and recent developments. *Synthesis Lectures on Distributed Computing Theory*, 4(1):1–171, 2013.

[BEK14] Leonid Barenboim, Michael Elkin, and Fabian Kuhn. Distributed (δ+1)-coloring in linear (in δ) time. *SIAM Journal on Computing*, 43(1):72–95, 2014.

[BEPS16] Leonid Barenboim, Michael Elkin, Seth Pettie, and Johannes Schneider. The locality of distributed symmetry breaking. *Journal of the ACM (JACM)*, 63(3):20, 2016.

[BFH+16] Sebastian Brandt, Orr Fischer, Juho Hirvonen, Barbara Keller, Tuomo Lempiäinen, Joel Rybicki, Jukka Suomela, and Jara Uitto. A lower bound for the distributed lovász local lemma. In *Proceedings of the forty-eighth annual ACM symposium on Theory of Computing*, pages 479–488. ACM, 2016.

[BGI+14] Amos Beimel, Ariel Gabizon, Yuval Ishai, Eyal Kushilevitz, Sigurd Meldgaard, and Anat Paskin-Cherniavsky. Non-interactive secure multiparty computation. In *Advances in Cryptology - CRYPTO 2014 - 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part II*, pages 387–404, 2014.

[BGT13] Elette Boyle, Shafi Goldwasser, and Stefano Tessaro. Communication locality in secure multi-party computation - how to run sublinear algorithms in a distributed setting. In *Theory of Cryptography - 10th Theory of Cryptography Conference, TCC 2013, Tokyo, Japan, March 3-6, 2013. Proceedings*, pages 356–376, 2013.

[BGW88] Michael Ben-Or, Shafi Goldwasser, and Avi Wigderson. Completeness theorems for non-cryptographic fault-tolerant distributed computation (extended abstract). In *Proceedings of the 20th Annual ACM Symposium on Theory of Computing, May 2-4, 1988, Chicago, Illinois, USA*, pages 1–10, 1988.

[BIPW17] Elette Boyle, Yuval Ishai, Rafael Pass, and Mary Wootters. Can we access a database both locally and privately? In *Theory of Cryptography - 15th International Conference, TCC 2017, Baltimore, MD, USA, November 12-15, 2017, Proceedings, Part II*, pages 662–693, 2017.

[BLO16] Aner Ben-Efraim, Yehuda Lindell, and Eran Omri. Optimizing semi-honest secure multiparty computation for the internet. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security, Vienna, Austria, October 24-28, 2016*, pages 578–590, 2016.

[BNP08] Assaf Ben-David, Noam Nisan, and Benny Pinkas. FairplayMP: a system for secure multi-party computation. In *Proceedings of the 2008 ACM Conference on Computer and Communications Security, CCS 2008, Alexandria, Virginia, USA, October 27-31, 2008*, pages 257–266, 2008.

[Bol04] Béla Bollobás. *Extremal graph theory*. Courier Corporation, 2004.

[CCD88] David Chaum, Claude Crépeau, and Ivan Damgård. Multiparty unconditionally secure protocols (extended abstract). In *Proceedings of the 20th Annual ACM Symposium on Theory of Computing, May 2-4, 1988, Chicago, Illinois, USA*, pages 11–19, 1988.

[CCG+14] Nishanth Chandran, Wutichai Chongchitmate, Juan A. Garay, Shafi Goldwasser, Rafail Ostrovsky, and Vassilis Zikas. Optimally resilient and adaptively secure multi-party computation with low communication locality. *IACR Cryptology ePrint Archive*, 2014:615, 2014.

[CHL+17] Yi-Jun Chang, Qizheng He, Wenzheng Li, Seth Pettie, and Jara Uitto. The complexity of distributed edge coloring with small palettes. *arXiv preprint arXiv:1708.04290*, 2017.

[CP17] Yi-Jun Chang and Seth Pettie. A time hierarchy theorem for the local model. *FOCS*, 2017.

[CPS17] Kai-Min Chung, Seth Pettie, and Hsin-Hao Su. Distributed algorithms for the lovász local lemma and graph coloring. *Distributed Computing*, 30(4):261–280, 2017.

[EJ73] Jack Edmonds and Ellis L Johnson. Matching, euler tours and the chinese postman. *Mathematical programming*, 5(1):88–124, 1973.

[EN17] Michael Elkin and Ofer Neiman. Efficient algorithms for constructing very sparse spanners and emulators. In *Proceedings of the Twenty-Eighth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 652–669. Society for Industrial and Applied Mathematics, 2017.

[Fan97] Genghua Fan. Minimum cycle covers of graphs. *Journal of Graph Theory*, 25(3):229–242, 1997.

[FG17] Manuela Fischer and Mohsen Ghaffari. Sublogarithmic distributed algorithms for lovász local lemma, and the complexity hierarchy. In *31st International Symposium on Distributed Computing, DISC 2017, October 16-20, 2017, Vienna, Austria*, pages 18:1–18:16, 2017.

[FHK16] Pierre Fraigniaud, Marc Heinrich, and Adrian Kosowski. Local conflict coloring. In *Foundations of Computer Science (FOCS), 2016 IEEE 57th Annual Symposium on*, pages 625–634. IEEE, 2016.

[FJS14] Joan Feigenbaum, Aaron D. Jaggard, and Michael Schapira. Approximate privacy: Foundations and quantification. *ACM Trans. Algorithms*, 10(3):11:1–11:38, 2014.

[FKN94] Uriel Feige, Joe Kilian, and Moni Naor. A minimal model for secure computation (extended abstract). In *STOC*, 1994.

[GGG+14] Shafi Goldwasser, S. Dov Gordon, Vipul Goyal, Abhishek Jain, Jonathan Katz, Feng-Hao Liu, Amit Sahai, Elaine Shi, and Hong-Sheng Zhou. Multi-input functional encryption. In *Advances in Cryptology - EUROCRYPT*, pages 578–602, 2014.

[Gha15] Mohsen Ghaffari. Near-optimal scheduling of distributed algorithms. In *Proceedings of the 2015 ACM Symposium on Principles of Distributed Computing, PODC*, pages 3–12, 2015.

[Gha16] Mohsen Ghaffari. An improved distributed algorithm for maximal independent set. In *Proceedings of the Twenty-Seventh Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 270–277. Society for Industrial and Applied Mathematics, 2016.

[GMW87] Oded Goldreich, Silvio Micali, and Avi Wigderson. How to play any mental game or A completeness theorem for protocols with honest majority. In *Proceedings of the 19th Annual ACM Symposium on Theory of Computing, 1987, New York, New York, USA*, pages 218–229, 1987.

[GP16] Mohsen Ghaffari and Merav Parter. Near-optimal distributed algorithms for fault-tolerant tree structures. In *Proceedings of the 28th ACM Symposium on Parallelism in Algorithms and Architectures*, pages 387–396. ACM, 2016.

[HIJ+16] Shai Halevi, Yuval Ishai, Abhishek Jain, Eyal Kushilevitz, and Tal Rabin. Secure multiparty computation with general interaction patterns. In *ITCS*, pages 157–168, 2016.

[HLP11] Shai Halevi, Yehuda Lindell, and Benny Pinkas. Secure computation on the web: Computing without simultaneous interaction. In *Advances in Cryptology - CRYPTO 2011 - 31st Annual Cryptology Conference, Santa Barbara, CA, USA, August 14-18, 2011. Proceedings*, pages 132–150, 2011.

[HSS16] David G Harris, Johannes Schneider, and Hsin-Hao Su. Distributed (+ 1)-coloring in sublogarithmic rounds. In *Proceedings of the 48th Annual ACM SIGACT Symposium on Theory of Computing*, pages 465–478. ACM, 2016.

[II86] Amos Israeli and Alon Itai. A fast and simple randomized parallel algorithm for maximal matching. *Information Processing Letters*, 22(2):77–80, 1986.

[IK97] Yuval Ishai and Eyal Kushilevitz. Private simultaneous messages protocols with applications. In *Fifth Israel Symposium on Theory of Computing and Systems, ISTCS 1997, Ramat-Gan, Israel, June 17-19, 1997, Proceedings*, pages 174–184, 1997.

[JT92] Ury Jamshy and Michael Tarsi. Short cycle covers and the cycle double cover conjecture. *Journal of Combinatorial Theory, Series B*, 56(2):197–204, 1992.

[KLR10] Eyal Kushilevitz, Yehuda Lindell, and Tal Rabin. Information-theoretically secure protocols and security under composition. *SIAM J. Comput.*, 39(5):2090–2112, 2010.

[Kus89] Eyal Kushilevitz. Privacy and communication complexity. In *30th Annual Symposium on Foundations of Computer Science, Research Triangle Park, North Carolina, USA, 30 October - 1 November 1989*, pages 416–421. IEEE Computer Society, 1989.

[Lin92] Nathan Linial. Locality in distributed graph algorithms. *SIAM Journal on Computing*, 21(1):193–201, 1992.

[LMR94] Frank Thomson Leighton, Bruce M Maggs, and Satish B Rao. Packet routing and job-shop scheduling ino (congestion+ dilation) steps. *Combinatorica*, 14(2):167–186, 1994.

[Lub86] Michael Luby. A simple parallel algorithm for the maximal independent set problem. *SIAM journal on computing*, 15(4):1036–1053, 1986.

[NS95] Moni Naor and Larry Stockmeyer. What can be computed locally? *SIAM Journal on Computing*, 24(6):1259–1277, 1995.

[Pel00] David Peleg. *Distributed Computing: A Locality-sensitive Approach*. SIAM, 2000.

[Suo13] Jukka Suomela. Survey of local algorithms. *ACM Computing Surveys (CSUR)*, 45(2):24, 2013.

[Yao82] Andrew Chi-Chih Yao. Protocols for secure computations (extended abstract). In *23rd Annual Symposium on Foundations of Computer Science, Chicago, Illinois, USA, 3-5 November 1982*, pages 160–164, 1982.

A Balanced Partitioning of a Tree

We show that every rooted tree T can be partitioned into two edge-disjoint rooted trees T_1 and T_2 such that (I) $E(T_1) \cup E(T_2) = E(T)$ and (II) $V(T_1), V(T_2) \leq 2/3 \cdot N$ where $N = |T|$. In addition, this partitioning maintains the layering structure of T as will be described later. To compute this partitioning, define the weight $w(v)$ of each vertex v in T to be the number of vertices in its subtree $T(v)$. First, consider the case, where there is a vertex v^* with weight $w(v^*) \in [1/3N, 2/3N]$. In such a case, define $T_1 = T_{v^*}$ and $T_2 = T \setminus E(T(v^*))$. By definition, both T_1 and T_2 are trees, all edges of T are covered and $|T_1|, |T_2| \in [1/3N, 2/3N]$.

Else, if no such balanced vertex exists, there must be a vertex v^* such that $w(v^*) \geq 2/3N$ but for each of its children in T, u_i, it holds that $w(u_i) \leq 1/3N$. In such a case, we consider the children of v^* from left to right $u_1, \ldots, u_k$ and sum up their weights until we get to a value in the range $[1/3N, 2/3N]$. Formally, let $\ell \in \{1, \ldots, k\}$ be the minimal index satisfying that $\sum_{i=1}^{\ell} w(u_i) \in [1/3N, 2/3N]$. Since each $w(u_i) \leq 1/3N$, such an index ℓ exists. We then set $T_1 = \bigcup_{i=1}^{\ell} (T(u_i) \cup \{(u_i, v^*)\})$ and $T_2 = T \setminus \bigcup_{i=1}^{\ell} V(T(u_i))$. By construction, all edges of T are covered by T_1 and T_2. In addition, by definition, $|T_1| \in [1/3N, 2/3N]$ and hence also $T_2 \in [1/3N, 2/3N]$.

Finally, we pick the roots r_1, r_2 of T_1, T_2 (respectively) to be the vertices the are close-most to the root r in T. We then get for $u, v \in T_1$, that if u is closer to the root than v in T, then also u is closer to the root r_1 than v in T_1.

B Missing Details for Algorithm DistCycleCover

Postorder Numbering on a Tree. Given a BFS tree T, We now show a procedure which assigns the vertices numbers $N : V \to [1, n]$ according to a postorder traversal on T in $O(D)$ rounds. Each node u first computes the number of vertices in $T(u)$. This can be done in $O(D)$ rounds by working from the leafs up. At the end, each node also knows the number of nodes in the subtree of each of its children. For a range of integers $R = [i, j]$, let $\max(R) = j$ and $\min(R) = i$. By working from root to leaf nodes, each vertex u computes a range $R(u)$, indicating the bucket of post-order numbers given to its vertices in $T(u)$. Let $R(s) = [1, n]$. Given that a vertex v has received its range $R(v)$, it sends to its children their ranges in the following manner. Let $u_1, \ldots, u_\ell$ be the children of v ordered from left to right. Then $R(u_1) = [\min(R(v)), \min(R(v)) + |T(u_1)| - 1]$ and for every $i \geq 1$, $R(u_i) = [\max(R(u_{i-1})) + 1, \max(R(u_{i-1})) + |T(u_i)|]$. This proceeds for $O(D)$ rounds, at the end every u knows $R(u)$. Now, each node u, sets its own number $N(u) = \max(R(u_i))$. Since by this numbering, each vertex has the maximum number in its subtree, it is indeed a postorder numbering.

Sending Information on Virtual Edges. For every virtual edge (a, a') added in phase $i \geq 1$, we show in the analysis section, that there is a precomputed a-a' walk $W_{a,a'}$ in G of length $O(2^i \cdot D)$. We assume that in phase i, all the edges on the walk of virtual edge added in phase $j \leq i - 1$, are already marked and that each such edge on the walk knows the endpoint of the virtual edges and the edges in G that this walk should cover. The analysis shows that in the virtual edges added in phase $i - 1$ are important of $O(2^{i-1})$ edges in G.

Assume that it is given for all virtual edges added in phase $j \leq i - 1$, we now show how the algorithm provides these properties for the virtual edges added in phase i. Assume that both edges (a, a') and (b, b') incident to the matched pair $\langle a, b \rangle$ are virtual. Then, we send the information on

the walks $\pi(a, b, T_{B_x})$, $W_{a,a'}$ and $W_{b,b'}$. This allows the endpoints of virtual edge $a', b'$to learn about each other. In addition, by passing the identifiers of the 2^{i-1} edges that are supposed to be covered by the walks $W_{a,a'}$ and $W_{b,b'}$, all the nodes on the new walk $W_{a',b'} = W_{a',a} \circ \pi(a, b, T_{B_x}) \circ W_{b,b'}$ can get this information. This is done on for all the virtual edges added in phase i simultaneously using the random delay approach. In the analysis section we show that the length of the walks $W_{a,a'}$ is bounded by $O(2^{1/\epsilon}D)$ and each edge appears on $O(n^{2\epsilon})$ many paths. Thus using the random delay approach, this can be done in $O(2^{1/\epsilon} \cdot D + n^{2\epsilon})$ rounds.

Marking Edges on Cycles and Making Them Simple. We first make every edge e (i.e., the endpoints of the edge) know the set of edges that are covered by the cycles that go through e. To do that, the endpoints of the virtual edges keep information of the covered edge endpoints. In phase 1, this is easily obtained since the endpoint of the virtual edges are also the vertices whose two edges should be covered by the cycle. Assuming that this information is kept up to phase $i-1$, in phase i, when adding the virtual edge $u - v$, the combined information traverses through the edge disjoint path $\pi(u, v)$ computed on the tree of the block.

We next make every edge e know all the edges of the cycles that go through e. Since each edge appears on n^{ϵ} cycles and the length of the cycles is $O(2^{1/\epsilon} \cdot D)$, using standard random delay techniques, it can be done in $\widetilde{O}(n^{2\epsilon} + 2^{1/\epsilon} \cdot D)$ rounds. Once each edge e sees the entire cycle, it can locally correct it to be simple as described in Section 4.1.