POPPY: Scalable and Secure Spectral Centrality for Distributed Graphs via Homomorphic Encryption

Claire Guichemerre
Université de Rennes, CNRS, Irisa
claire.guichemerre@irisa.fr

Tristan Allard
Université de Rennes, CNRS, Irisa
tristan.allard@irisa.fr

Sofiane Azogagh
Université du Québec à Montréal
azogagh.sofiane@courrier.uqam.ca

Marc-Olivier Killijian
Université du Québec à Montréal
killijian.marc-olivier.2@uqam.ca

Sébastien Gambs
Université du Québec à Montréal
gambs.sebastien@uqam.ca

Amr El Abbadi
University of California Santa Barbara
amr@cs.ucsb.edu

Abstract

In this paper, we introduce Poppy, a novel suite of privacy-preserving algorithms designed for computing spectral centrality measures over graphs distributed across mutually distrustful data centers. Poppy is the first approach that achieves together *generality*, *accuracy*, and *scalability* with respect to the number of participants. Poppy uses the CKKS fully homomorphic encryption scheme to support the arithmetic division operation over ciphertexts. Poppy consists of three variants: Poppy$_S$, optimized for sparse graphs using SIMD operations; Poppy$_D$, tailored for dense graphs through efficient encrypted matrix-vector multiplication; and Poppy$_H$, a hybrid between Poppy$_S$ and Poppy$_D$ for coping with contexts involving a large number of remote nodes. In addition to its core algorithms, Poppy comes with a pruning strategy based on a new notion of node equivalence called INC-equivalence. Pruning is indeed of utmost importance when using fully homomorphic encryption in order to minimize the number of encrypted operations performed. The INC-equivalence notion allows us to eliminate redundant nodes efficiently and without any impact on the accuracy of the centrality scores. Our comprehensive theoretical analysis and empirical evaluation on real-world and synthetic datasets demonstrate that Poppy achieves together *generality, accuracy, and scalability.*

Keywords

Privacy-preserving data analytics, spectral centrality measure, distributed graph processing, fully homomorphic encryption.

1 Introduction

Graph data structures, which model entities and their relationships, are pervasive [44, 47]. A significant number of graphs contain personal data and are commonly partitioned across multiple data centers due to performance requirements, compliance with privacy regulations, or contractual terms. For example, financial transaction graphs are often distributed across various banks or other financial institutions. Large-scale social networks like Facebook or Twitter (X) are managed by a single entity but both the infrastructures that store the graph and the related individuals are geographically distributed. As a result, computing global graph analytics over these distributed graphs poses major privacy challenges. When data is held by mutually distrustful parties (*e.g.*, competing financial institutions), sharing personal data is restricted by legal, contractual or reputational constraints. Even within a single organization, data protection laws like the GDPR (EU) or the CCPA (California) may impose conflicting restrictions on cross-border data transfers. A striking example is the €1.2 billion fine imposed on Meta IE in 2023 for transferring EU user data to the USA.

In this work, we focus on *spectral centrality measures*, which quantify the relative importance of nodes in a graph. Such measures are widely used in applications like fake news mitigation, disease spread modeling [21], and financial fraud detection[1]. They are typically computed from the dominant eigenvectors of the graph's adjacency matrix and include metrics such as Eigenvector Centrality [13], PageRank [14], Katz [32], HITS [34] and SALSA [38]. Despite their widespread use, securely and efficiently computing these measures in a distributed setting remains challenging. In particular, prior work exhibits notable limitations. First, algorithms based on *differential privacy* (*e.g.,* [51]) can support common spectral centrality measures but might strongly degrade the accuracy due to noise injection. Second, algorithms based on *additively homomorphic encryption* (*e.g.*, [45]) are more accurate but they are restricted to a few spectral centrality measures because they only support additions and scalar multiplications. Third, algorithms based on *secure multi-party computation protocols* (*e.g.*, [5, 50]) do not scale with the number of participants. Finally, works focusing on outsourcing the graph to an untrusted server (*e.g.*, [37, 40, 46, 49]) do not address the core problems of secure distributed computation of spectral centrality measures. Trusted Execution Environments (TEEs) have also been proposed to support privacy-preserving graph analytics by isolating computations in hardware-protected enclaves [19]. However, TEEs rely on strong hardware trust assumptions and have been shown to be vulnerable to various side-channel and microarchitectural attacks [16], which may limit their applicability in highly adversarial multi-party settings.

To circumvent these limitations, we propose Poppy[2], a suite of algorithms for computing spectral centrality measures over graphs partitioned among mutually distrustful participants. Poppy is the first approach that achieves together *full generality* (it is not restricted to any single spectral centrality measure), *accuracy* (it introduces much less noise than differentially private algorithms),

Proceedings on Privacy Enhancing Technologies 2026(4), 869–890

https://doi.org/10.56553/popets-2026-0149

[1] https://www.oracle.com/financial-services/aml-financial-crime-compliance/graph-analytics-powers-the-game/
[2] Poppy means POppy Provides Privacy to centralitY measures computation.

and *scalability* with the number of participants. The key enabler brought by Poppy is the support of the arithmetic division operation thanks to the use of the CKKS *fully homomorphic encryption* scheme [18]. CKKS indeed supports approximate arithmetics over encrypted data (*e.g.*, additions, multiplications, divisions). Fully homomorphic encryption schemes are well-known for their large space and time overheads that make it challenging to achieve affordable performances, even for small-sized graphs. In particular, the dominant "think-as-a-vertex" graph processing paradigm leads to prohibitive costs when using fully homomorphic encryption, as shown in our experiments. Poppy is the result of a deep and careful redesign of both data structures and graph processing algorithms.

Poppy comes in three variants. The first variant, Poppy$_S$, leverages the SIMD capabilities of CKKS to process multiple centrality scores within a single ciphertext, significantly reducing the number of ciphertexts required. It is particularly well suited to sparse graphs with low in-degrees. The second variant, Poppy$_D$, reformulates the computation as a matrix–vector product, thereby reducing the number of costly homomorphic rotations. This makes Poppy$_D$ more suitable for denser graphs with a limited number of remote nodes. The third variant, Poppy$_H$, combines ideas from Poppy$_S$ and Poppy$_D$ by reducing the matrix dimension used in Poppy$_D$. Like Poppy$_D$, it handles denser graphs well, but it performs better when the number of remote nodes is large. In addition to these core algorithms, Poppy includes a pruning strategy based on a new notion of node equivalence called INC-equivalence. Pruning is crucial when using fully homomorphic encryption in order to minimize encrypted operations. INC-equivalence enables us to remove redundant nodes efficiently without affecting the accuracy of the centrality scores.

Our main contributions are the following:

- We propose Poppy, a suite of fully homomorphic encryption–based algorithms for computing spectral centrality measures over partitioned graphs held by mutually distrustful data centers. Poppy achieves *generality, accuracy, and scalability*. For concreteness, we instantiate Poppy with PageRank. The three Poppy variants offer complementary trade-offs and adapt to different graph topologies.
- We introduce a new structural equivalence relation, INC-equivalence, and a corresponding pruning algorithm, PINC, that reduces the number of nodes without affecting centrality scores. We prove that INC-equivalence generalizes graph orbits [23] and that PINC is complete and sound.
- We prove the security of all Poppy variants in the honest-but-curious model under the standard *simulatability* paradigm, and we analyze their asymptotic complexities.
- We provide an extensive experimental evaluation on real and synthetic graphs. Our results show that Poppy achieves near-perfect utility (Kendall Tau close to 1), scales with the number of data centers, and remains practical in CPU and bandwidth for medium-sized graphs.[3] We further show that PINC significantly outperforms existing pruning tools, pruning up to 3.64× more nodes and running up to three orders of magnitude faster.

[3]Up to 35k nodes and 103k edges.

2 Problem Statement and Building Blocks

n	Number of nodes in the graph $\mathcal{G}$
k	Denominator of the teleportation term
d	PageRank Damping factor
u_j's	Set of encrypted vectors for Poppy$_S$ computation
M	Encrypted matrix for Poppy$_D$ computation
$\mathcal{D}_i^{in}$	Set of distant incoming data centers wrt DC_i
$\mathcal{D}_i^{out}$	Set of distant outgoing data centers wrt DC_i
τ	Number of iterations
ℓ	Number of slots in a ciphertext

Table 1: Summary of the main notations used in this paper.

2.1 System Model and Spectral Centrality Mesures

We consider the setting in which a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ is partitioned across a set $\mathcal{DC}$ of mutually distrustful data centers [4]. Each $DC_i \in \mathcal{DC}$ stores a subgraph $\mathcal{G}_i = (\mathcal{V}_i, \mathcal{E}_i)$ such that $\mathcal{V}_i \cap \mathcal{V}_j = \emptyset$ and $\mathcal{E}_i \cap \mathcal{E}_j = \emptyset$ for $i \neq j$. Each edge $(u, v) \in \mathcal{E}_i$ satisfies $u \in \mathcal{V}_i$ and $v \in \mathcal{V}$, meaning DC_i holds all outgoing edges of its nodes. We further distinguish between *intra-data center edges*, in which both endpoints belong to the same DC_i, from *inter-data center edges*, which link nodes across different data centers. Two data centers DC_i and DC_j are said to be connected if there exists at least one inter-data center edge (see Fig. 1). If this inter-data center edge is from DC_i to DC_j, we call DC_i a *distant incoming data center* with respect to DC_j and DC_j a *distant outgoing data center* with respect to DC_i. Figure 1 shows the running example used along this section.

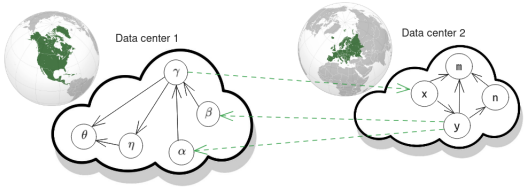


Figure 1: A graph partitioned over two data centers.

Centrality measures assign scores to nodes based on their structural position in a graph. In particular, *spectral centrality measures* rely on the stationary distribution of random walks and can be computed by approximating iteratively the dominant eigenvectors of the adjacency matrix. Common instances include the **Eigenvector centrality (EV) [13]**, that scores each node based on the score of its incoming neighbors, the **PageRank centrality (PR) [14]**, that extends EV by weighting the scores transmitted by an incoming neighbor with its out-degree and by including a *teleportation* term, the **Katz centrality [32]**, that generalizes PR by including multi-hop contributions, and the **HITS [34] and SALSA [38] centralities**, that assign two scores per node (hub and authority) according to both the incoming and the outgoing edges. The iterations rely on simple arithmetic operations, such as multiplications by a constant (all methods), additions and divisions by a constant (all but EV),

[4]Each node $u \in \mathcal{V}$ is described by a node identifier rather than its real-world identity.

and divisions by a sum of scores (all when normalized, HITS in any case).

Throughout the paper, we use PageRank as a representative spectral centrality measure. For a node u at iteration t, the PageRank score $PR_t(u)$ is computed as:

$$PR_t(u) = \frac{1-d}{k} + d \sum_{v \in \mathtt{in}(u)} \frac{PR_{t-1}(v)}{\deg_{out}(v)} \quad (1)$$

in which d is a user-defined constant between 0 and 1 and called the *damping factor* (typically 0.85), k is a user-defined constant between 1 and n, $\mathtt{in}(u)$ is the set of incoming neighbors of u, $\deg_{out}(v)$ is the out-degree of v. PR values are initialized uniformly with $PR_0(u) = 1/n$ for all $u \in \mathcal{V}$. The left-term of the equation is called the *teleportation term* and represents the importance given to random "teleportations" from one node to another during the random walk. It depends on the user-defined constants d and k. An optional term might be added to each PageRank score at each iteration: the sum of the PageRank of the nodes that have no outgoing edges (also called *dangling nodes*) multiplied by the damping factor and divided by the total number of nodes [28].

The possibility to include the optional term and to choose various arbitrary values for d and k impacts the convergence speed of the iterative computation and the PageRank scores of nodes. More specifically, without any normalization, the distribution of the PageRank scores is not a probability distribution in general. The PageRank scores are thus often normalized, either at the end of each iteration or during the last iteration only. During iteration t the normalized PageRank score of node u is:

$$\overline{PR}_t(u) = \frac{PR_t(u)}{\sum_{v \in \mathcal{V}} PR_t(v)} \quad (2)$$

PageRank is relatively simple with only one score per node, while both HITS and SALSA compute two scores per node. We give an overview of HITS below before providing more details on the applicability of our solution to deal with HITS and SALSA in Appendix B.3. For each node u and iteration t, HITS assigns two scores, $H_t(u)$ and $A_t(u)$, computed iteratively in two distinct steps:

$$A'_t(u) = \sum_{v \in \mathtt{in}(u)} H_{t-1}(v) \quad \text{and} \quad H'_t(u) = \sum_{v \in \mathtt{out}(u)} A_{t-1}(v), \quad (3)$$

in which $\mathtt{in}(u)$ (resp. $\mathtt{out}(u)$) is the set of incoming (resp. outgoing) neighbors of u. Then, both score vectors are normalized as follows:

$$A_t(u) = \frac{A'_t(u)}{\sqrt{\sum_{v \in \mathcal{V}} A'_t(v)^2}} \quad \text{and} \quad H_t(u) = \frac{H'_t(u)}{\sqrt{\sum_{v \in \mathcal{V}} H'_t(v)^2}}. \quad (4)$$

Each score is initialized with $A_0(u) = H_0(u) = 1$ for all nodes $u \in \mathcal{V}$.

2.2 Homomorphic Encryption

Poppy performs computations over encrypted data using the CKKS fully homomorphic encryption scheme [18]. CKKS supports approximate arithmetics over floats, making it well-suited for spectral centrality measures.

CKKS has *Single Instruction-Multiple Data* (SIMD) capabilities enabling the encoding and encryption of fixed size cleartext vectors allowing slot-wise homomorphic operations. Each ciphertext supports a limited number of homomorphic multiplications, called the *multiplicative depth*. To support more multiplications, the ciphertext can be *bootstrapped* and recover part of its initial multiplicative capacity (not all). The multiplicative depth is a user-defined parameter. The larger the depth, the more the number of operations supported but the costlier each CKKS operation. CKKS operates on a *ring dimension* denoted N hereafter, a power of two, and derived from a user-defined security parameter λ and the user-defined multiplicative depth. The number of float values encoded together and encrypted with CKKS (*i.e.*, the vector size discussed above) is denoted ℓ and is at most the ring dimension divided by two (*e.g.*, typically 2^{15} to 2^{16}). In addition to arithmetic operations, CKKS supports *rotations* that cyclically shift each vector element. All CKKS encrypted operations inject a small Gaussian error to the encrypted values. Indeed, the semantic security of CKKS relies on the *Ring Learning With Errors (RLWE)* problem. RLWE provides IND-CPA security, which guarantees, in particular, that a ciphertext is computationally indistinguishable from a random element.

The CKKS operations used in this paper are:

- `KeyGen`: Creation of public and secret keys.
- `Encode`, `Decode`: Conversion between cleartext and plaintext.
- `Encrypt`, `Decrypt`: Conversion between plaintext and ciphertext.
- `Add`, `Mult`: Homomorphic addition and multiplication of ciphertexts or plaintext/ciphertext pairs.
- `ScalarAdd`, `ScalarMul`: Addition and multiplication with a cleartext scalar.
- `Rotate`: Rotation by i slots. Given $c = (m_1, \ldots, m_\ell)$, a rotation by i produces a ciphertext c' where $c' = (m_{i+1}, \ldots, m_\ell, m_1, \ldots, m_i)$ if $i > 0$ or $c' = (m_{\ell-|i+1|}, \ldots, m_\ell, m_1, \ldots, m_{\ell-|i|})$ if $i < 0$.
- `FastRotation`: Optimized execution of multiple rotations of the same ciphertext by reusing shared precomputation, thereby reducing the amortized cost compared to performing independent `Rotate` operations.
- `EvalChebyshev`: Evaluation of an approximate real-valued function on encrypted data using a Chebyshev polynomial interpolation over a bounded input interval.

2.3 Adversary Models

Attack model. The data centers participating to Poppy are assumed to follow the "honest-but-curious" model: they follow the protocol while inferring as much information as possible from the messages exchanged and their own knowledge. Adversarial data centers might collude or not together. The main objective of the adversary is to learn information on the structure of the local subgraph of non-corrupted data centers.

Additionally, we assume, without loss of generality, that all the cryptographic material necessary to run Poppy is available at each data center at the beginning of the protocol. In practice, this can be implemented by a *trusted third party* – responsible for initiating Poppy and for performing the final decryption – or by *threshold FHE* [6, 43], compatible with all RLWE-based FHE schemes. In the case of threshold FHE, both key generation and decryption are performed collaboratively by the participants. The main additional step to be performed online is the final collaborative decryption, which

consists of (1) computing a partial decryption of the final ciphertext at each participant and (2) broadcasting the results to all participants. The computational overhead is negligible compared to using a trusted third party while the communication cost is $O(|\mathcal{DC}|^2)$, instead of $O(|\mathcal{DC}|)$ when a trusted third party is used (where $|\mathcal{DC}|$ denotes the number of data centers). Note that a partial decryption is a ring element whose size is close to the size of a ciphertext. Thus, threshold FHE ensures that the decryption key is not disclosed to any single entity, removing the trust assumption required when using a trusted third party. In our implementation, though, for simplicity, we assume that a trusted third party generates the cryptographic material and holds the decryption key.

Security model. Our security model is based on the *simulatability paradigm* [24] well-known for proving the security of secure multiparty computation algorithms. Loosely speaking, it requires that the information available to the adversary in a *real* Poppy execution be *computationally indistinguishable* from the information available in an *ideal* execution by a trusted third party of an algorithm computing the same centrality measure. The information available to the adversary in the real setting consists of the following: the parameters of the spectral centrality measure to be computed (including the constants d and k), the cryptographic material sent by the trusted third party (including the number of slots ℓ in a ciphertext), the final centrality measures of its local nodes, their incoming and outgoing inter-data center edges as well as the remote nodes. Additionally, with respect to the adversary's knowledge, we consider that the following information is known to all participants: the number of data centers, the total number of nodes in $\mathcal{G}$ and the existence of inter-data center edges. We call *leakage function* the function that outputs all the above information. We denote it $\mathtt{L}(\mathcal{DC}, \{\mathcal{G}_i\})$. Colluding parties can be modeled as a single adversary whose knowledge is the union of their individual knowledge. This collusion does not introduce additional leakage beyond what is captured by the leakage function. Our formal security model is presented in Appendix A.

3 Basic Poppy Algorithms

We consider a distributed setting in which the data centers collaboratively compute a spectral centrality measure using Poppy. Without loss of generality, we instantiate all algorithms below on the well-known PageRank centrality measure. Thanks to the full generality offered by Poppy, they can easily be instantiated on other spectral centrality measures. The "think-as-a-vertex" paradigm is widely used in distributed graph processing frameworks such as Pregel [41] and PowerGraph [25] because it can be easily parallelized within each data center. It treats each node in $\mathcal{V}$ as a computing unit. In the cleartext setting, at each PageRank iteration each node inputs the PageRank scores of its in-neighbors in order to update its own PageRank score. The computation stops when a termination criterion is met (*e.g.*, convergence, maximum number of iterations). Designing a "think-as-a-vertex" approach that uses the CKKS fully homomorphic encryption scheme is trivial. Each node simply maintains its own encrypted PageRank score, updating it iteratively using homomorphic operations. However, this approach does not leverage the SIMD capabilities of CKKS: each ciphertext contains only one value, occupying a single slot out of ℓ slots available. This results in as many ciphertexts as there are nodes in the graph, leading to a rapidly growing space usage and thus to main memory exhaustion and prohibitive bandwidth consumption. While conceptually simple, the specifics of fully homomorphic encryption with SIMD capabilities make vertex-centric programming inappropriate. Our experiments confirm that (see Section 6). This finding motivates the design of our Poppy variants.

3.1 Poppy$_S$ Variant

Poppy$_S$ exploits the SIMD capabilities of CKKS by encoding all local PageRank scores in a single ciphertext per data center: each slot holds the score of one node. This reduces the number of ciphertexts by a factor proportional to the number of slots (typically 2^{15} to 2^{16}). For clarity, we assume that each DC fits its nodes in one ciphertext; the extension to multiple ciphertexts is straightforward.

Poppy$_S$ essentially consists of the distributed iterative computation of the encrypted PageRank values. Remember that each PageRank value results from a sum over (encrypted) incoming PageRank values, multiplied by a scalar, and added to another scalar. Hence, the key point of Poppy$_S$ is the creation of the ciphertexts that must be added together in order to compute correctly the encrypted sum. To this end, we perform each iteration in three steps: **synchronization**, **update**, and **preparation** (see Appendix B for the detailed algorithms). The synchronization and the preparation steps are similar: together, they aim at generating the set of (encrypted) vectors that will be summed up together during the arithmetic computation of the PageRank values. However, the synchronization step inputs the remote (encrypted) vectors while the preparation one inputs the local (encrypted) vectors. The update step performs the arithmetic computations over the encrypted vectors generated. We describe these three steps below, starting with the synchronization and preparation ones.

Synchronization and preparation steps. The goal of these two steps is to generate a set of encrypted vectors, denoted u_j, whose values are slot-aligned for homomorphic addition. Since CKKS does not support intra-slot addition, the values to be aggregated must be located at the same slot across different ciphertexts. During the synchronizations step, each data center receives pre-aligned ciphertexts from its neighboring data centers, containing only the relevant PageRank values for its nodes. These remote u_j's are constructed by rotating the sender's PageRank vector and masking out all non-relevant slots. The received ciphertexts are summed up locally and stored for later use during the update step. The preparation step performs a similar operation on the local PageRank vector by rotating and masking the ciphertext to isolate each value required during the computation. The number of such ciphertexts per data center is thus bounded by the maximum in-degree of the local subgraph. This two-phase mechanism ensures that each u_j contains exactly one non-zero slot at the correct position, allowing the update step to aggregate encrypted PageRank values with no additional rotation.

Update step. The update step homomorphically sums up locally all the u_j ciphertexts (from both local and remote sources), multiplies the result by the damping factor d, and adds the constant term $(1-d)/k$. Each operation is performed once for all the slots thanks

to the SIMD capabilities of CKKS (see Appendix B for an illustration of this step).

3.2 Poppy$_D$ Variant

While Poppy$_S$ fully exploits the SIMD features of CKKS, it may incur significant rotation overhead – especially in dense graphs – since slot alignment requires one rotation per in-neighbor. In the worst case (*e.g.*, a complete graph), this leads to a quadratic number of rotations. To mitigate this, Poppy$_D$ replaces the summation of rotated vectors with a matrix-vector multiplication between each local adjacency matrix and the local PageRank vector. This results in a number of rotations that is linear in the number of nodes, as explained below.

In Poppy$_D$, each data center precomputes a local *un-encrypted* node-by-node adjacency matrix M in which each slot in a column is divided by the outgoing degree of the corresponding node, and which includes additional rows and columns to accommodate remote nodes (for remote nodes, the division by the outgoing degree is performed by the distant data center during the preparation step on the encrypted PageRank vector sent). The matrix remains constant across iterations.

The encrypted PageRank vector is structured as in Poppy$_S$, with one slot per node. The matrix-vector multiplication is performed using the Halevi-Shoup algorithm [27], which efficiently computes encrypted linear transformations using rotations, leading to the quadratic-to-linear worst-case reduction in the number of rotations discussed above. In a nutshell, the Halevi-Shoup algorithm proceeds by iterating over the generalized diagonals of the plaintext matrix. For each generalized diagonal of the matrix, the encrypted PageRank vector is rotated and multiplied by the plaintext diagonal. The ciphertexts resulting from the multiplications of the generalized diagonals are then summed up.

Similar to Poppy$_S$, each iteration in Poppy$_D$ consists of a **synchronization** step, an **update** step, and a **preparation** step. While the update step involves the matrix-vector multiplication presented above in addition to the scalar multiplication with d and the scalar addition with $(1-d)/k$ (see Appendix B for an illustration of this step), the synchronization and preparation steps are significantly simpler than in Poppy$_S$ (no need to perform *adhoc* alignments): the synchronization step involves injecting remote values into the local PageRank vector, and the preparation step masks and extracts only the relevant values to be sent to distant data centers.

3.3 Poppy$_H$ Variant

Finally, the Poppy$_H$ variant is a hybrid between Poppy$_D$ and Poppy$_S$: it uses a matrix multiplication for the local nodes, and an aditional u_j vector for the remote nodes. This reduces the size of the matrix, leading to a lower number of generalized diagonals, and consequently to a lower number of rotations, additions and multiplications during the Halevi-Shoup matrix multiplication. The price to pay is an additional number of rotations and masking to perform for the remote nodes.

Similar to the others Poppy variants, each iteration in Poppy$_H$ consists of a **synchronization** step, an **update** step and a **preparation** step. While the update step is similar to Poppy$_D$, the synchronization and preparation steps are similar to Poppy$_S$ in order to construct the ciphertext of the remote nodes. See Appendix B for an illustration of the update step.

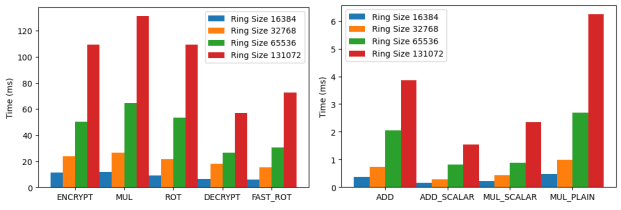
3.4 Normalization

Normalizing PageRank scores is needed in centralized contexts for obtaining a probability distribution. In a distributed context with mutually distrustful data centers, normalization is a must: each data center only accesses the PageRank scores of its local nodes and would be unable to understand their relative importance without normalizing them. Poppy can perform normalization at the end of each iteration or during the last iteration, similar to the centralized context. Normalization consists of (1) summing up at each data center the local PageRank scores, (2) summing up the local sums of all data centers, and (3) dividing each PageRank score by the total sum. The local sum consists of summing up, at each data center, the slots of its encrypted PageRank vector through the Rotate-and-Sum algorithm [27] ($O(\log \ell)$ rotations and additions). The total sum is computed by broadcasting the local sums to all data centers and adding them locally. Finally, each data center applies the EvalChebyshev interpolation to homomorphically approximate $1/|x|$ where x is the total sum. The resulting ciphertext is then multiplied by the encrypted PageRank vector in order to obtain encrypted normalized PageRank scores. The EvalChebyshev interpolation raises two issues. First, a range must be given to EvalChebyshev in which the total sum lies. In our context though, we found experimentally that this issue is actually minor: a coarse but realistic range can be defined without harming the interpolation (see Section 6 for details). Second, the sum of the normalized PageRank scores accumulates the (tiny) CKKS errors of the scores, and is slightly different from one. However, the issue is minor as well because estimating the accumulated error locally at each data center is easy: the distribution of the CKKS errors is public knowledge.

3.5 Using Several Vectors

The number of slots in a CKKS ciphertext is fixed by the ring dimension, which depends on the security level and the multiplicative depth. For instance, with 128-bit security and depth 25, ciphertexts typically provide 2^{16} = 65,536 slots. This suffices for many real-world graphs [22], especially when the graph is partitioned across data centers. However, larger graphs may exceed this capacity. Increasing the ring dimension to accommodate more slots significantly raises the computational cost of encrypted operations and the space consumption of ciphertexts. As illustrated in Figure 2, doubling the number of slots can more than double the cost of multiplications. Moreover, in practice, libraries such as OpenFHE [10], Lattigo [2] and HElib [1] cap ring sizes (*e.g.*, typically to 2^{17}), thus limiting the number of usable slots (*e.g.*, typically to 2^{16}).

To avoid artificially inflating cryptographic parameters, Poppy algorithms support PageRank vectors split over multiple ciphertexts. For Poppy$_S$, coping with multiple vectors is straightforward and does not require any adaptation. For Poppy$_D$, the PageRank vector is partitioned into subvectors of size ℓ (the number of slots in a single ciphertext) and each local matrix is split into $\ell \times \ell$ square submatrices. The usual Halevi-Shoup algorithm is then applied independently to each submatrix-subvector pair. Note that padding might be necessary to guarantee that the size of the local matrix is a



(a) Costliest operations **(b) Others operations**

Figure 2: Average time (ms) over 100 runs on configuration c2 (see Table 3) to perform each CKKS operation over multiple ring dimensions, 128-bit security level, multiplicative depth set to 4 using the OpenFHE library.

multiple of ℓ. Further strategies for reducing the padding overhead by reducing the size of the matrices can be designed. For simplicity, we do not elaborate in this paper. The proof of correctness follows the well-known divide-and-conquer algorithm applied to matrix multiplication. For Poppy$_H$, it is similar to Poppy$_D$ for the matrix and to Poppy$_S$ for the vector.

3.6 Poppy Performance Tuning

The intricate set of parameters of a homomorphic encryption scheme like CKKS and the large time and space overheads implied, increase the complexity of the design of practical solutions to applications with large datasets. Poppy is no exception to this rule. In addition to the core algorithms described above, we have carefully designed the implementation of Poppy to make it work in practice. First, Poppy tackles the high cost of rotations, one of the most expensive CKKS operations, (1) by reducing the cost of the rotation operation based on the *fast rotation* operation (30% gain in time per rotation – see Figure 2), and (2) by reducing the total number of rotations by avoiding to perform redundant rotations of the PageRank vector (a single rotation shifts *all* the slots). Note that using fast rotation impacts the design of our algorithms because a preprocessing needs to be performed on the ciphertext prior to applying the fast rotation operation, and must be re-applied after performing any operation other than a rotation in order to apply fast rotation again. Second, Poppy tackles the large space overhead of CKKS data structures[5] (1) by reducing the number of rotation keys used from $\frac{N}{2} - 1$ keys to $\log(\frac{N}{2})$ keys (where N is the CKKS ring dimension) at the cost of expressing each rotation as a sequence of rotations (similar to [4]), and (2) by minimizing the number of ciphertexts stored in parallel in RAM (*e.g.,* aggregating the u_j's in a single ciphertext, storing a unique rotated ciphertext during the synchronization step and the preparation step). Third, Poppy might need to bootstrap the ciphertext regularly during the iterations due to the consumption of a non-negligible number of multiplicative levels per iteration (e.g., the plaintext-ciphertext multiplications of the PageRank). Moreover, since the bootstrapping operation consumes a non-negligible number of multiplicative levels, the initial multiplicative depth is never fully recovered. For simplicity, we set the multiplicative depth such that after a bootstrapping operation there remains enough multiplicative levels for performing exactly one iteration.

[5]For the ring dimension that matches Poppy's needs (i.e., $N = 131{,}072$ for a 128-bit security level and a multiplicative depth of 25) each ciphertext consumes ≈ 39 MB and each rotation key consumes ≈ 3.5 GiB (with $\frac{N}{2} - 1 = 65{,}536$ rotation keys in total).

4 Pruning the Graph

As discussed earlier, rotations are the dominant cost factor for Poppy, scaling with the number of nodes in the graph. Hereafter, we discuss *pruning* strategies designed to reduce the number of nodes involved in the computation without affecting the centrality values. Our pruning strategies exploit key structural properties of the graph – *source nodes*, *dangling nodes* and *equivalent nodes* – that enable us to ignore a subset of nodes during the execution of Poppy. While our pruning strategies are presented in the context of encrypted distributed computation, they are broadly applicable to other contexts as well and thus are of independent interest.

4.1 Structural Properties of Prunable Nodes

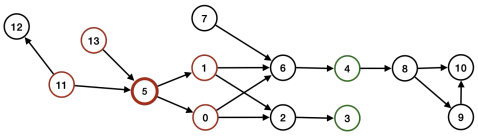
A *prunable* node is a node that can be ignored during the execution of Poppy without any impact on the resulting centrality scores. The possibility to prune a node depends on the specific centrality measure to compute (see Section 2).

4.1.1 Constant Nodes, Dangling Nodes, Equivalent Nodes. *Source nodes*, also called *constant nodes*, have no incoming edge. For the centrality measures that depend only on incoming edges (*e.g.,* PageRank, Eigenvector, Katz), source nodes have a constant centrality score (*e.g.,* equal to $(1-d)/k$ for the PageRank) and can be pruned out of the graph during the computation (their final values can be assigned after the computation).

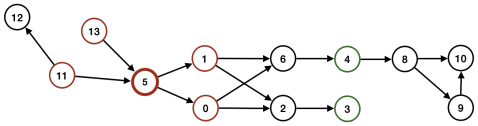
Dangling nodes have no outgoing edge. For the centrality measures that depend only on incoming edges, they do not influence the centrality score of any other node [30]. They can be pruned out of the graph during all the iterations except the final one: their final centrality score can be computed from the penultimate iteration.

Equivalent nodes share identical connectivity patterns, yielding equal centrality scores at every iteration. For each set of equivalent nodes, we only need to include, in the execution of Poppy, a single representative node, the others being pruned out of the graph. After the computation, the centrality score of each representative node is copied to the equivalent nodes. The strictest notion of equivalence is *node orbits* [23]. Two nodes are in the same orbit if they can be permuted through an automorphism preserving the structure of the graph. This guarantees that two nodes in the same orbit have the same centrality scores at each iteration. While this notion of equivalence is sufficient for all spectral centrality measures, it is not necessary for measures based on incoming edges only and thus performs poorly for these measures: nodes with identical incoming patterns but differing outgoing structures will not appear in the same orbit despite having equal centrality scores. We introduce below a novel relaxed notion of equivalence called INC-equivalence designed for measures based on incoming edges only.

4.1.2 INC-equivalence. Two *INC-equivalent nodes* are connected by one or more incoming edges to subgraphs that have the same structure (see Definition 2). INC-equivalence not only captures more nodes than node orbits, leading to a stronger pruning, but can also be computed much more efficiently. We define INC-equivalence below and show in Theorem 1 that it generalizes the notion of



(a) $\mathcal{G} = (\mathcal{V}, \mathcal{E})$



(b) $\mathcal{H} = (\mathcal{W}, \mathcal{F})$

Figure 3: Example of two graphs where the set of common ancestors of nodes 3 and 4 are in red (i.e., $\mathcal{A}_3 \cap \mathcal{A}_4 = \{0, 1, 5, 11, 13\}$) and the set of their generalized lowest common ancestors is in bold font (i.e., $\mathcal{S}_{(3,4)} = \{5\}$).

orbits. Based on this novel notion of equivalence, we design the PINC pruning algorithm (see Section 4.2) that is much faster than orbit computation algorithms and that prunes in practice many more nodes.

The notion of INC-equivalence between two nodes relies on the definition of the subgraph in which the equivalence between the two nodes is verified. This is roughly the subgraph located between the two nodes and their lowest common ancestors. Since the graph $\mathcal{G}$ can contain cycles, we first define the notion of *generalized lowest common ancestors* (glca for short). Roughly speaking, a node w is in the set of glca of nodes u and v if every path from any common ancestor of w to either u and v goes through w. If there is no node in the intersection of the ancestors of u and v, the glca of u and v is an empty set. Figure 3 illustrates this definition. In the following, let the ancestors (resp. descendants) of a node u be the set of nodes that can reach u (resp. that can be reached from u) by a path in the graph. Let $\mathcal{A}_u$ (resp. $\mathcal{D}_u$) be the set of ancestors (resp. descendant) of u.

Definition 1 (Generalized Lowest Common Ancestors). *Let $d(u, v)$ denote the length of the path from u to v. Let $\mathcal{D}_{x,y} = \{a \in \mathcal{A}_u \cap \mathcal{A}_v | \{d(a, u), d(a, v)\} = \{x, y\}\}$ be the set of common ancestors of u and v such that each node of $\mathcal{D}_{x,y}$ is at distance x of u and y of v or y of u and x of v.*

The set of generalized lowest common ancestors *of u and v, denoted $\mathcal{S}_{(u,v)}$, is defined as follows. If $\mathcal{A}_u \cap \mathcal{A}_v \neq \emptyset$, then $\mathcal{S}_{(u,v)} = \{a \in \mathcal{A}_u \cap \mathcal{A}_v, |\mathcal{D}_{d(a,u),d(a,v)}| = 1\}$. $\mathcal{S}_{(u,v)}$ is the set of ancestors of u and v such that every path from any ancestor of a glca to either u and v passes through the glca. Otherwise, $\mathcal{S}_{(u,v)} = \emptyset$.*

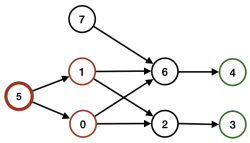
We are now able to define INC-equivalence[6] (see Definition 2). Two nodes are said to be INC-equivalent nodes if and only if there exists an automorphism mapping one to the other in the subgraph $\mathcal{G}_R$ that lies between them and their glca.

Definition 2 (INC-equivalence of u and v). *Let $\mathcal{S}_{(u,v)}$ be the set of glca of u and v. Now, let $M = \{x \in \mathcal{V} | (x \in \mathcal{A}_u \cup \mathcal{A}_v) \wedge (x \notin \mathcal{A}_a, \forall a \in \mathcal{S}_{(u,v)})\}$ be the set of nodes that lies between u and v and their glca. Note that if $\mathcal{S}_{(u,v)} = \emptyset$, M is the set of all ancestors of u and v. Let $C = \{x \in \mathcal{V} | (x \in \mathcal{D}_u \cup \mathcal{D}_v) \wedge (x \in \mathcal{A}_u \cup \mathcal{A}_v)\}$ be the set of nodes of the cycle u and v belong to. Let $D = \{x \in \mathcal{V} | \exists y \in (M \cup C), (y, x) \in \mathcal{E}\}$ be the immediate children of each node of M or C. Finally, let $\mathcal{G}_R = (\mathcal{V}_R, \mathcal{E}_R)$ be the induced subgraph of $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ such that:*

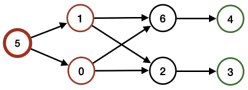
- *$\mathcal{V}_R = M \cup C \cup D$ the union of all the nodes of the subgraph.*
- *$\mathcal{E}_R = \{(x, y) \in E | x \in \mathcal{V}_R$ and $y \in \mathcal{V}_R\}$*

The nodes u and v are INC-equivalent nodes iff there exists an automorphism of $\mathcal{G}_R$, π mapping u to v: $\pi(u) = v$.

[6]The concept of OUT-equivalence can be defined in a similar manner. For centrality metrics focusing on both incoming and outgoing edges (e.g, HITS or SALSA), the well-known notion of orbits already combines INC-equivalence and OUT-equivalence.



(a) Subgraph $\mathcal{G}_R$



(b) Subgraph $\mathcal{H}_R$

Figure 4: Subgraphs used for defining whether node 3 and node 4 are INC-equivalent nodes (see Def. 2) in the two graphs from Figure 3. The subgraph $\mathcal{G}_R = \mathcal{G}[\mathcal{V}_R]$ (Fig. 4a) is induced in $\mathcal{G}$ by $\mathcal{V}_R = M_{(3,4)} \cup C_{(3,4)} \cup D_{(3,4)}$ $\mathcal{G}$, and $\mathcal{H}_R = \mathcal{H}[\mathcal{W}_R]$ (Fig. 4b) is induced in $\mathcal{H}$ by $\mathcal{W}_R = M_{(3,4)} \cup C_{(3,4)} \cup D_{(3,4)}$.

Figure 4 illustrates the INC-equivalence definition between the nodes 3 and 4 of the graphs $\mathcal{G}$ and $\mathcal{H}$ from Figure 3. The automorphisms used are respectively defined on $\mathcal{G}_R$ (Fig. 4a) and $\mathcal{H}_R$ (Fig. 4b). Nodes 3 and 4 from $\mathcal{G}$ are not INC-equivalent nodes, while nodes 3 and 4 from $\mathcal{H}$ are indeed INC-equivalent nodes. However, in both $\mathcal{G}$ and $\mathcal{H}$, nodes 3 and 4 do not belong to the same orbit.

Finally, Theorem 1 shows that INC-equivalence generalizes node orbits (see the proof in Appendix C).

Theorem 1 (INC-equivalence generalizes node orbits). *Node orbits form a subset of the INC-equivalence classes.*

4.1.3 Completeness and Soundness of Prunable Nodes. Theorem 2 shows the completeness of the structural properties identified above, and Theorem 3 shows their soundness. The proofs are in Appendix C.2.

Theorem 2 (Completeness). *The structural properties defined above (1) for spectral centrality measures based on incoming edges only (i.e., constant, dangling, INC-equivalent nodes), and (2) for spectral centrality measures based on incoming and outgoing edges, identify all the nodes that can be pruned out without impacting the centrality scores.*

Theorem 3 (Soundness). *Any node identified as being prunable is either constant, dangling, or has the same centrality score as a representative node that is not pruned out.*

4.2 Computing INC-equivalent Nodes by PINC

We now focus on computing equivalent nodes, which boils down to computing graph automorphisms. While there is no known polynomial time algorithm for computing graph automorphisms in general [9], efficient algorithms have been designed and implemented for coping with classes of graphs that arise in practice (see, e.g., the algorithms based on the "individualization-refinement"

paradigm [42]). This has led to the development of well-known libraries like `nauty and Traces` [42] that can be used for computing node orbits. However, using these algorithms for computing the INC-equivalence relationship for all pairs of nodes in a graph would be prohibitively costly. In order to compute INC-equivalent nodes efficiently, we design PINC an efficient algorithm dedicated to INC-equivalence. Similar to Section 3, we describe PINC below in the context of the PageRank centrality measure, but stress that it can be easily adapted to the other spectral centrality measures that are based only on incoming edges. For the sake of efficiency, the PINC algorithm does not capture INC-equivalent nodes that appear in cycles. Capturing the INC-equivalent nodes that fall within a cycle would result in non-negligible space and time overheads for pruning gains that would be minor in real-life graphs.

The PINC algorithm, described in Algorithm 1, is based on the observation that the PageRank score of a node is a weighted sum of the PageRank scores of its incoming neighbors. Therefore, if a node $u \in \mathcal{V}$ involved in the computation of the PageRank score of $v \in \mathcal{V}$ is equivalent to $\tilde{u} \in \mathcal{V}$, then we can permute u and $\tilde{u}$ in the computation of the PageRank score of v. We will capture this inheritance through the *pattern* of a node.

Definition 3 (Pattern of node t). *Let $\mathcal{G}_p = (\mathcal{V}_p, \mathcal{E}_p)$ be a graph. The pattern of node t is defined as the set of pairs (s, w) where $s \in \mathcal{V}_p$ is a node such that $(s, t) \in \mathcal{E}_p$, and $w \in \mathbb{R}_{>0}$ is a weight. If t is a constant node its pattern is $\emptyset$.*

Algorithm 1: PINC algorithm for pruning nodes

Input : A graph $\mathcal{G}$
Output: The pruned graph $\mathcal{G}_p = (\mathcal{V}_p, \mathcal{E}_p)$

$\mathcal{G}_p \longleftarrow \mathcal{G}$
repeat
  $c \leftarrow$ True
  Let P be the pattern vector for each node of $\mathcal{G}_p$.
  `// Regroup nodes sharing similar pattern`
  Let G be the vector of groups.
  for $(u, v) \in (\mathcal{V}_p)^2$ *s.t* $P[u] = P[v]$ *and* $u < v$ **do**
    $G[u].append(v)$
  end
  `// Merge groups with more than one node`
  for $g \in G$ *s.t.* $|g| > 1$ **do**
    $c \leftarrow$ False
    Let r be the representative node of g.
    for $u \in g \setminus \{r\}$ **do**
      Replace all the occurences of u by r in P
      for $x \in V_p$ *s.t.* $(u, x) \in \mathcal{E}_p$ **do**
        `// Update the weights of` x
        $P[x].weight+ = \frac{1}{deg_{out}(u)}$
      end
    end
  end
  Update $\mathcal{G}_p$: removing all non-representative nodes in G.
until c;
return $\mathcal{G}_p$

The PINC algorithm does not compute the automorphisms used in the INC-equivalence definition directly, but it groups nodes in the same way an automorphism would. At initialization, the weights w of all patterns $\{(s, w)\}$ are set to the out-degree of the source node s. Then, PINC processes iteratively by (1) assigning nodes with identical patterns to the same group, (2) representing each node by a single *representative node* of its group, (3) keeping a single outgoing edge from the representative node to each outgoing node, and (4) updating the patterns of each outgoing node by summing up the weights associated to the source nodes they were connected to. A toy example is given in Appendix D.2. This approach echoes the *individualization-refinment method* to detect automorphism [31, 42, 48], which assigns the same color to nodes with identical signatures, analogous to the node patterns in our context.

We now show that the PINC algorithm is sound and that it is complete when excluding INC-equivalent nodes that fall within a cycle. All proofs can be found in Appendix D.3. Lemma 1 first shows that PINC maintains patterns such that two nodes having the same pattern are INC-equivalent nodes and Theorem 4 builds on Lemma 1 for showing that each node pruned out from the graph is INC-equivalent nodes to a single non-pruned node. Finally, Theorem 5 shows that PINC is complete when excluding INC-equivalent nodes that fall within a cycle.

In the following, let P_0 be the patterns at initialization and P_n the patterns at the end of iteration n of PINC.

Lemma 1 (Same pattern implies INC-equivalence). *For all iterations $n \geqslant 1$, for each nodes $u, v \in \mathcal{V}^2$ if $P_{n-1}[u] = P_{n-1}[v]$ then u and v are INC-equivalent nodes.*

Theorem 4 (Soundness). *Each node pruned from $\mathcal{G}$ is INC-equivalent to a single non-pruned node in $\mathcal{G}_p$. In other words, at each iteration, each node $x \in \mathcal{V} \setminus \mathcal{V}_p$ is represented by a node $y \in \mathcal{V}_p$ s.t. x and y are INC-equivalent nodes.*

Theorem 5 (Completeness). *In a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$: for all iterations $n \geqslant 1$, for each nodes $u, v \in \mathcal{V}^2$ if u and v are INC-equivalent nodes then $P_{n-1}[u] = P_{n-1}[v]$.*

4.3 Pruning Nodes in Subgraphs

Each data center computes the constant nodes, the dangling nodes, and the equivalent nodes locally over its own subgraph. Identifying the constant nodes and the dangling nodes can indeed be performed trivially by each data center independently from the others. Idenfying the full sets of equivalent nodes is however harder. For simplicity, Poppy computes the equivalence relationship only locally. In order to avoid introducing false positives, we consider that all inter-DC edges are incoming from distinct nodes or outgoing to distinct nodes. As shown in the experimental evaluation, the resulting pruning rates are still not negligible.

5 Analysis

Security. Theorem 6 shows that the Poppy algorithms are secure.

Theorem 6 (Poppy$_S$, Poppy$_D$, and Poppy$_H$ are secure). *Any instance π of Poppy$_S$, or of Poppy$_D$, or of Poppy$_H$ executed on a set of honest-but-curious data centers is secure.*

The proof for Theorem 6 is straightforward and essentially relies on the leakage function and the semantic security of CKKS. The

leakage function defined in Section 2 covers all the information leaked during an execution of Poppy (see Algorithm 2). The proofs thus consist in building a simulator that outputs random bitstrings in order to simulate the CKKS ciphertexts exchanged during the execution of π. Note that the security of Poppy when sending ciphertexts at each iteration is equivalent to a single-round protocol under the IND-CPA security of CKKS, since ciphertexts do not reveal information about the underlying plaintexts. See Appendix A for a proof sketch.

Complexity. We summarize the complexities in Table 2 (the baseline is the "think-as-a-vertex" algorithm discussed in Section 3). We can see that Poppy_S is more efficient on sparse graphs than Poppy_D as its complexity depends on the number of edges. Poppy_D is more efficient on denser graphs, and less efficient with a high number of distant nodes μ. Poppy_H is more efficient then Poppy_D on graphs with a high number of distant nodes and a small number of distant incoming edges $|\mathcal{E}_i^{in}|$.

Algo.	Time	Space	Bandwidth
Baseline	$O(n_i)$	$O(n_i)$	$O(\lvert\mathcal{E}_i\rvert)$
Poppy_S	$O(\lvert\mathcal{E}_i\rvert + \log(\ell))$	$O(n_i/\ell + \log(\ell))$	$O(n_i/\ell)$
Poppy_D	$O(n_i/\ell + \mu + \log(\ell))$	$O(n_i/\ell + \log(\ell) + \mu)$	$O(n_i/\ell)$
Poppy_H	$O(n_i/\ell + \lvert\mathcal{E}_i^{in}\rvert + \log(\ell))$	$O(n_i/\ell + \log(\ell))$	$O(n_i/\ell)$

Table 2: Complexities for a single data center DC_i where n_i is its number of local nodes.

6 Experimental Evaluation

6.1 Experimental Settings

Implementation and experimental setup. We implement the "think-as-a-vertex" *baseline* algorithm and the three Poppy variants in C++ (approximately 10K lines of code), using the CKKS OpenFHE [10] implementation. Our current implementation of Poppy is a proof-of-concept that was tested on CPUs. Exploiting the parallelism allowed by the design of Poppy is left as future work (*e.g.*, by distributing the computation within each data center across local computing nodes or by using GPUs [3]) and, according to our preliminary results, would lead to orders of magnitude performance improvements. Each experiment runs on a single machine for all data centers, looping sequentially over the data centers for each iteration. By default, we run each experiment three times and report the average performance and quality measures. The PINC algorithm is implemented in C++ as well and compared to the widely used `nauty and Traces` library [42]. Our codebase is publicly available[7]. The experiments are primarily conducted on the Grid'5000 computing cluster [11], employing up to four distinct hardware settings to vary the available resources. Table 3 summarizes these configurations.

Config.	CPU	RAM
C1	2 CPUs Intel Xeon Gold 6254, 18 cores/CPU	384GiB
C2	2 CPUs Intel Xeon Gold 5118, 12 cores/CPU	512GiB
C3	2 CPUs AMD EPYC 7H12, 64 cores/CPU	1TiB
C4	Apple M1, 8 cores	16GiB

Table 3: Experimental configurations

[7] https://gitlab.inria.fr/cguichem/poppy-pets-artifacts

Datasets. We use both real-world graphs [39] and power-law synthetic graphs generated by the `scale_free_graph` function of NetworkX [26]. We use the real-life graphs to evaluate performance and quality, and synthetic datasets to evaluate scalability. The eight synthetic datasets are called `synthetic-x` where $x \in \{1,000; 5,000; 10,000; 15,000; 20,000; 25,000; 30,000; 35,000\}$ is the number of nodes of the graph (see Appendix E for details) and Table 4 summarizes the five real datasets that we use. We fix the random seed and the number of nodes, and without further information, let the other parameters of the `scale_free_graph` function to default values. The graphs were distributed across multiple data centers according to the geographical distribution of Twitter users [51], allocating randomly and proportionally 43%, 31%, 11%, 10% and 5% of nodes to five separate data centers.

Graph	$\lvert\mathcal{V}\rvert$	$\lvert\mathcal{E}\rvert$	k_{max}	|Const.|	|Dang.|	Density
`Students`	185	311	6	17	22	0.0091
`email-Eu-core`	1,005	25,571	212	14	137	0.0253
`WikiVote`	7,115	103,689	576	4734	1005	0.0020
`p2p-Gnutella04`	10,876	39,994	72	20	5941	0.0003
`Slashdot`	77,360	905,468	2540	0	44	0.0002

Table 4: Real Datasets, where |Const.| (resp. |Dang.|) denotes the number of constant (resp. dangling) nodes.

Centrality measure. We use the PageRank centrality measure for our experiments. We set $d = 0.85$, k to the maximum in-degree (see Table 4 for the real dataset values of k) and do not use the optional term of the PageRank. The total number of iterations is set to $\tau = 10$. We perform normalization once, after the final iteration. See Appendix G for details on the choice of k and τ.

Performance and quality metrics. We evaluate performances in terms of latency (wall-clock time), RAM consumption, and bandwidth consumption. We measure quality by comparing the centrality scores computed by Poppy against the groundtruth, *i.e.,* the scores obtained by a cleartext computation, over the same number of iterations. We use five quality metrics:

Kendall Tau (resp. Top-10 Kendall Tau) The Kendall Tau is a rank-based correlation measure that evaluates the similarity between two ordered lists. It is computed between the scores of the nodes obtained from the groundtruth cleartext computation and the scores of the nodes after an execution of Poppy. The Top-10 Kendall Tau is computed similarly, but restricted to the set of nodes in the top-10 of the groundtruth ranking.

Mean Absolute Error and Mean Squared Error The MAE and MSE are defined as follows: $MAE = \frac{1}{n}\sum_{v\in\mathcal{V}} |p_v - \hat{p_v}|$ and $MSE = \frac{1}{n}\sum_{v\in\mathcal{V}}(p_v - \hat{p_v})^2$ where p_v denotes Poppy's final centrality score for node v and $\hat{p_v}$ denotes the groundtruth centrality score of node v.

Relative Error The relative error is defined as $\eta_v = \frac{|p_v - \hat{p_v}|}{|\hat{p_v}|}$.

6.2 Results and Analysis

6.2.1 Absolute Performance. Figure 5 displays the *total number* of encrypted operations performed *across all data centers* for computing Poppy_S, Poppy_D and Poppy_H on the `p2p-Gnutella04` graph

over $\tau = 10$ iterations. Rotations constitute the dominant computing cost, accounting for both the most frequent operation and the greatest computing expense. Notably, Poppy$_S$ performs nearly twice as many rotations as Poppy$_D$, primarily due to additional rotations during the **synchronization** and **preparation** steps (as highlighted in Figure 6a). Consequently, Poppy$_D$ exhibits significantly reduced wall-clock time compared to Poppy$_S$ on dense graphs, as confirmed by the results in Figure 6. Importantly, the performance of each Poppy variant strongly depends on the underlying graph structure; no single variant consistently outperforms the others across all scenarios. For instance, Poppy$_D$ shows better performance on `p2p-Gnutella04` than Poppy$_S$ and Poppy$_H$ (Figure 6). In contrast on synthetic graphs, Poppy$_S$ performs better due to their sparser structure compared to `p2p-Gnutella04` (see Figure 7). While plaintext multiplications and additions are respectively the second and third most frequent operations, their contribution to the overall latency remains negligible. This is due to their relatively low frequency – typically 3 to 4 times less than rotations – and to their significantly lower computation cost (approx. two orders of magnitude faster than a rotation – see Figure 2).

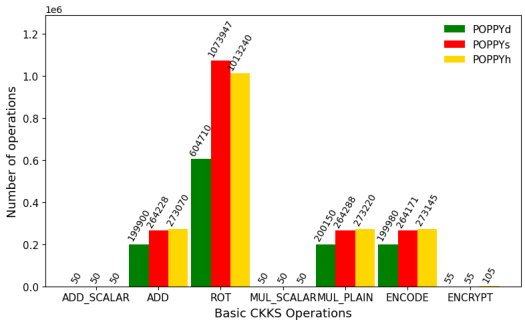


Figure 5: Number of operations on `p2p-Gnutella04` for $\tau = 10$ iterations (config. `C1`).

Figure 6 shows the maximum latency per iteration (*i.e.,* slowest data center) on `p2p-Gnutella04` broken down by algorithmic step (a single run for each Poppy variant): **synchronization**, **update**, and **preparation**, to which we add **bootstrapping**. Each iteration in Poppy$_S$ takes between 41, 404 seconds (roughly 11.5 hours) for the first iteration and 6, 083 seconds (approximately 1.69 hours) in later iterations. The execution time steadily decreases during the first several iterations – up to iteration 8 for Poppy$_S$ and Poppy$_H$ and iteration 6 for Poppy$_D$ – before reaching a plateau. This is due to the progressive consumption of multiplicative levels in CKKS, which reduces the ciphertext sizes and the associated computing overhead. From this point onward, bootstrapping is triggered at each iteration to replenish the required levels, which adds a small constant cost.

Poppy$_D$ is generally faster than Poppy$_S$ and Poppy$_H$ in this experiment. The **update** step exhibits the highest latency due to the matrix-vector multiplications. In contrast, the **synchronization** and **preparation** steps dominate in Poppy$_S$, as each centrality score from a remote data center must be individually rotated and masked for every relevant edge, resulting in a significantly higher number of encrypted operations. However, the optimal choice depends on the graph structure.

Figure 6 also shows the bandwidth consumption per iteration. Synchronization and preparation consume bandwidth by sending and receiving ciphertexts between data centers. If there exists at least one incoming and one outgoing inter-data center edge between each pair of data centers, then the number of ciphertexts in transit is equal to $2 \cdot (|\mathcal{DC}| - 1)$, where $\mathcal{DC}$ is the set of participating data centers. As discussed above, the ciphertext size decreases across iterations, reducing the bandwidth consumption of each party by a factor of 3 to 4. More precisely, each Poppy variant consumes approximately 400 MB for the first iteration and 100 MB in later iterations.

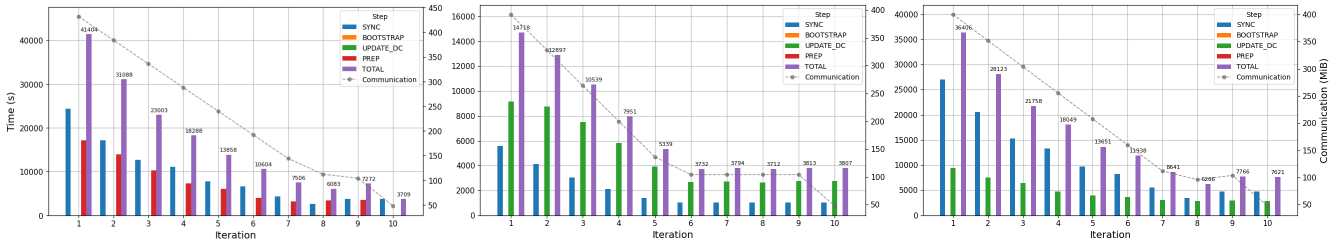
6.2.2 Scalability. Figure 7 shows the scalability of the Poppy variants using synthetic graphs of increasing size. These graphs are all sparse, with the number of edges remaining below twice the number of nodes. In this setting, Poppy$_S$ consistently outperforms Poppy$_D$ in terms of latency. This aligns with our theoretical analysis: the complexity of Poppy$_S$ depends on the number of edges, whereas the complexity of Poppy$_D$ depends on the number of nodes (local and remote), and the complexity of the hybrid Poppy$_H$ depends on the number of local nodes and the number of distant edges.

Figure 8 displays the min/max/avg latency at a data center as the number of data centers increases. In these two experiments, graphs are uniformly partitioned among the data centers. We observe that for all Poppy variants, the latency decreases with the number of data centers (time is reduced by a factor of approximately 2 when increasing from 5 to 10 data centers). This is explained by the decrease of the number of nodes per data center, linear in the number of data centers. This observation confirms that each data center could further distribute its local subgraph across multiple local computing grids to reduce computation time.

Note that we observed much lower latencies when running the "think-as-a-vertex" baseline algorithm on tiny graphs. However, the execution crashed for graphs with $20k$ nodes or more due to memory exhaustion on configuration `C2` (512 GiB RAM).

6.2.3 RAM Consumption. Figure 9 shows the RAM consumption of the data center holding the largest subgraph of the `p2p-Gnutella04` graph (pruning enabled). We observe a decrease from iteration 2 up to iteration 6 or 8, again due to the loss of multiplicative levels. Bootstrapping starts after, providing the ciphertext with the minimum multiplicative depth required to perform one Poppy iteration. Therefore, from iteration 8, the memory consumption remains constant across iterations for all Poppy variants.

6.2.4 Quality. Table 5 presents the quality results for three real-world graphs. All Poppy variants introduce very low errors both in the PageRank scores and in their relative order. The Top-10 Kendall Tau is consistently high (between 0.9555 and 1.0), indicating that the order of the most important nodes is almost perfectly preserved. The high Kendall Tau on the full set of nodes (between 0.9312 and 0.9976) confirms that the relative rankings of the vast majority of nodes remain stable. The few discrepancies observed typically affect nodes with very low PageRank scores: the noise introduced by the CKKS encryption scheme may slightly alter their ordering. In all cases, the MSE is extremely low (on the order of 10^{-9} or less), underscoring



(a) Poppy$_S$ without pruning. (b) Poppy$_D$ without pruning. (c) Poppy$_H$ without pruning.

Figure 6: Maximum latency and bandwidth consumption (slowest data center) per iteration on `p2p-Gnutella04` (config. `C1`).

Graph	$\|\mathcal{V}\|$	$\|\mathcal{E}\|$	Poppy variant	Kendall Tau	Top-10 Kendall Tau	MAE	MSE
`email-Eu-core`	1005	25571	Poppy$_S$	0.9965	1.0000	$7,10.10^{-6}$	$1,1.09^{-10}$
			Poppy$_D$	0.9976	1.0000	$4,66.10^{-6}$	$5,1.10^{-11}$
			Poppy$_H$	0.9965	0.9555	$7,11.10^{-6}$	$1,04.10^{-10}$
`WikiVote`	7115	103689	Poppy$_S$	0.97006	1.0000	$8,94.10^{-6}$	$2,32.10^{-10}$
			Poppy$_D$	0.9312	1.0000	$2,15.10^{-5}$	$1,43.10^{-9}$
			Poppy$_H$	0.9473	0.9555	$1,37.10^{-5}$	$4,57.10^{-10}$
`p2p-Gnutella04`	10876	39994	Poppy$_S$	0.9610	1.0000	$2,24.10^{-5}$	$9,2.10^{-9}$
			Poppy$_D$	0.9629	1.0000	$2,13.10^{-5}$	$9,2.10^{-9}$
			Poppy$_H$	0.9636	1.0000	$2,05.10^{-5}$	$8,9.10^{-9}$

Table 5: Errors for three real graphs (without pruning).

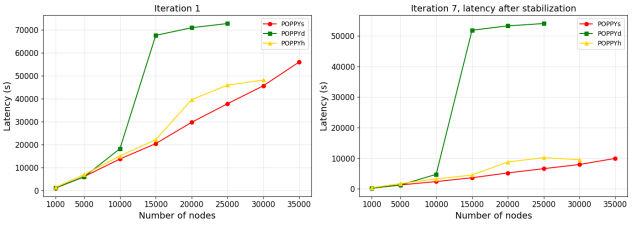


Figure 7: Maximum latency (slowest data center) at various iterations of Poppy on synthetic graphs of varying sizes (config. `C2`).

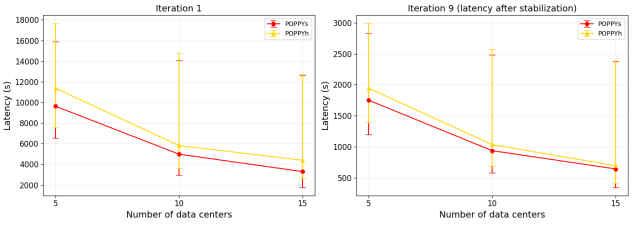


Figure 8: Latency (min, max, average) at various iterations of Poppy on `synthetic-25000` with a varying number of data center (config. `C2`).

the high numerical fidelity of the encrypted computation. Finally, the cumulative distribution functions (CDF) of the relative errors obtained by the three Poppy variants on the `p2p-Gnutella04` graph confirm the tight error bounds (see Appendix F for details). Overall, the quality results confirm that Poppy introduces only minimal error, with negligible impact on both the ranking and the magnitude of

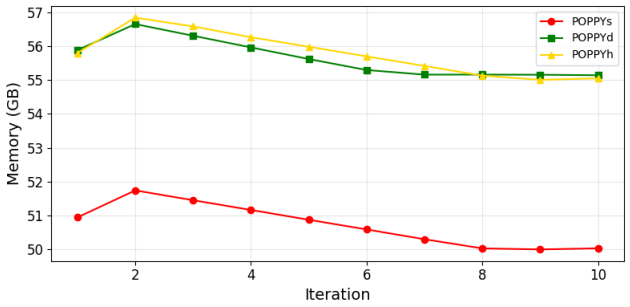


Figure 9: RAM consumption on `p2p-Gnutella04` (config. `C1`).

the centrality scores. This validates the effectiveness of CKKS-based secure computation in preserving high analytical accuracy, even in large-scale distributed environments.

6.2.5 Impact of Pruning. Table 6 presents the pruning rates, latency, and memory usage of both `PINC` and `nauty and Traces` [42] on both real-life and synthetic graphs. We also plot the number of nodes that are pruned by only one out of the two algorithms (called *exclusive pruning* in the table). `PINC` consistently prunes more nodes than `nauty and Traces` on all synthetic graphs and on most real graphs (up to 3.64x more pruned nodes). The notable exception is the `Slashdot` graph, which contains numerous cycles; since our method deliberately avoids pruning nodes within cycles, `nauty` achieves a slightly higher pruning rate in this specific case, removing 449 nodes that `PINC` conservatively retains. This is counterbalanced by the 184 nodes exclusively pruned by `PINC`, leading finally to a pruning rate that is only 0.35% lower. Our algorithm also

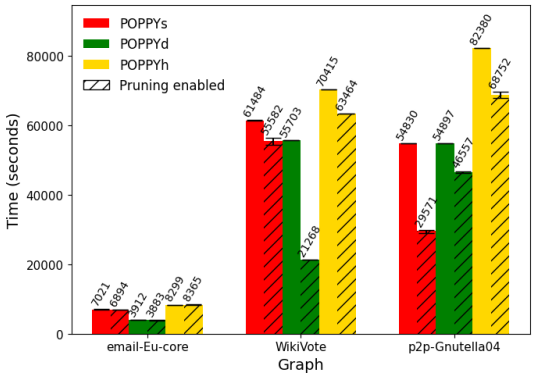


Figure 10: Average latency for all 10 iterations of Poppy, with and without pruning (config. C1).

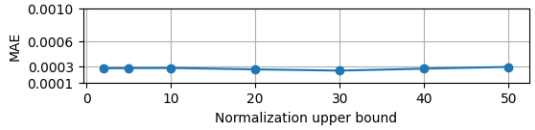


Figure 11: Impact of upper bound choice when fixing lower bound to 1 on synthetic-1000 (config. C3).

shows strong advantages in latency and memory consumption, being up to three orders of magnitude faster than nauty and Traces and using significantly less memory (up to 3.5x less). Finally, the rightmost columns of Table 6 examine the pruning performance when graphs are partitioned across the five data centers. PINC is not designed for distributed graphs and each remote node is conservatively assumed to be a non-equivalent node. Consequently, the number of INC-equivalent nodes detected decreases when PINC is applied to subgraphs. However, in practice with real datasets, due to geographic propinquity and the well-studied homophily principle in social networks, connected nodes should often be located in the same data center, which mitigates this effect. Still, a non-negligible proportion of prunable nodes can be identified and removed, especially in synthetic graphs (up to 27% reduction in the 10k-node case).

To quantify the performance gains enabled by pruning, Figure 10 reports the average latency for a data center to compute Poppy over $\tau = 10$ iterations, with and without pruning. For example, on WikiVote, using Poppy$_S$ variant, the latency is 61.484 seconds without pruning and 55.582 seconds with pruning (1.11x faster), and on p2p-Gnutella04 the latency is respectively 54.830 seconds against 29.571 seconds (1.85x faster). Similar trends are observed for Poppy$_D$ and Poppy$_H$: pruning consistently leads to faster computation, lower memory usage, and reduced ciphertext operations. This highlights the value of lightweight detection techniques in large-scale encrypted graph processing.

6.2.6 Impact of Normalization. Since the normalization steps are similar for all Poppy variants, we run our experiments on Poppy$_S$ only. Figure 11 displays the mean absolute error of Poppy$_S$ on synthetic-1000 with a lower bound set to 1 (before normalization the total sum is greater than one) and a varying upper bound. We observe that the upper bound has a minor impact on the error (between 2.10^{-4} and 3.10^{-4}). Table 7 displays the overhead due to normalization (latency and error) for real and synthetic graphs. We set the range of EvalChebyshev to ±0.5 around the exact total sum. We observe that the latency of the normalization is affordable (less then 30s on average) and that the accumulated error in the normalized sum is between 10^{-3} and 10^{0} and most of the time close to 10^{-1}. Note that each data center could locally compute an estimate of the accumulated error (the CKKS error is public knowledge) and use this estimate for reducing the overall error (*e.g.*, a cleartext division of each PageRank score by the error estimate).

7 Related Work

To the best of our knowledge, no prior work has tackled the problem of computing spectral centrality measures over large distributed graphs while preserving data confidentiality using fully homomorphic encryption.

The work closest to ours is due to Sangers et al. [45], who propose a privacy-preserving PageRank algorithm using the Damgård-Jurik additively homomorphic encryption scheme [20], an extension of the Paillier cryptosystem. Since PageRank involves real-valued computations, they resort to scaling and rounding to operate in the integer domain $\mathbb{Z}_N$. Their approach follows a vertex-centric computation paradigm similar to our baseline algorithm and transmits one ciphertext per node at each iteration—resulting in substantial communication overhead. In contrast, each Poppy variant leverage the SIMD capabilities of CKKS to pack multiple PageRank values into a single ciphertext, substantially reducing bandwidth, especially when ciphertexts are sufficiently full. See Appendix H for more details on the comparison between [45] and Poppy. Additionally, CKKS supports ciphertext-ciphertext multiplications and divisions, enabling the normalization of the set of PageRank values and of more complex spectral algorithms such as HITS or SALSA, which are not feasible under additive-only encryption like Damgård-Jurik.

PGPregel [51] also addresses privacy in distributed graph analytics, but it relies on differential privacy rather than encryption. While differential privacy offers provable privacy guarantees, it fundamentally alters the output distribution through noise injection, thereby degrading utility. In contrast, FHE-based methods like Poppy preserve exact semantics of the computation (up to CKKS approximation). The underlying algorithms also diverge significantly: differential privacy uses randomized aggregation protocols over plaintext data, whereas FHE requires the design of homomorphic circuits to evaluate arithmetic expressions over encrypted values.

Several works explore the use of SMPC protocols for centrality computation [5, 15, 29, 35, 36]. However, they generally assume that all parties share a common node set and hold partial edge information. In Poppy, each data center owns a disjoint subset of nodes and protects them against other parties. Moreover, SMPC-based approaches often involve extensive interaction and require secure computation protocols such as garbled circuits or secret sharing, which may become communication bottlenecks for iterative algorithms like PageRank. Unlike SMPC, Poppy's use of FHE supports non-interactive computation once ciphertexts are exchanged, offering better scalability for distributed settings. Recently [50] proposes

Graph	Exclusive Pruning		Pruning Rate		Time (s)		Memory (MB)		Distributed graph			
	nauty	PINC	nauty	PINC	nauty	PINC	nauty	PINC	Inter-DC edges	|Const.|	|Dang.|	Pruning Rate
Students	0	76	9.73%	51.35%	0.0003	0.003	4.9	1.4	72.65%	16	22	0.54%
email-Eu-core	0	22	2.89%	5.07%	0.02	0.07	13.3	8.5	38.49%	14	135	1.89%
WikiVote	0	2956	18.4%	66.9%	35.4	0.34	39.7	22.9	37.07%	371	909	18.07%
p2p-Gnutella04	0	816	6.69%	14.21%	18.6	0.50	22.2	13.3	45.21%	20	5520	4.95%
Slashdot	449	184	18.89%	18.54%	6626	4.39	252	205.3	19.08%	0	15	4.53%
synthetic-1000	0	339	43.6%	83.7%	0.38	0.01	5.7	2.03	37.67%	505	109	24.9%
synthetic-5000	0	1545	50.7%	87.4%	15.9	0.03	11.3	4.2	36.80%	2545	518	26.84%
synthetic-10000	1	3474	47.25%	87.2%	75.82	0.08	20.8	7.06	35.91%	5120	1072	26.1%

Table 6: Pruning performances of PINC on real and synthetic graphs (config. C4), where |Const.| (resp. |Dang.|) denotes the number of constant (resp. dangling) nodes.

Graph	Avg norm. latency (s)	$\sum PRs$ after norm	MAE before norm	MAE after norm	MSE before norm	MSE after norm	Top-10 Kendall tau	Top-10 Precision
email-Eu-core	23	1.0012	$6,59.10^{-6}$	$2,13.10^{-6}$	$1,01.10^{-10}$	$9,99.10^{-12}$	1.0000	1.0000
WikiVote	23	1.0539	$8.80.10^{-6}$	$7,59.10^{-6}$	$2.40.10^{-10}$	$2,13.10^{-10}$	1.0000	1.0000
p2p-Gnutella04	27	1.9132	$2,02.10^{-5}$	$8,40.10^{-5}$	$9,06.10^{-9}$	$8,85.10^{-9}$	0.9407	0.9667
synthetic-1000	24	1.1094	$1,45.10^{-6}$	$1,1.10^{-4}$	$3,4.10^{-11}$	$2,00.10^{-7}$	1.0000	1.0000
synthetic-10000	26	1.1172	$2,07.10^{-6}$	$1,19.10^{-5}$	$9,2.10^{-11}$	$1,12.10^{-8}$	0.9556	1.0000
synthetic-15000	27	1.1169	$2,12.10^{-6}$	$8,12.10^{-6}$	$8,3.10^{-11}$	$6,89.10^{-9}$	1.0000	1.0000
synthetic-20000	29	1.1150	$3,81.10^{-6}$	$6,80.10^{-6}$	$4,8.10^{-10}$	$4,52.10^{-9}$	0.9556	1.0000

Table 7: Average normalization latency for Poppy$_S$ with pruning and normalization ($\tau = 10$ iterations). The normalization operation uses a lower and upper bound equal to the true PageRank sum $+/-0.5$. The column Top-10 Precision displays the fraction retrieved of the groundtruth top-10 nodes (config. C1 for real graphs and config. C3 for synthetic graphs).

two-party graph analysis algorithm (as PageRank) that is communiction efficient using a combination of FHE and SMPC. However, it is restricted to two parties whereas Poppy supports multiple parties.

A distinct body of literature investigates privacy-preserving queries on encrypted graphs using searchable symmetric encryption (SSE) or FHE [17, 37, 40, 46, 49]. These works generally assume a client-server architecture, where a trusted client issues queries over a graph outsourced to an untrusted cloud server. In contrast, Poppy assumes a distributed setting where each party holds its data locally in the clear and can bypass the trusted entity thanks to secure algorithms distributed over the participating data centers. Furthermore, Poppy performs complex iterative analytics (spectral centrality), not graph querying.

Kim et al. [33] apply homomorphic encryption to matrix factorization for recommendations, using encrypted matrices. However, they assume a centralized graph and does not address centrality computation. More recent efforts have explored encrypted machine learning on graph-structured data [8, 12, 40]. Although these works leverage FHE, they neither consider the distributed setting nor support spectral centralities. Crypto'Graph [8] addresses link prediction using Diffie-Hellman-based secret sharing, but does not use homomorphic encryption and operates in a different threat model.

8 Conclusion

In this paper, we introduced Poppy, the first approach able to compute spectral centrality measures over a graph partitioned on mutually distrutful data centers with full generality, accuracy, and scalability. To this end, Poppy exploits the full power of the CKKS fully homomorphic encryption scheme that supports approximate arithmetic operations over encrypted floats, including the encrypted approximation of the division operation, a key enabler for achieving full generality. Poppy consists of a suite of distributed algorithms, Poppy$_S$, Poppy$_D$, and Poppy$_H$, and of a graph pruning algorithm, called PINC, that uses the novel INC-equivalence notion between nodes. Notably, Poppy$_S$ efficiently leverages SIMD operations for sparse graphs while Poppy$_D$ significantly reduces computational overhead through encrypted matrix-vector multiplications on dense graphs. Poppy$_H$ is a hybrid variant that reconciles the best of the two worlds for coping with moderately dense graphs. Extensive experiments demonstrate the almost perfect accuracy of Poppy on real and synthetic graphs and its scalability with respect to the number of nodes in the graph and of participating data centers. To further improve scalability, each data center could further distribute the computation across local computing nodes. Future work includes developing a comprehensive cost model to choose the most appropriate Poppy variant according to the graph, designing approximate pruning techniques for increasing the pruning rate without impacting performances and using GPUs to accelerate CKKS operations. Beyond the classical honest-but-curious adversary, we are also currently investigating covert adversaries [7], a threat model relevant to our setting. More precisely, a covert adversary can be defined as a participant that is able to cheat or deviate from the protocol to its advantage provided that the detection probability of its malicious actions remain low. This could for instance correspond to a scenario in which a company, through its data center, is trying to learn information about the network architecture of competing participating companies.

Acknowledgments

This work is supported by the "ANR 22-PECY-0002" IPoP project (Interdisciplinary Project on Privacy) of the Cybersecurity PEPR, by the "ANR-18-EURE-0004" PIA EUR CyberSchool project, by the DEEL Project CRDPJ 537462-18 funded by the National Science and Engineering Research Council of Canada (NSERC), and by the Consortium for Research and Innovation in Aerospace in Québec (CRIAQ), together with its industrial partners Thales Canada inc, Bell Textron Canada Limited, CAE inc and Bombardier inc[8]. Sébastien Gambs is also supported by the Canada Research Chair program and a Discovery Grant from NSERC while Marc-Olivier Killijian is supported by Discovery Grants from NSERC. The authors would like to thank the anonymous reviewers for their insightful comments. Claire Guichemerre and Tristan Allard are the corresponding authors.

References

[1] 2023. The HElib open-source software library. https://github.com/homenc/HElib.
[2] 2024. Lattigo v6. https://github.com/tuneinsight/lattigo. EPFL-LDS, Tune Insight SA.
[3] Carlos Agulló-Domingo, Óscar Vera-López, Seyda Guzelhan, Lohit Daksha, Aymane El Jerari, Kaustubh Shivdikar, Rashmi Agrawal, David Kaeli, Ajay Joshi, and José L. Abellán. 2025. FIDESlib: A Fully-Fledged Open-Source FHE Library for Efficient CKKS on GPUs. In *International Symposium on Performance Analysis of Systems and Software*. IEEE.
[4] Ishtiyaque Ahmad, Laboni Sarker, Divyakant Agrawal, Amr El Abbadi, and Trinabh Gupta. 2021. Coeus: A System for Oblivious Document Ranking and Retrieval. *Proceedings of the ACM Symposium on Operating Systems Principles* (2021).
[5] Gilad Asharov, Francesco Bonchi, David Garcia-Soriano, and Tamir Tassa. 2017. Secure Centrality Computation Over Multiple Networks. *Proceedings of the International Conference on World Wide Web*, 957–966.
[6] Gilad Asharov, Abhishek Jain, Adriana López-Alt, Eran Tromer, Vinod Vaikuntanathan, and Daniel Wichs. 2012. Multiparty computation with low communication, computation and interaction via threshold FHE. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*. Springer, 483–501.
[7] Yonatan Aumann and Yehuda Lindell. 2007. Security against covert adversaries: efficient protocols for realistic adversaries. In *Proceedings of the 4th Conference on Theory of Cryptography (TCC'07)*. 137–156.
[8] Sofiane Azogagh, Zelma Aubin Birba, Sébastien Gambs, and Marc-Olivier Killijian. 2024. Crypto'Graph: Leveraging Privacy-Preserving Distributed Link Prediction for Robust Graph Learning. In *Proceedings of the ACM Conference on Data and Application Security and Privacy*. 199–210.
[9] László Babai. 2015. Graph isomorphism in quasipolynomial time [extended abstract]. *Proceedings of the forty-eighth annual ACM symposium on Theory of Computing* (2015).
[10] Ahmad Al Badawi, Andreea Alexandru, Jack Bates, Flavio Bergamaschi, David Bruce Cousins, Saroja Erabelli, Nicholas Genise, Shai Halevi, Hamish Hunt, Andrey Kim, Yongwoo Lee, Zeyu Liu, Daniele Micciancio, Carlo Pascoe, Yuriy Polyakov, Ian Quah, Saraswathy R.V., Kurt Rohloff, Jonathan Saylor, Dmitriy Suponitsky, Matthew Triplett, Vinod Vaikuntanathan, and Vincent Zucca. 2022. OpenFHE: Open-Source Fully Homomorphic Encryption Library. Cryptology ePrint Archive, Paper 2022/915.
[11] Daniel Balouek, Alexandra Carpen Amarie, Ghislain Charrier, Frédéric Desprez, Emmanuel Jeannot, Emmanuel Jeanvoine, Adrien Lèbre, David Margery, Nicolas Niclausse, Lucas Nussbaum, Olivier Richard, Christian Pérez, Flavien Quesnel, Cyril Rohr, and Luc Sarzyniec. 2013. Adding Virtualization Capabilities to the Grid'5000 Testbed. In *Cloud Computing and Services Science*. Vol. 367. 3–20.
[12] Fabian Boemer, Yixing Lao, Rosario Cammarota, and Casimir Wierzynski. 2019. nGraph-HE: a graph compiler for deep learning on homomorphically encrypted data. In *Proceedings of the ACM International Conference on Computing Frontiers*. 3–13.
[13] Phillip Bonacich. 1972. Factoring and weighting approaches to status scores and clique identification. *Journal of Mathematical Sociology* 2 (1972), 113–120.
[14] Sergey Brin and Lawrence Page. 1998. The Anatomy of a Large-scale Hypertextual Web Search Engine. In *Proceedings of the International World Wide Web Conference*. 107–117.
[15] Andreas Brüggemann, Nishat Koti, Varsha Bhat Kukkala, and Thomas Schneider. 2025. MultiCent: Secure and Scalable Computation of Centrality Measures on Multilayer Graphs. *Proceedings on Privacy Enhancing Technologies* 2025, 4 (2025), 944–968.
[16] Jo Van Bulck, Marina Minkin, Ofir Weisse, Daniel Genkin, Baris Kasikci, Frank Piessens, Mark Silberstein, Thomas F. Wenisch, Yuval Yarom, and Raoul Strackx. 2018. Foreshadow: Extracting the Keys to the Intel SGX Kingdom with Transient Out-of-Order Execution. In *USENIX Security Symposium*. 991–1008.
[17] Melissa Chase and Seny Kamara. 2010. *Structured Encryption and Controlled Disclosure*. 577–594.
[18] Jung Hee Cheon, Andrey Kim, Miran Kim, and Yongsoo Song. 2017. Homomorphic Encryption for Arithmetic of Approximate Numbers. In *International Conference on the Theory and Application of Cryptology and Information Security*.
[19] Victor Costan and Srinivas Devadas. 2016. Intel SGX Explained. Cryptology ePrint Archive, Paper 2016/086. https://eprint.iacr.org/2016/086
[20] Ivan Damgård and Mads Jurik. 2001. A Generalisation, a Simplification and Some Applications of Paillier's Probabilistic Public-Key System. In *Public Key Cryptography, 4th International Workshop on Practice and Theory*, Vol. 1992. Springer, 119–136.
[21] Guilherme Ferraz de Arruda, André Luiz Barbieri, Pablo Martín Rodríguez, Francisco A. Rodrigues, Yamir Moreno, and Luciano da Fontoura Costa. 2014. Role of centrality for the identification of influential spreaders in complex networks. *Phys. Rev. E* 90 (2014), 032812.
[22] Sergey Edunov, Dionysios Logothetis, Cheng Wang, Avery Ching, and Maja Kabiljo. 2018. Generating Synthetic Social Graphs with Darwini. *IEEE International Conference on Distributed Computing Systems* (2018), 567–577.
[23] Modjtaba Ghorbani, Matthias Dehmer, Abdullah Lotfi, Najaf Amraei, Abbe Mowshowitz, and Frank Emmert-Streib. 2021. On the relationship between PageRank and automorphisms of a graph. *Information Sciences* 579 (2021), 401–417.
[24] Oded Goldreich. 2005. Foundations of cryptography: a primer. *Found. Trends in Theoretical Computer Science* 1 (2005), 1–116.
[25] Joseph E. Gonzalez, Yucheng Low, Haijie Gu, Danny Bickson, and Carlos Guestrin. 2012. PowerGraph: Distributed Graph-Parallel Computation on Natural Graphs. In *USENIX Symposium on Operating Systems Design and Implementation*.
[26] Aric A. Hagberg, Daniel A. Schult, and Pieter J. Swart. 2008. Exploring Network Structure, Dynamics, and Function using NetworkX. In *Proceedings of the 7th Python in Science Conference*, Gaël Varoquaux, Travis Vaught, and Jarrod Millman (Eds.). Pasadena, CA USA, 11 – 15.
[27] Shai Halevi and Victor Shoup. 2014. Algorithms in HElib. In *Advances in Cryptology – CRYPTO 2014*. 554–571.
[28] James Hendler. 2006. Google's PageRank and Beyond. *Physics Today* 60 (2006), 64–64.
[29] Wen Hu, Xin Xia, Xiao Ding, Xingyi Zhang, Kai Zhong, and Haifeng Zhang. 2023. SMPC-Ranking: A Privacy-Preserving Method on Identifying Influential Nodes in Multiple Private Networks. *IEEE Transactions on Systems, Man, and Cybernetics: Systems* 53 (2023), 2971–2982.
[30] Ilse C. F. Ipsen and Teresa M. Selee. 2007. PageRank Computation, with Special Attention to Dangling Nodes. *SIAM J. Matrix Anal. Appl.* 29 (2007), 1281–1296.
[31] Tommi Junttila and Petteri Kaski. 2011. Conflict propagation and component recursion for canonical labeling. In *International Conference on Theory and Practice of Algorithms in (Computer) Systems*. Springer, 151–162.
[32] Leo Katz. 1953. A new status index derived from sociometric analysis. *Psychometrika* 18 (1953), 39–43.
[33] Jinsu Kim, Dongyoung Koo, Yuna Kim, Hyunsoo Yoon, Junbum Shin, and Sungwook Kim. 2018. Efficient Privacy-Preserving Matrix Factorization for Recommendation via Fully Homomorphic Encryption. *ACM Transactions on Privacy and Security* 21 (2018), 1 – 30.
[34] Jon M. Kleinberg. 1999. Authoritative sources in a hyperlinked environment. In *ACM-SIAM Symposium on Discrete Algorithms*.
[35] Varsha Bhat Kukkala and S. Iyengar. 2018. Computing Betweenness Centrality: An Efficient Privacy-Preserving Approach. (2018), 23–42.
[36] Varsha Bhat Kukkala and S. S. Iyengar. 2020. Identifying Influential Spreaders in a Social Network (While Preserving Privacy). *Proceedings on Privacy Enhancing Technologies* (2020), 537 – 557.
[37] Shangqi Lai, Xingliang Yuan, Shifeng Sun, Joseph K. Liu, Yuhong Liu, and Dongxi Liu. 2019. GraphSE^2: An Encrypted Graph Database for Privacy-Preserving Social Search. *Proceedings of the ACM Asia Conference on Computer and Communications Security* (2019).
[38] R. Lempel and S. Moran. 2001. SALSA: the stochastic approach for link-structure analysis. *ACM Trans. Inf. Syst.* 19, 2 (2001).
[39] Jure Leskovec and Andrej Krevl. 2014. SNAP Datasets: Stanford Large Network Dataset Collection. http://snap.stanford.edu/data.
[40] Ling Liang, Jilan Lin, Zheng Qu, Ishtiyaque Ahmad, Fengbin Tu, Trinabh Gupta, Yufei Ding, and Yuan Xie. 2023. SPG: Structure-Private Graph Database via SqueezePIR. *Proc. VLDB Endow.* 16 (2023), 1615–1628.
[41] Grzegorz Malewicz, Matthew H. Austern, Aart J.C Bik, James C. Dehnert, Ilan Horn, Naty Leiser, and Grzegorz Czajkowski. 2010. Pregel: A System for Large-scale Graph Processing. In *Proceedings of the 2010 ACM SIGMOD International*

[8] https://deel.quebec

Conference on Management of Data. 135–146.
[42] Brendan D. McKay and Adolfo Piperno. 2014. Practical graph isomorphism, II. *Journal of Symbolic Computation* 60 (2014), 94–112.
[43] Christian Mouchet, Elliott Bertrand, and Jean-Pierre Hubaux. 2023. An efficient threshold access-structure for rlwe-based multiparty homomorphic encryption. *Journal of Cryptology* 36, 2 (2023), 10.
[44] Siddhartha Sahu, Amine Mhedhbi, Semih Salihoglu, Jimmy Lin, and M. Tamer Özsu. 2017. The ubiquity of large graphs and surprising challenges of graph processing: extended survey. *Proc. VLDB Endow.* 11, 4 (2017), 420–431.
[45] Alex Sangers, Maran van Heesch, Thomas Attema, Thijs Veugen, Mark Wiggerman, Jan Veldsink, Oscar Bloemen, and Daniël Worm. 2019. Secure Multiparty PageRank Algorithm for Collaborative Fraud Detection. In *Financial Cryptography and Data Security*. 605–623.
[46] Laltu Sardar, Gaurav Bansal, Sushmita Ruj, and Kouichi Sakurai. 2021. Securely Computing Clustering Coefficient for Outsourced Dynamic Encrypted Graph Data. *International Conference on communication systems & networks* (2021), 465–473.
[47] Yuanyuan Tian. 2022. The World of Graph Databases from An Industry Perspective. *ACM SIGMOD Record* 51 (2022), 60 – 67.
[48] Boris Weisfeiler and Andrei Leman. 1968. The reduction of a graph to canonical form and the algebra which appears therein. *nti, Series* 2, 9 (1968), 12–16.
[49] Pengtao Xie and Eric P. Xing. 2014. CryptGraph: Privacy Preserving Graph Analytics on Encrypted Graph. *ArXiv* abs/1409.5021 (2014).
[50] Jiping Yu, Kun Chen, Yunyi Chen, Xiaoyu Fan, Xiaowei Zhu, Cheng Hong, and Wenguang Chen. 2025. GraphAce: secure two-party graph analysis achieving communication efficiency. Article 136, 20 pages.
[51] Amelie Chi Zhou, Ruibo Qiu, Thomas Lambert, Tristan Allard, Shadi Ibrahim, and Amr El Abbadi. 2022. PGPregel: An End-to-End System for Privacy-Preserving Graph Processing in Geo-Distributed Data Centers. In *Symposium on Cloud Computing*. ACM, 386–402.

A Security Model

Recall that the adversary in the real setting learns the output of the leakage function $\mathsf{L}(\mathcal{DC}, \{\mathcal{G}_i\})$. Its output is defined as follows.

- The parameters of the spectral centrality computation, including d and k.
- The cryptographic public parameters, including ℓ.
- The final centrality scores of local nodes.
- The incoming and outgoing inter-data center edges of local nodes, including the identifiers of their corresponding remote nodes.
- The existence of all inter-data center edges.
- The number of data centers $|\mathcal{DC}|$.
- The total number of nodes n in $\mathcal{G}$.

The security model of Poppy is defined in Definition 4 using this leakage function.

Definition 4. *Let π be an instance of a Poppy algorithm executed by the set of data centers $\mathcal{DC}$ for computing a given spectral centrality measure over a partitioned graph $\{\mathcal{G}_i\}$, and $\mathsf{L}(\mathcal{DC}, \{\mathcal{G}_i\})$ be the leakage function. We say that π is secure iff for every adversary A_r attacking π, there exists an adversary A_i in the ideal model so that for every $\chi \in \{0,1\}^*$, the distribution representing the adversarial knowledge over the input graph in the real setting is* computationally indistinguishable *from the distribution representing the adversarial knowledge in an ideal setting in which a trusted third party* ttp *computes the same spectral centrality measure: $\mathsf{REAL}_{\pi,\mathsf{A}_r(\chi,\mathcal{L})}(\{\mathcal{G}_i\}) \overset{c}{\equiv} \mathsf{IDEAL}_{\mathsf{ttp},\mathsf{A}_i(\chi,\mathcal{L})}(\{\mathcal{G}_i\})$ where* REAL *denotes the adversarial knowledge in the real setting and* IDEAL *its counterpart in the ideal setting.*

Theorem 6 shows that the Poppy algorithms are secure according to Definition 4. The proof of Theorem 6 is straightforward (see the sketch below).

Proof (Sketch). The proof consists of listing and analyzing the information obtained by a data center (1) before executing an instance π of a Poppy algorithm and (2) during the execution of π. First, the information obtained before π starts precisely consists of the information already available to the data centers (*e.g.*, the inter-data center edges) and of the parameters needed by π (*e.g.*, the values of ℓ, n, d and k). All this information, included in REAL, is provided to IDEAL through the leakage function $\mathsf{L}(\mathcal{DC}, \{\mathcal{G}_i\})$. Second, it is easy to see from the Poppy algorithms (*e.g.*, Algorithm 3) that the information obtained during the execution of π consists only of CKKS ciphertexts. Since CKKS provides semantic security, CKKS ciphertexts are computationally indistinguishable from random bitstrings: they can be simulated in the IDEAL setting by a random bitstring generator parameterized by the cryptosystem parameters given in the leakage function (*e.g.*, to obtain the bitlength of the ciphertexts). As a result, we can conclude that REAL and IDEAL are computationally indistinguishable from each other. □

B Poppy Algorithms

B.1 Poppy$_S$

Algorithm 2 describes the full execution sequence of Poppy$_S$ at DC_i. Before launching the algorithm, data centers all agree on the number of iterations τ, the dumping factor d, and the denominator k of the teleportation term. Line 4 of Algorithm 2 references to Algorithm 3. The output of Algorithm 2 is the decrypted PageRank scores of the nodes of DC_i.

Algorithm 2: Execution sequence of Poppy$_S$ at DC_i.

Input : The damping factor d, the denominator of the teleportation term k, the number of iterations τ, the subgraph $\mathcal{G}_i$, $\mathcal{D}_i^{in}$ (resp. $\mathcal{D}_i^{out}$) the set of distant incoming (resp. outgoing) data centers of DC_i.
Output : The decrypted PageRanks of the nodes of DC_i.

1 Let η_i be the size of the vectors needed for DC_i.
2 $pks, n, \ell \leftarrow \mathsf{Get}(n_i, \eta_i)$ // Reception of the total number of nodes n, the public keys pks and the size of the vectors ℓ from the trusted entity.
3 Let $\mathcal{U}$ be the set of u_j's initialized ciphertexts, $\mathcal{A}$ be the set of plaintext masks to isolate border nodes, and $\mathcal{C}$ be the set of plaintext masks to isolate border constant nodes.
4 $P \leftarrow \mathsf{Compute}(d, \tau, k, \mathcal{U}, \mathcal{A}, \mathcal{C}, \mathcal{D}_i^{in}, \mathcal{D}_i^{out})$
5 $v \leftarrow \mathsf{Get}(P)$ // Decryption of P by the trusted entity.
6 **return** v.

Algorithm 3 describes the **synchronization** step, the **update** step, and the **preparation** step along the iterations of Poppy$_S$. The output of Algorithm 3 is the ciphertext of the PageRank scores of the nodes of DC_i.

Finally, Figure 12 illustrates the **update** step of Poppy$_S$ for data center DC_2 from our running example (see Figure 1).

B.2 Poppy$_D$ and Poppy$_H$

Figure 13 (resp. Figure 14) illustrates the **update** step of Poppy$_D$ (resp. Poppy$_H$) for data center DC_2 from our running example (see Figure 1).

Algorithm 3: Poppy$_S$ for PageRank in data center DC_i

Input : The damping factor d, the denominator of the teleportation term k, the number of iterations τ, $\mathcal{U}$ the set of u_j's initialized ciphertexts, $\mathcal{A}$ the set of plaintext masks to isolate border nodes, C the set of plaintext masks to isolate border constant nodes, $\mathcal{D}_i^{in}$ (resp. $\mathcal{D}_i^{out}$) the set of distant incoming (resp. outgoing) data centers of DC_i.

Output : The ciphertext P for the PageRanks of DC_i.

for $t \leftarrow 1$ **to** τ **do**
  // Synchronization
  for *each* DC_j *in* $\mathcal{D}_i^{in}$ **do**
    Let $\overline{\mathcal{U}}$ be the remote u_j's.
    $\overline{\mathcal{U}} \leftarrow$ Synchronize(DC_j)
  end
  // Update PRs
  $P \leftarrow \frac{1-d}{k} + d \cdot (\sum_{i \in |\mathcal{U}|} \mathcal{U}[i] + \sum_{i \in |\overline{\mathcal{U}}|} \overline{\mathcal{U}}[i])$
  // Preparation
  $\mathcal{U} \leftarrow$ Reconstruct(P)
  for *each* DC_j *in* $\mathcal{D}_i^{out}$ **do**
    Let $\overline{P}$ be the PageRank vector masked for a remote data center.
    $\overline{P} \leftarrow P \cdot \mathcal{A}[j] + C[j]$
    Send $\overline{P}$ to DC_j.
  end
end
/* Last iteration */
// Synchronization
for *each* DC_j *in* $\mathcal{D}_i^{in}$ **do**
  $\overline{\mathcal{U}} \leftarrow$ Synchronize(DC_j)
end
// Update PRs
$P \leftarrow \frac{1-d}{k} + d \cdot (\sum_{i \in |\mathcal{U}|} \mathcal{U}[i] + \sum_{i \in |\overline{\mathcal{U}}|} \overline{\mathcal{U}}[i])$
return P.

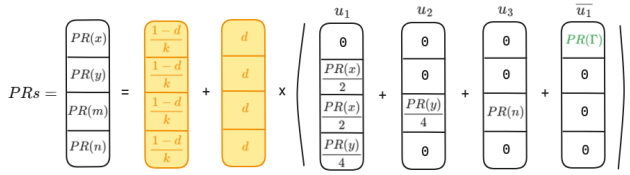


Figure 12: Update step in Poppy$_S$: computing the PageRank ciphertext in data center DC_2.

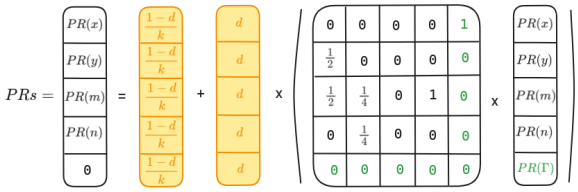


Figure 13: Update step in Poppy$_D$ in data center DC_2

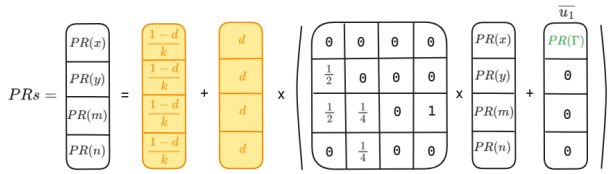


Figure 14: Update step in Poppy$_H$ in data center DC_2

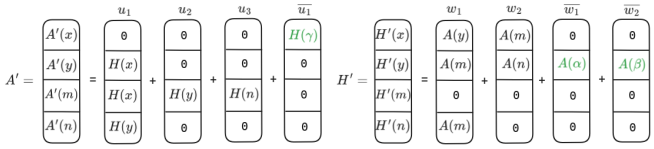


Figure 15: Beginning of the update step in Poppy$_S$ for HITS in data center DC_2

B.3 Poppy$_S$ for HITS and SALSA

HITS and SALSA assign two scores to each node: authority scores, which depend on the hub scores from the previous iteration, and hub scores, which depend on the authority scores from the previous iteration. Each Poppy algorithm can be adapted with minor modifications to support these measures.

Algorithm 4 describes the **synchronization**, **update** and **preparation** steps (similar to PageRank) performed at each iteration of Poppy$_S$ to compute HITS at data center DC_i. The output of Algorithm 4 consists of the ciphertexts of the authority and hub scores for the nodes in DC_i. Similarly to the PageRank computation, Poppy$_S$ requires a set of encrypted vectors for the update step. Let $\mathcal{U}$ (resp. $\mathcal{W}$) denote the set of initialized ciphertexts u_j (resp. w_j) used to compute the authority (resp. hub) scores.

Figure 15 illustrates the beginning of the update step of Poppy$_S$ to compute HITS in data center DC_2 from our running example (see Figure 1). The number of u_j ciphertexts is bounded by the maximum in-degree of the local subgraph, while the number of w_j ciphertexts is bounded by the maximum out-degree. The second part of the update step normalizes each score computed in the first part by applying the Euclidean norm. This differs slightly from the normalization of PageRank scores in that the Euclidean norm is used: squared values are summed up, and a square root applies to the resulting sum. In our homomorphic encryption setting, the squaring operation corresponds to a ciphertext multiplication of the score by itself, and the EvalChebyshev homomorphic operation interpolates $1/\sqrt{x}$ instead of $1/x$. This does not significantly impact the computational cost of the homomorphic evaluation.

The main difference between HITS and SALSA is that, in SALSA, each contribution in the score update is divided by a locally known constant, similarly to PageRank during the beginning of the update step. Therefore, Poppy$_S$ can support SALSA with only minor modifications to the preparation step, and more precisely in the reconstruction of the u_j and w_j ciphertexts. Additionally, the update step of SALSA is simpler, since no normalization is required, which reduces the cost of one SALSA iteration compared to HITS.

Algorithm 4: POPPY$_S$ for HITS in data center DC_i

Input : The number of iterations τ, $\mathcal{U}$ and $\mathcal{W}$ the set of u_j's and w_j's initialized ciphertexts, $\mathcal{A}_A$ (resp. $\mathcal{A}_H$) the set of plaintext masks to isolate border nodes with inter-DC outgoing (resp. incoming) edges, C the set of plaintext masks to isolate border constant nodes, $\mathcal{D}_i^{in}$ (resp. $\mathcal{D}_i^{out}$) the set of distant incoming (resp. outgoing) DCs of DC_i.

Output : The ciphertexts A and H for the authorities and hubs of DC_i.

for $t \leftarrow 1$ **to** τ **do**
  // Synchronization
  Let $\overline{\mathcal{U}}$ (resp. $\overline{\mathcal{W}}$) be the remote u_j's (resp. w_j's).
  for *each* DC_j *in* $\mathcal{D}_i^{in}$ **do**
    $\overline{\mathcal{U}} \leftarrow \texttt{Synchronize}(DC_j)$
  end
  for *each* DC_j *in* $\mathcal{D}_i^{out}$ **do**
    $\overline{\mathcal{W}} \leftarrow \texttt{Synchronize}(DC_j)$
  end
  // Update PRs
  $A' \leftarrow \sum_{i\in|\mathcal{U}|} \mathcal{U}[i] + \sum_{i\in|\overline{\mathcal{U}}|} \overline{\mathcal{U}}[i]$
  $H' \leftarrow \sum_{i\in|\mathcal{W}|} \mathcal{W}[i] + \sum_{i\in|\overline{\mathcal{W}}|} \overline{\mathcal{W}}[i]$
  $Norm'_A \leftarrow \frac{1}{\sqrt{\sum_{v\in\mathcal{V}} A'[v]^2}}$, $Norm'_H \leftarrow \frac{1}{\sqrt{\sum_{v\in\mathcal{V}} H'[v]^2}}$
  $A \leftarrow Norm'_A \cdot A'$, $H \leftarrow Norm'_H \cdot H'$
  // Preparation
  $\mathcal{U} \leftarrow \texttt{Reconstruct}(H)$, $\mathcal{W} \leftarrow \texttt{Reconstruct}(A)$)
  Let $\overline{A}$ (resp. $\overline{H}$) be the authority (resp. hub) vector masked for a remote data center.
  for *each* DC_j *in* $\mathcal{D}_i^{out}$ **do**
    $\overline{A} \leftarrow A \cdot \mathcal{A}_A[j] + C[j]$
    Send $\overline{A}$ to DC_j.
  end
  for *each* DC_j *in* $\mathcal{D}_i^{in}$ **do**
    $\overline{H} \leftarrow H \cdot \mathcal{A}_H[j]$
    Send $\overline{H}$ to DC_j.
  end
end
/* **Last iteration** */
// Synchronization
for *each* DC_j *in* $\mathcal{D}_i^{in}$ **do**
  $\overline{\mathcal{U}} \leftarrow \texttt{Synchronize}(DC_j)$
end
for *each* DC_j *in* $\mathcal{D}_i^{out}$ **do**
  $\overline{\mathcal{W}} \leftarrow \texttt{Synchronize}(DC_j)$
end
// Update PRs
$A' \leftarrow \sum_{i\in|\mathcal{U}|} \mathcal{U}[i] + \sum_{i\in|\overline{\mathcal{U}}|} \overline{\mathcal{U}}[i]$
$H' \leftarrow \sum_{i\in|\mathcal{W}|} \mathcal{W}[i] + \sum_{i\in|\overline{\mathcal{W}}|} \overline{\mathcal{W}}[i]$
$Norm'_A \leftarrow \frac{1}{\sqrt{\sum_{v\in\mathcal{V}} A'[v]^2}}$, $Norm'_H \leftarrow \frac{1}{\sqrt{\sum_{v\in\mathcal{V}} H'[v]^2}}$
$A \leftarrow Norm'_A \cdot A'$, $H \leftarrow Norm'_H \cdot H'$
return A and H.

C INC-equivalence proofs

C.1 INC-equivalence Generalizes Node Orbits

Theorem 1 shows that INC-equivalence generalizes node orbits. For this proof we need the following notation of INC-equivalence class. The INC-*equivalence class* of a node $u \in \mathcal{V}$ is the set of all INC-equivalent nodes of u. INC-equivalence is more general than orbital equivalence.

Proof. For the following proof u is in the orbit of v is defined as $u, v \in \mathcal{V}^2, u \in Orbits(v)$.
Let $u, v \in \mathcal{V}^2, u \in Orbits(v)$ then there is an automorphism π of $\mathcal{G}$ s.t. $\pi(u) = v$.
By definition of the automorphism: $\forall(x, y) \in \mathcal{E}, (\pi(x), \pi(y)) \in \mathcal{E}$.
In particular, if $(x, u) \in \mathcal{E}$, $(\pi(x), \pi(u)) \in \mathcal{E} \Rightarrow (\pi(x), v) \in \mathcal{E}$.

Using Definition 2, we constucte the induced subgraph $\mathcal{G}_R = (\mathcal{V}_R, \mathcal{E}_R)$ from u and v with $\mathcal{V}_R \subseteq \mathcal{V}$, $\mathcal{E}_R \subseteq \mathcal{E}$ and $u, v \in \mathcal{V}_R^2$. We define the partial automorphism of the graph $\mathcal{G}$:

$$\pi_R : \begin{array}{ccc} \mathcal{V}_R & \longrightarrow & \mathcal{V}_R \\ x & \longmapsto & \pi_R(x) = \begin{cases} \pi(x) \text{ if } \pi(x) \in \mathcal{V}_R \\ x \text{ otherwise} \end{cases} \end{array}$$

Then, for all $x \in \mathcal{V}_R$ s.t. $(x, u) \in \mathcal{E}_R, (\pi_R(x), \pi_R(u)) \in \mathcal{E}_R \Rightarrow (\pi_R(x), v) \in \mathcal{E}_R \Rightarrow (\pi_R(x), v) \in \mathcal{E}$ because $\mathcal{V}_R \subseteq \mathcal{V}$ and $\mathcal{E}_R \subseteq \mathcal{E}$. Finally, $\forall x \in \mathcal{V}_R, \pi_R(x) = \pi(x)$ then all orbits are included in INC-equivalence, which concludes the proof. □

C.2 Completeness and Soundness of Pruning

Theorem 2 states that the structural properties capture all the nodes that can be pruned out. We consider two cases: centrality measures that depend only on incoming edges (Lemma 2) and centrality measures that depend on both incoming and outgoing edges (Lemma 3).

Lemma 2 (Completeness – Measures based on incoming edges). *A node that can be pruned out without any impact on the spectral centrality measure is either a constant node, a dangling node, or is* INC*-equivalent nodes to a node that is not pruned out.*

Proof (Sketch). The proof consists in a *reductio ad absurdum*. Let assume that there is a node that is not constant, not dangling, not INC-equivalent to any other node, but that can still be pruned out without any impact on the centrality measure. First, being non constant implies that its centrality score changes over the iterations. Hence, its centrality score must be computed at least during the final iteration. Second, being non dangling implies that its centrality score is input at each iteration in the computation of the centrality score of at least one other node. Hence, its non constant centrality score must be computed during all the iterations in order to be used as an input. Third, having no INC-equivalent nodes for a given node implies that the computation of its centrality score is not performed for another node. As a result, pruning out this node would impact the centrality measure. This is a contradiction. □

Lemma 3 (Completeness – Measures based on incoming and outgoing edges). *A node that can be pruned out without any impact on the spectral centrality measure is equivalent (node orbit) to a node that is not pruned out.*

Proof (Sketch). The proof is similar. Let assume that there is a node that is not equivalent to any other node, but that can still be pruned out without any impact on the centrality measure. Having no equivalent node for a given node implies that the computation of its centrality score is not performed for another node. As a result, pruning out this node would impact the centrality measure. This is a contradiction. □

Theorem 3 states that prunable nodes are either constant, dangling, or have the same centrality score as a node that is not pruned out. The proof is sketched below.

Proof (Sketch). By definition, constant nodes and dangling nodes are prunable. We focus only on equivalent nodes. The proof is similar for both spectral centrality measures that depend on incoming and outgoing edges and spectral centrality measures that depend on incoming edges only. The only difference is the graph on which the equivalence relationship is defined. First, for spectral centrality measures that depend on both incoming and outgoing edges, a prunable node is in the node orbit of a non-pruned node (by definition). This implies that the two nodes can be permuted by an automorphism without any impact on the structure of the graph. As a result, they have the same spectral centrality score. Second, for spectral centrality measures that depend only on incoming edges, a prunable node is INC-equivalent nodes to a non-pruned node (by definition). This implies that the two nodes can be permuted by an automorphism without any impact on the structure of the incoming *subgraph*. As a result, since their centrality score only depends on incoming edges, they have the same spectral centrality score. □

D PINC

D.1 Description

Intuitively, the pattern of a node represents the nodes and the weights involved in the computation of its PageRank score. The pattern vector of a graph $\mathcal{G}$ is a vector P of length $|\mathcal{V}|$ in which $P[u]$ stores the pattern of node u. Therefore, if $P[u] = P[\tilde{u}]$, then at each iteration t, the PageRank score of u is equal to the PageRank score of $\tilde{u}$). Algorithm 1 elaborates on this principle. Nodes with identical patterns are grouped together and a single representative node per group is retained. Then the patterns of the nodes they connect to is updated by replacing all the nodes in the same group by the representative node and by accumulating the weights. These operations are repeated until no further grouping is possible. In Algorithm 1, the vector of groups G is a vector of size $|\mathcal{V}|$, where $G[u]$ represents the set of INC-equivalent nodes of u. During the execution of PINC, the size of $\mathcal{V}_p$ decreases by removing the equivalent nodes. Thus, the size of the vectors P and G will decrease too. Algorithm 1 stops when all groups of G contain a single node.

D.2 Toy Example

We present a toy example of the execution of Algorithm 1 on a small graph.

At initialization, we have the following pattern vector P_0:

$$P_0 = \big\{\{(5, \tfrac{1}{3})\}, \{(5, \tfrac{1}{3})\}, \{(0, \tfrac{1}{1}), (1, \tfrac{1}{2})\}, \{(2, \tfrac{1}{2}), (6, \tfrac{1}{2})\}, \{(2, \tfrac{1}{2}), (6, \tfrac{1}{2})\}, \varnothing, \{(1, \tfrac{1}{2}), (7, \tfrac{1}{1})\}, \{(5, \tfrac{1}{3})\}\big\}$$

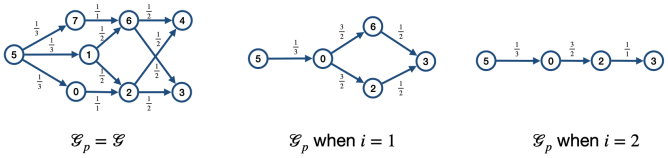


Figure 16: An illustration of the pruning process on a small graph using Algorithm 1

To be more explicit, let's list the pattern vector P_0 for each node of $\mathcal{G}$:

$$P_0[0] = \{(5, \tfrac{1}{3})\}, \quad P_0[1] = \{(5, \tfrac{1}{3})\},$$
$$P_0[2] = \{(0, \tfrac{1}{1}), (1, \tfrac{1}{2})\}, \quad P_0[3] = \{(2, \tfrac{1}{2}), (6, \tfrac{1}{2})\},$$
$$P_0[4] = \{(2, \tfrac{1}{2}), (6, \tfrac{1}{2})\}, \quad P_0[5] = \varnothing,$$
$$P_0[6] = \{(1, \tfrac{1}{2}), (7, \tfrac{1}{1})\},$$
$$P_0[7] = \{(5, \tfrac{1}{3})\}$$

Then at iteration 1, we group the nodes sharing the same pattern:

P_0	$(5, \frac{1}{3})$	$(0, 1), (1, \frac{1}{2})$	$(2, \frac{1}{2}), (6, \frac{1}{2})$	$\varnothing$	$(1, \frac{1}{2}), (7, 1)$
G_1	{0, 1, 7}	{2}	{3, 4}	{5}	{6}

After choosing the representative node for each group (for the sake of simplicity, let's say the smallest node in the group), we apply three consecutive changes on the pattern vector P :

(1) Erase the pattern of the non-representative nodes :

$$P_1 = \big\{\{(5, \tfrac{1}{3})\}, \cancel{\{(5, \tfrac{1}{3})\}}, \{(0, \tfrac{1}{1}), (1, \tfrac{1}{2})\}, \{(2, \tfrac{1}{2}), (6, \tfrac{1}{2})\}, \cancel{\{(2, \tfrac{1}{2}), (6, \tfrac{1}{2})\}}, \varnothing, \{(1, \tfrac{1}{2}), (7, \tfrac{1}{1})\}, \cancel{\{(5, \tfrac{1}{3})\}}\big\}$$

(2) Replace all the occurences of the non-representative nodes by their representant :

$$P_1 = \big\{\{(5, \tfrac{1}{3})\}, \cancel{\{(5, \tfrac{1}{3})\}}, \{(0, \tfrac{1}{1}), (0, \tfrac{1}{2})\}, \{(2, \tfrac{1}{2}), (6, \tfrac{1}{2})\}, \cancel{\{(2, \tfrac{1}{2}), (6, \tfrac{1}{2})\}}, \varnothing, \{(0, \tfrac{1}{2}), (0, \tfrac{1}{1})\}, \cancel{\{(5, \tfrac{1}{3})\}}\big\}$$

(3) Update the patterns and sum the weights of the representative nodes in P:

$$P_1 = \big\{\{(5, \tfrac{1}{3})\}, \cancel{\{(5, \tfrac{1}{3})\}}, \{(0, \tfrac{3}{2})\}, \{(2, \tfrac{1}{2}), (6, \tfrac{1}{2})\}, \cancel{\{(2, \tfrac{1}{2}), (6, \tfrac{1}{2})\}}, \varnothing, \{(0, \tfrac{3}{2})\}, \cancel{\{(5, \tfrac{1}{3})\}}\big\}$$

At iteration 2, we group nodes according to P_1:

P_1	$(5, \frac{1}{3})$	$(0, 1), (1, \frac{1}{2})$	$(2, \frac{1}{2}), (6, \frac{1}{2})$	$\varnothing$
G_2	{0}	{2, 6}	{3}	{5}

Which leads to the following sequence of pattern vector P_2 :

(1) Erase

$$P_2 = \big\{\{(5, \tfrac{1}{3})\}, \cancel{\{(5, \tfrac{1}{3})\}}, \{(0, \tfrac{3}{2})\}, \{(2, \tfrac{1}{2}), (6, \tfrac{1}{2})\}, \cancel{\{(2, \tfrac{1}{2}), (6, \tfrac{1}{2})\}}, \varnothing, \cancel{\{(0, \tfrac{3}{2})\}}, \cancel{\{(5, \tfrac{1}{3})\}}\big\}$$

(2) Replace

$$P_2 = \big\{\{(5, \tfrac{1}{3})\}, \cancel{\{(5, \tfrac{1}{3})\}}, \{(0, \tfrac{3}{2})\}, \{(2, \tfrac{1}{2}), (2, \tfrac{1}{2})\}, \cancel{\{(2, \tfrac{1}{2}), (6, \tfrac{1}{2})\}}, \varnothing, \cancel{\{(0, \tfrac{3}{2})\}}, \cancel{\{(5, \tfrac{1}{3})\}}\big\}$$

(3) Update

$$P_2 = \{\{(5, \tfrac{1}{3})\}, \cancel{\{(5, \tfrac{1}{3})\}}, \{(0, \tfrac{3}{2})\}, \{(2, \tfrac{1}{1})\},$$
$$\cancel{\{(2, \tfrac{1}{2}), (6, \tfrac{1}{2})\}}, \varnothing, \cancel{\{(0, \tfrac{3}{2})\}}, \cancel{\{(5, \tfrac{1}{3})\}}\}$$

At iteration 3, we group nodes according to P_2:

P_2	$(5, \frac{1}{3})$	$(0, \frac{3}{2})$	$(2, \frac{1}{1})$	$\varnothing$
G_3	$\{0\}$	$\{2\}$	$\{3\}$	$\{5\}$

Since each group has only one node, we stop the algorithm and return the pruned graph $\mathcal{G}_p$ which is the graph with only the nodes in $\{0, 2, 3, 5\}$ as illustrated in Figure 16.

D.3 Proofs

D.3.1 Soundness. Lemma 1 shows that two nodes with the same pattern, as defined in Definition 3, are INC-equivalent nodes.

In the following, let G_0 be the set of group classes at initialization and G_i the vector of group classes at iteration i of PINC. Similarly, let P_0 be the pattern vector at initialization and P_i the pattern vector at the end of iteration i of PINC. Recall that $\mathcal{G}_p = \mathcal{G}$ at iteration 1, and that $\mathcal{G}_p$ is modified at the end of each iteration of PINC.

Proof. During the following proof, *merging* u and v means keeping a representative (for example node u) in the pruned graph $\mathcal{G}_p$ and removing the other node (for example node v).

The pattern of node u is defined as the set of pairs (s, w) for each edge $(s, u) \in \mathcal{E}_p$ where

$$w = \sum_{v \in C(s)} \frac{1}{deg_{out}(v)}$$

and $C(s)$ represents the set of nodes in the group of node s in Algorithm 1. Our proof is by induction. At iteration 1, Algorithm 1 merges two nodes u and v with the same pattern (i.e. if $P_0[u] = P_0[v]$). They have the same pattern if they have the same set of parents and the same weight. We start by considering the parents. By definition, if nodes u and v have the same set of parents then:

$$\forall x \in \mathcal{V}_p, (x, u) \in \mathcal{E}_p \Leftrightarrow (x, v) \in \mathcal{E}_p$$

We consider the weight now. By definition, if nodes u and v have the same weight then:

$$\sum_{(y,u) \in \mathcal{E}_p} \sum_{\substack{z \in C(y) \\ \text{s.t. } (z,u) \in \mathcal{E}_p}} \frac{1}{deg_{out}(z)} = \sum_{(y,v) \in \mathcal{E}_p} \sum_{\substack{z \in C(y) \\ \text{s.t. } (z,v) \in \mathcal{E}_p}} \frac{1}{deg_{out}(z)}$$

Since at iteration 1, no node has been merged ($\forall x \in \mathcal{E}, |C(x)| = 1$) we have the following equality of weights:

$$\sum_{(y,u) \in \mathcal{E}_p} \frac{1}{deg_{out}(y)} = \sum_{(y,v) \in \mathcal{E}_p} \frac{1}{deg_{out}(y)}$$

Now, lets try to find an automorphism, i.e, a way to map the nodes of the graph onto each other while keeping all edges unchanged. Using Definition 2, the induced graph $\mathcal{G}_R$ constructed from nodes u and v is equal to $\mathcal{G}_R = (\mathcal{V}_R, \mathcal{E}_R)$ with $\mathcal{V}_R = \{u, v\} \cup \mathcal{S}_{(u,v)}$ and $\mathcal{E}_R$ is, by definition, the set of edges between the nodes of $\mathcal{V}_R$. Moreover, we observe that the set of glcas of u and v is the set of their direct parents. Indeed, if $P_0[u] \neq P_0[v]$ then one of the nodes would have a parent that the other does not. This means u and v must have the same set of parents which, by Definition 1, is the set of glcas. Then nodes u and v are trivially permutable in $\mathcal{G}_R$ and the mapping

$$\pi(u) = v, \quad \pi(v) = u, \quad \pi(x) = x \text{ for all } x \in \mathcal{V}_R \setminus \{u, v\}$$

is an automorphism of $\mathcal{G}_R$. As a result, u and v are INC-equivalent nodes. To conclude, at iteration 1, two nodes with the same pattern are INC-equivalent nodes.

Let assume now that the implication holds for iteration n and prove that it holds as well for iteration $n + 1$.

At iteration $n + 1$, let nodes u and $v \in \mathcal{G}_p$ be two nodes with the same pattern. We start by considering the parents of nodes u and v. By definition:

$$\forall x \in \mathcal{V}_p, (x, u) \in \mathcal{E}_p \Leftrightarrow (x, v) \in \mathcal{E}_p$$

Then we consider the weigth. By definition:

$$\sum_{(y,u) \in \mathcal{E}_p} \sum_{\substack{z \in C(y) \\ \text{s.t. } (z,u) \in \mathcal{E}_p}} \frac{1}{deg_{out}(z)} = \sum_{(y,v) \in \mathcal{E}_p} \sum_{\substack{z \in C(y) \\ \text{s.t. } (z,v) \in \mathcal{E}_p}} \frac{1}{deg_{out}(z)}$$

For node $y \in \mathcal{V}_p$, for each node $z \in \mathcal{V}$ s.t. $z \in C(y)$ and $z \neq y$ then (1) node z has been removed from $\mathcal{G}_p$ at the end of iteration n (y and z has been merged in $\mathcal{G}_p$) and (2) the pattern of each node a such that $(y, a) \in \mathcal{E}_p$ has been updated as follow: $P_n[a].weight+ = \frac{1}{deg_{out}(z)}$.

Now lets try to find a new automorphism such that nodes u and v are in the same group. Let $\mathcal{V}_R = \{u, v\} \cup \mathcal{S}_{(u,v)}$, and let $\mathcal{G}_R$ be the subgraph induced by $\mathcal{V}_R$. We have assumed that all nodes that were merged in earlier iterations are INC-equivalent nodes, so replacing them by their representative in $\mathcal{G}_p$ preserves the structure and the weight of the edges in $\mathcal{G}_R$. Because $P_n[u] = P_n[v]$, u and v have the same set of parents in $\mathcal{G}_R$ and contribute exactly the same weights. Hence, the mapping

$$\pi(u) = v, \quad \pi(v) = u, \quad \pi(x) = x \text{ for all } x \in \mathcal{V}_R \setminus \{u, v\}$$

is an automorphism of $\mathcal{G}_R$ preserving adjacency and weights. Therefore u and v are INC-equivalent nodes at iteration $n + 1$.

To conclude, for each iteration $n \geqslant 1$, if $P_{n-1}[u] = P_{n-1}[v]$ then u and v are INC-equivalent nodes. □

Theorem 4 states that each node pruned from the graph $\mathcal{G}$ is INC-equivalence to a single non-pruned node in $\mathcal{G}_p$. We prove this theorem below.

Proof. The proof of Theorem 4 follows directly from Lemma 1.

Our proof is by induction. At iteration 1, to construct G_1 we merge nodes with the same pattern in P_0. Then, $\forall u \in \mathcal{V}$,

$$G_1[u] = \{x : \forall x \in \mathcal{V}, P_0[x] = P_0[u]\}$$

is the group class of node u. From Lemma 1, two nodes with the same pattern are INC-equivalent nodes. We conclude that each node in the group class of u, i.e, in $G_1[u]$, is INC-equivalent with u. Then we chose a representative and remove all other nodes in $\mathcal{V}_p$.

Let assume that the implication holds for iteration n and prove that it holds as well for iteration $n + 1$. $\forall u \in \mathcal{V}_p$

$$G_i[u] = \{x : \forall x \in C(u)\}$$

is the INC-equivalence classe of node u. It holds that two nodes in G_i are INC-equivalent nodes if they have the same pattern in P_{i-1},

i.e. if they share the same ancestors and the same weights. Thus, $\forall u \in \mathcal{V}_p, G_i[u]$ is equals to

$$G_i[u] = \{x : \forall x \in \mathcal{V}_p, P_{i-1}[x] = P_{i-1}[u]\}$$

The pattern of node $u \in \mathcal{V}_p$, $(v, u) \in \mathcal{E}_p$ is equals to

$$P_i[u] = \{(v, w(v))\}$$

At iteration $n + 1$, to construct G_{n+1} we merge nodes with the same pattern in P_n. Then, $\forall u \in \mathcal{V}_p$,

$$G_{n+1}[u] = \{x : \forall x \in \mathcal{V}_p, P_n[x] = P_n[u]\}$$

is the group class of node u. From Lemma 1, two nodes with the same pattern are INC-equivalent nodes. We can conclude that each node in the group class of u is INC-equivalent with u. Then we can chose a representative and remove all other nodes in $\mathcal{V}_p$. Each pruned node in $\mathcal{V} \setminus \mathcal{V}_p$ has a single representative in $\mathcal{V}_p$. □

D.3.2 Completeness. Theorem 5 states that PINC is complete when excluding INC-equivalent nodes that fall within a cycle, i.e., each INC-equivalent nodes pruned from PINC shares the same pattern with a non pruned node. The proof for this theorem his very similar to the proof of Lemma 1 using an induction.

Proof (Sketch). The proof consists of a recurrence. Two INC-equivalent nodes after the end of the first iteration of PINC necessarily shares the same set of parents with the same weights. By definition, they have the same pattern. Assume that each INC-equivalent nodes at the end of iteration n, shares the same pattern. At iteration $n + 1$, if two nodes are INC-equivalent nodes they necessarily have the same set of parents and the same weigth in $\mathcal{G}_p$ because PINC keep only one representative per INC-equivalence class at the end of each iteration. □

E Synthetic Datasets

Table 8 summarizes the characteristics of the synthetic datasets.

Graph	$\lvert\mathcal{V}\rvert$	$\lvert\mathcal{E}\rvert$	k_{max}	|Cons.|	|Dang.|	Density
synthetic-1000	1,000	1,816	278	748	111	0.0018
synthetic-5000	5,000	9,434	1110	3,798	548	0.0004
synthetic-10000	10,000	19,232	2168	7,573	1,106	0.0002
synthetic-15000	15,000	28,961	3121	11,350	1,677	0.0001
synthetic-20000	20,000	39,004	4037	15,149	2,190	0.0001
synthetic-25000	25,000	49,064	4936	18,947	2,731	0.0001
synthetic-30000	30,000	58,825	5824	22,737	3,312	0.0001
synthetic-35000	35,000	68,793	6676	26,540	3,844	0.0001

Table 8: Synthetic Datasets, where |Const.| (resp. |Dang.|) denotes the number of constant (resp. dangling) nodes.

F Relative Error of Poppy

Figure 17 displays the cumulative distribution functions of the relative error for Poppy on graph p2p-Gnutella04. We observe that 99% of PageRank scores exhibit a relative error below $8.85 \cdot 10^{-2}$ and 90% below $9.65 \cdot 10^{-3}$.

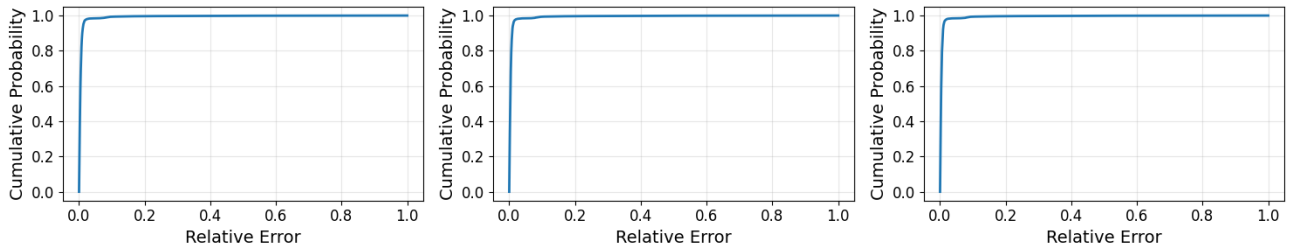
G Convergence Rate

Figure 18 displays the average, maximum and minimum of PageRank plaintext scores over iterations for different teleportation denominators and across different graphs. We chose to present the convergence rate for a single synthetic graph (synthetic-35000), as the similar structures of the synthetic graphs leads to comparable convergence patterns regardless of the number of nodes. The choice of the teleportation denominator has only a limited impact on the convergence speed, but significantly affects the range of the PageRank scores. For most graphs, convergence is reached before iteration 10. The email-Eu-core graph is the only dataset that does not converge completely within the first 10 iterations: the maximum PageRank score converges after around 20 iterations. Using 10 iterations in our experiments provides a reasonable trade-off between computational cost and ranking stability. Figure 19 shows the same metrics (average, maximum and minimum) for both plaintext and encrypted PageRank over iterations. We observe that computing PageRank with Poppy has no impact on the convergence speed. Indeed, all curves are nearly overlapping for all metrics.

H Comparison with [45]

Figure 20 compares Poppy with [45] on synthetic-25000, in terms of computation cost and communication cost per iteration, for the slowest data center. Note that we do not include Poppy_D to this figure because Poppy_D is not competitive on synthetic-25000. The implementation of [45] is not available so we have reimplemented [45] in Python. We observe that the computation cost and the communication cost per party remain constant across iterations for [45], while Poppy's costs decrease during the first iterations until it reaches a plateau. After reaching this plateau, Poppy_S becomes competitive in terms of computation cost compared to [45]. However, [45] has a lower communication cost (KiB) at all iterations. Figure 21 displays the communication in KiB for Poppy and [45] for various graphs. We observe that the communication cost of [45] is linear in the size of the graph while Poppy's is constant before reaching the maximum number of slots available. Indeed, Poppy and [45] differ fundamentally on the structure of their ciphertexts. Poppy uses CKKS that has SIMD capabilities and that is such that the number of slots used in a ciphertext has no impact on the ciphertext size. [45] is based on the "think-as-a-vertex" paradigm, which leads to a number of ciphertexts exchanged at each iteration equal to the number of nodes in the full graph. Thanks to this observation, we simulated the communication of [45] for synthetic-300000 by multiplying by 10 the size (in KiB) of the ciphertexts communicated per iteration during an execution of [45] on synthetic-30000 – see Figure 21d. In this setting, assuming that the graph is uniformly distributed among 5 data centers (leading to 60,000 nodes per data center), the number of slots in a CKKS ciphertext becomes close to the maximum number of slots available (i.e., 65,536 slots at most), and we expect Poppy to be competitive with [45] after 4 to 6 iterations depending on the Poppy algorithm used. As a result, as the size of the graph increases and the CKKS ciphertexts become fully exploited, the communication gap reverses, and Poppy becomes competitive. Furthermore, since Poppy exhibits decreasing computational and communication costs across iterations, it is expected to be more efficient on graphs with slower convergence rates. Finally,

we stress that additively homomorphic encryption does not support homomorphic division, preventing [45] to support normalization in PageRank and in any other spectral centrality measures, like HITS (to which normalization is required for convergence).



(a) PoppyS without pruning. (b) PoppyD without pruning. (c) PoppyH without pruning.

Figure 17: Cumulative distribution of relative errors on `p2p-Gnutella04` (config. `C3`).

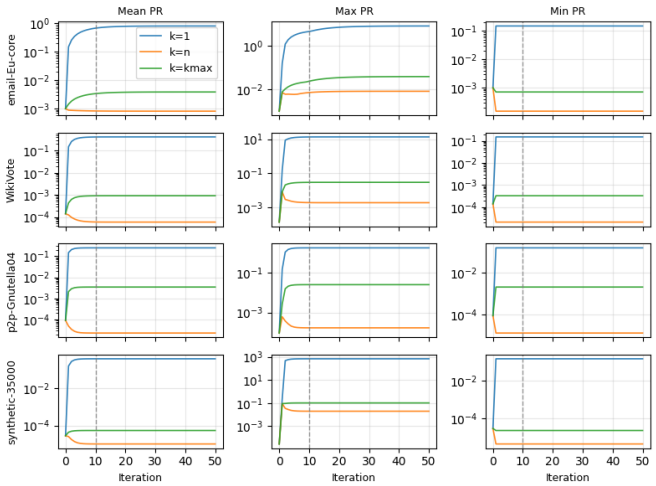


Figure 18: Impact of teleportation denominator on convergence rate.

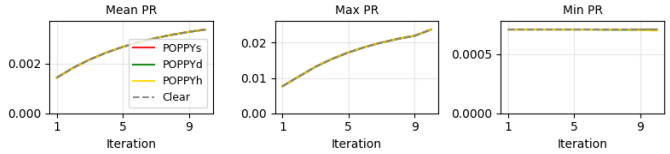


Figure 19: Impact of encryption on convergence for `email-Eu-core`.

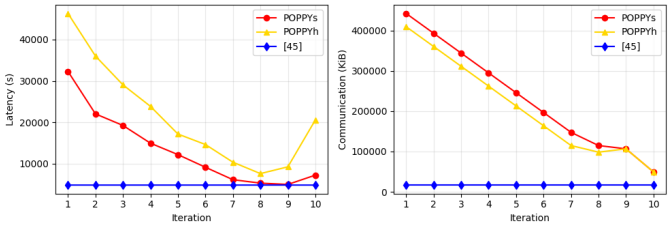
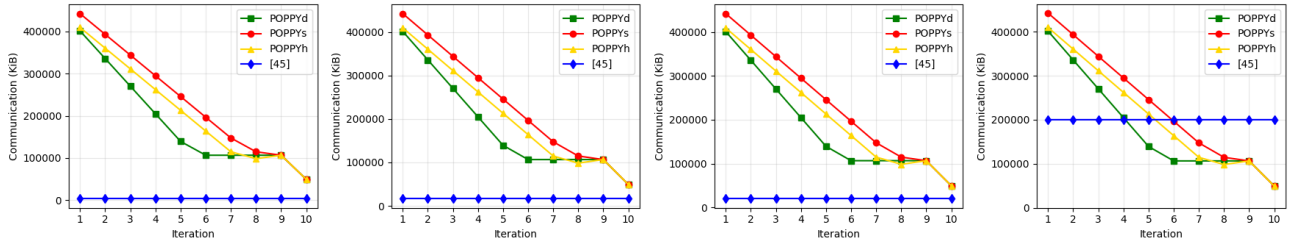


Figure 20: Maximum latency and communication (slowest data center) on `synthetic-25000` (config. `C1`).



(a) `p2p-Gnutella04` (b) `synthetic-25000` (c) `synthetic-30000` (d) `synthetic-300000` (simulated)

Figure 21: Maximum communication (slowest data center) on various graphs (config. `C1`).