PPGA: A Framework for Privacy-Preserving Graph Algorithms using the Prepare/Propagate/Gather/Apply Paradigm

Andreas Brüggemann, Simon Brunnauer, and Thomas Schneider

Technical University of Darmstadt, Germany
brueggemann@encrypto.cs.tu-darmstadt.de simon.brunnauer@stud.tu-darmstadt.de
schneider@encrypto.cs.tu-darmstadt.de

Abstract. Privacy-preserving graph computation enables the secure analysis of graphs modeling social interactions and networks, financial transactions, etc., while keeping the graph private. One approach is to use secure multiparty computation (MPC), allowing different clients to contribute parts of a graph without need of disclosure, and multiple MPC protocols have been proposed for running message-passing-based graph analysis. Yet, existing public implementations are prototypes for fixed analysis functions that require substantial MPC expertise to modify and extend. We present *PPGA*, a practical framework based on the MultiCent protocol (PETS'25). PPGA lets users define their custom message-passing graph algorithms with a high-level interface, clients input parts of the graph, and then runs the algorithm in MPC. It significantly improves performance compared to MultiCent's implementation, running BFS on a graph of size $|\mathsf{V}|+|\mathsf{E}| = 30$ million in 23 minutes without leaking the graph topology.

1 Introduction

Graph analysis is a highly useful tool for many applications, such as social network analysis [OR02], fraud detection [SvA+19], or contact tracing [AFO+21]. The graphs to be analyzed often model private and sensitive data, calling for a privacy-preserving solution. Yet, realistic graphs can be large, requiring such a solution to be scalable. Scalability can be achieved with message-passing algorithms, where nodes iteratively pass their state to their outgoing edges, after which each node updates its internal state considering the data that it receives from its incoming edges. This was made privacy-preserving by a line of works [NWI+15,AFO+21,KKPG24,BKKS25] utilizing secure multiparty computation (MPC). Here, MPC provides the solution that enables different clients to provide their parts to a larger joint graph without need of disclosure, outsourcing the computation to a few compute parties which, on their own, learn nothing but the graph size in the process.

The foundation for message-passing algorithms in MPC is given by [NWI+15], high scalability has been achieved by [AFO+21], and this has later been further optimized by [KKPG24] and finally [BKKS25], providing the current state of the art in an MPC outsourcing setting. Other MPC message-passing protocols such as [ZLS+24,ZLS+25,KKK+25] further optimize performance by assuming that different parties each locally hold subgraph inputs and also run the computation, enabling them to leverage plaintext knowledge of parts of the joint graph. However, requiring each input provider to also be a party running the computation can be quite restrictive. First, it prevents lightweight clients to simply provide their inputs once, but requires all of them to have sufficient hardware resources and ongoing network connectivity. Second, it scales poorly to many input providers, which can exclude applications such as contact tracing.

The tooling to apply outsourced privacy-preserving graph analysis is quite sparse: [AFO+21] has no public implementation, and [KKPG24] only implements some building blocks for benchmarking and no end-to-end solution. The code and protocol of [NWI+15] suffer from lower scalability, and the state-of-the-art protocol from [BKKS25] provides an end-to-end implementation of the core graph analysis, but only as a proof of concept for computing certain centrality measures, without a clear interface for generalization and with subpar performance of the implementation.

Our Contributions. In this work, we introduce *PPGA*, a practical C++ framework, publicly available under the MIT license at `https://encrypto.de/code/PPGA-Graph-MPC`, that enables to run private message-passing style graph algorithms. The contributions of PPGA are as follows:

Intuitive high-level interface to implement custom message-passing algorithms: PPGA provides a well-documented, easy-to-use interface, enabling its users to implement custom message-passing algorithms.

The user implementation defines which data is sent over edges and how node states are updated after receiving messages from incoming edges. No in-depth MPC knowledge is required, as the backend automatically takes care of computing the message-passing algorithm in a privacy-preserving way.

State-of-the-art MPC: The backend of PPGA implements the state-of-the-art MultiCent protocol [BKKS25] for running the user-defined graph algorithms in MPC, outsourcing computation to two servers, assisted by an additional dealer that provides preprocessing material.

Privacy: Assuming non-colluding honest-but-curious servers and dealer, neither the servers nor the dealer learn any information besides the following public parameters: the numbers of nodes and edges, the node representation size, the chosen algorithm, and the number of iterations.

Presets for multiple algorithms: PPGA contains example implementations using its high-level interface, enabling to compute breadth-first-search (BFS) and three different centrality measures considered in [BKKS25], including Katz centrality [Kat53], out of the box.

Input/Output Interface: PPGA provides a simple way for clients to provide secret-shared inputs and receive outputs, which was missing in [BKKS25].

High efficiency: PPGA inherits the communication efficiency of [BKKS25], but our better implementation improves run time by 1.6× and memory utilization by 6.0× over [BKKS25]. This allows to scale privacy-preserving breadth-first search (BFS) to graph sizes[1] up to 30 million, at only 23 minutes of total run time.

2 Graph Analysis in the PPGA Paradigm

We begin by explaining the *Prepare/Propagate/Gather/Apply Paradigm*, based on [NWI+15,BKKS25], serving as our message-passing abstraction for graph analysis in which we formulate our graph algorithms. Using this paradigm, nodes iteratively broadcast messages depending on their current state to all outgoing edges, and nodes receiving incoming messages use them to update their own state. Fig. 1 illustrates this paradigm and we explain it in detail next.

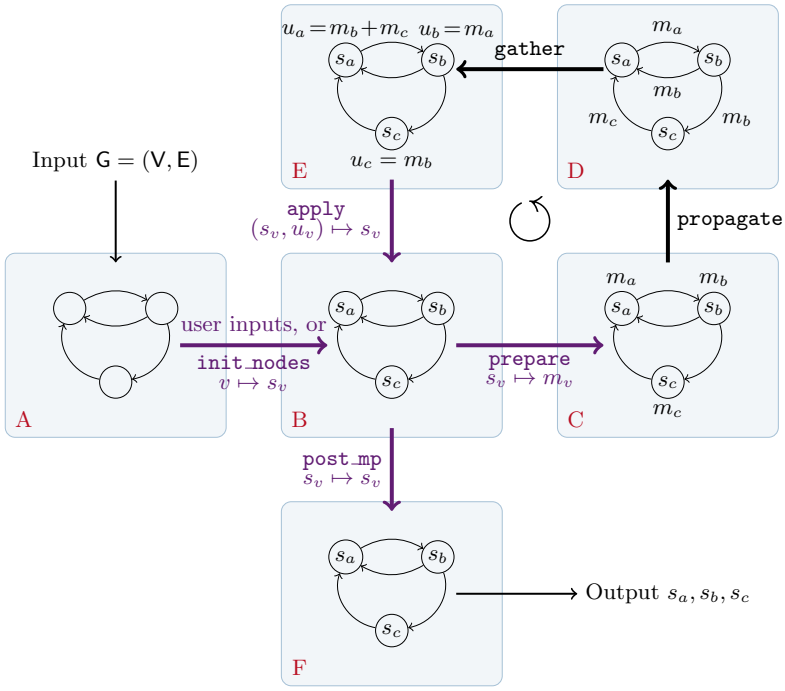


Fig. 1. The abstract message-passing performed by Prepare/Propagate/Gather/Apply (PPGA). Customizable operations are highlighted in purple.

[1] the combined number of nodes and edges

After the graph is provided by the clients (A in Fig. 1), the state s_v of each node v is initialized. PPGA supports clients providing the initial states, or letting the algorithm set them to fixed initial values using the customizable function `init_nodes`. After each node has been assigned a state (B), the customizable function `prepare` computes from each state s_v the message m_v (C) to then be propagated to all of v's outgoing edges (D). PPGA then gathers an update value u_v for each node v, which is the sum over the messages on all incoming edges (E).[2] Finally, the function `apply` customizes how, given the prior state s_v and the update value u_v, the new state is computed for each node. The process of preparing, propagating, gathering, and applying is repeated up to a parametrizable iteration depth. Finally, the node states are revealed as outputs, after the customizable function `post_mp` enables to run some post-processing if desired.

2.1 Example: Breadth-First Search (BFS)

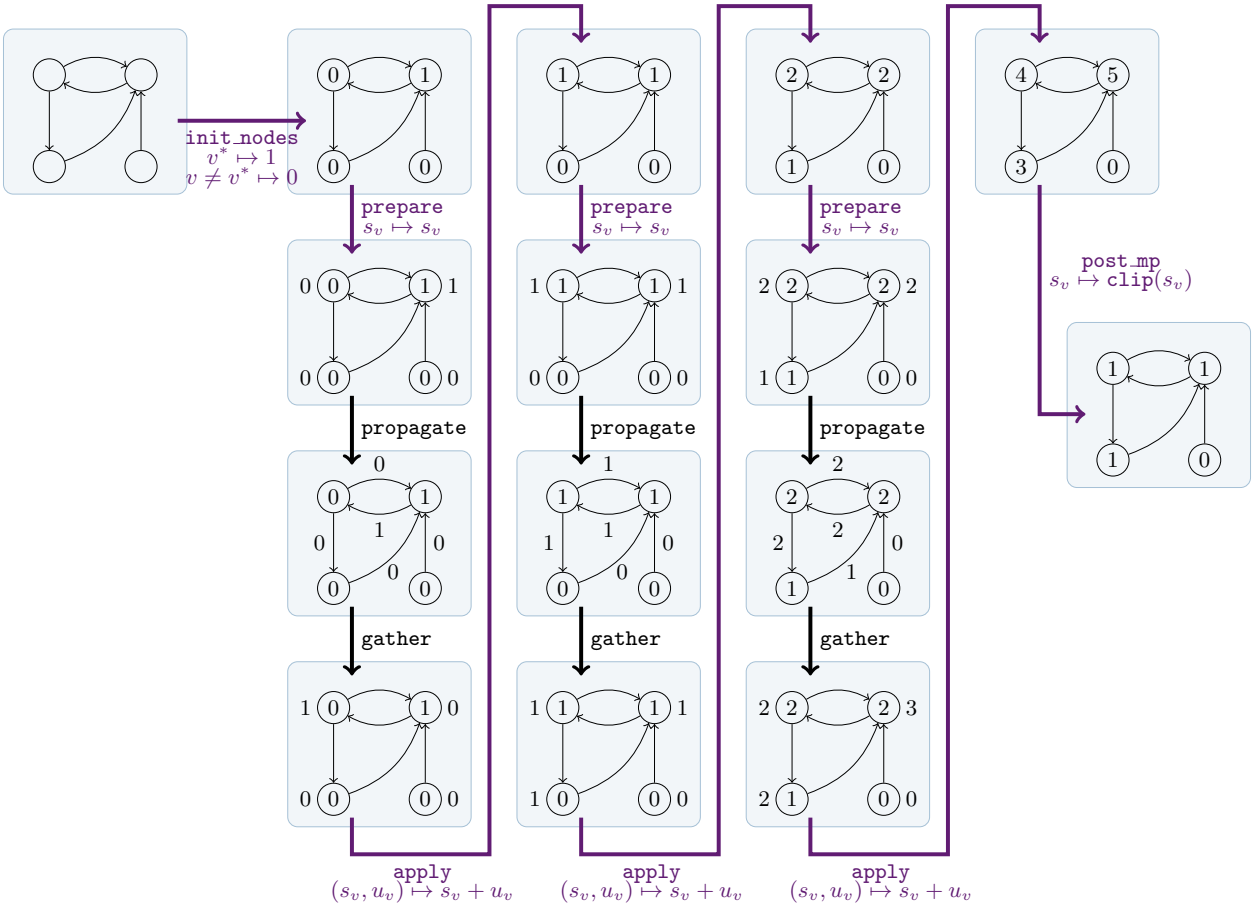


Fig. 2. Illustration of running depth $d = 3$ BFS using 3 iterations of message-passing in the PPGA paradigm.

A classic example for a graph algorithm that can be expressed via message-passing is breadth-first search (BFS). Its goal is to identify all nodes that can be reached from a starting node v^* with a distance of

[2] Only supporting to compute the sum among many possible aggregation function enables highly efficient message-passing following the techniques of [KKPG24], while it also currently is the main functional limitation of PPGA. One may extend PPGA to support non-linear aggregation functions, e.g., the max function, following the approach from [AFO+21]. Yet, that would significantly decrease performance and increase the backend's complexity.

at most d edges for some depth-parameter d. The algorithm iteratively identifies all nodes with distance 1, then all nodes with distance 2, etc. To achieve that using the PPGA paradigm, we first let `init_nodes` define node states s_v with $s_{v^*} = 1$ and $s_v = 0$ for all $v \neq v^*$, with 1 indicating that a node has been reached. In each iteration, all nodes propagate their current state, i.e., $\texttt{prepare}(s_v) = s_v$. Each node v now receives $u_v > 0$ iff any of its neighbors has been reached before and now passed its state to v. Thus, v has been reached if $s_v > 0$, i.e., it has been reached before, or if $u_v > 0$, so we simply define $\texttt{apply}(s_v, u_v) = \texttt{clip}(s_v + u_v)$ where $\texttt{clip}(x) = 0$ if $x = 0$ and $\texttt{clip}(x) = 1$ otherwise. As observed in [KKPG24], we can relax the definition of node states to $s_v > 0$ iff v has been reached (instead of $s_v = 1$), clipping values only a single time, namely after the final iteration, which behaves equivalently if there are no integer overflows. This reduces the number of clipping operations, increasing performance as clipping comes at the expense of requiring additional communication between the servers. Hence, we set $\texttt{apply}(s_v, u_v) = s_v + u_v$ and $\texttt{post_mp}(s_v) = \texttt{clip}(s_v)$. A concrete example for running BFS using PPGA is provided in Fig. 2.

3 Our PPGA Code Framework

The architecture of our PPGA framework is depicted in Fig. 3. It consists of three main components, which work as follows:

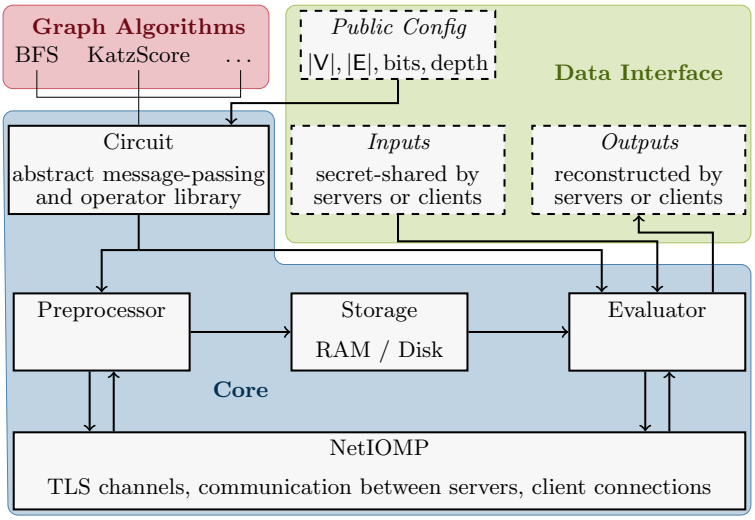


Fig. 3. Architecture of the PPGA framework.

1) The *Core* component (blue in Fig. 3) is responsible for running the MPC protocol. A *circuit* defines the graph algorithm to be computed. It defines the message-passing in an abstract way (cf. §2), leaving the specification of the individual steps open to the user. In addition, it provides a library of operations such as additions, subtractions, multiplications, and clipping on secret data that the user can use to implement the individual message-passing steps. The *preprocessor* creates correlated randomness w.r.t. the circuit to later speed up the online phase, once the inputs are provided. The correlated randomness is stored (on RAM or optionally disk) and later consumed by the *evaluator* which runs the online phase. The *evaluator* receives the circuit as well as the secret inputs, which are secret-shared between the servers, it then runs the computation, and finally, it provides the outputs by handing shares to the intended output receiver(s). Preprocessor and evaluator use the *NetIOMP* layer, which allows communication between the servers, manages secure channels between them, etc. Importantly, a user of PPGA does not need to modify or understand the core component, but only interacts with the interfaces provided by the circuit to define the graph algorithm and interact with the evaluator to provide inputs and receive outputs.

2) The *Graph Algorithms* component (red in Fig. 3) is where users can implement their custom message-passing algorithms, or can choose from a collection of pre-defined algorithms like BFS or Katz centrality [Kat53]. Each custom algorithm is a subclass of *circuit* and gives concrete instantiations for the message-passing primitives such as `prepare` and `apply` (cf. §2) to instantiate the algorithm.

3) The *Data Interface* component (green in Fig. 3) lets the user and/or input providing clients define some public parameters: the total number of nodes and edges, the number of bits used to represent each node, and the iteration depth. Moreover, it handles the secret inputs (i.e., the structure of the graph), and the outputs (i.e., generally, one value per node) which are handed to and received from the evaluator.

3.1 Technical Improvements Over MultiCent [BKKS25]

From a technical perspective, PPGA optimizes graph computation in multiple regards, compared to the prior state-of-the-art implementation of MultiCent [BKKS25]. The main bottleneck of MultiCent's implementation is memory utilization, resulting from a suboptimal circuit presentation. MultiCent partially uses single instruction multiple data (SIMD) gates decreasing memory utilization, and further gates with many inputs, such as secure shuffling. Yet, it uses one wire per individual scalar secret-shared value, resulting in millions of wires as inputs and outputs of SIMD gates, shuffling gates, etc. For instance, a single shuffling gate for one million values allocates 15.3 MiB as references to the corresponding input and output wires alone. We instead use wires holding arbitrarily sized vectors of secret-shared values, systematically pursuing a SIMD approach, following the design paradigms of generic MPC frameworks such as MOTION [BDST22] and SEEC [DHKS25]. That also enables freeing memory of secret-shares that are not used anymore effectively.[3] We also carefully balance the memory load between servers where the data sent by the dealer is unevenly distributed between the two servers, and move some local operations by the parties, such as PRF calls, from the preprocessing to the online phase, favoring a lower memory footprint over minor online compute improvements. Furthermore, we enable to store preprocessing information on disk to optionally decrease memory utilization at the cost of higher access times. Finally, we integrate parallelization using OpenMP.

3.2 Additionally Supported Functionality

Following [BKKS25], PPGA also supports computation on multigraphs, optionally converting multigraph inputs to simple graphs in cases where inputs may contain unwanted duplicates. It provides a clear interface for passing messages over edges backwards instead of forwards where needed (e.g., for computing Katz centrality [Kat53] as demonstrated in [BKKS25]), and running multiple (possibly separately defined) instances of message-passing on different payloads at once, optionally aggregating in the end using a custom function.

3.3 Implementing BFS Using PPGA's Code Interface

The interface of PPGA enables to intuitively transfer the description of BFS from §2.1 into code, utilizing the library of pre-defined operations provided by the circuit class. An example BFS implementation (with the starting node v^* being the one with index 0) is provided in Listing 1. We note that some functions (`init_nodes`, `prepare`, `post_mp`) have trivial default implementations (using user inputs, simply setting $m_v = s_v$, keeping the final state). Furthermore, some functions also receive the number of the current iteration `i`, and a column index `column` if multiple message-passing instances run in parallel, enabling to also customize the operations per iteration and instance. In comparison, an implementation using the codebase of [BKKS25], excluding any comments, would require approximately 50 lines of code and almost 8× the characters compared to our implementation in PPGA from Listing 1.

[3] This could be added to the original MultiCent too, but would free 4 bytes of memory per scalar value while still requiring 8 bytes for the corresponding (scalar) wire ID.

```
class BFSCircuit : public Circuit {
 public:
  BFSCircuit(ProtocolConfig &conf) : Circuit(conf) {}

 protected:
  // Could also return std::nullopt to use user inputs
  std::optional<mp_val> init_nodes(size_t column) override {
    std::vector<Ring> ohv(nodes, 0);
    ohv[0] = 1; // or any other starting point
    return set_const_vector(ohv);
  }

  //prepare() sets m_v=s_v as default if not overridden

  mp_val apply(mp_val &state, mp_val &update, size_t i, size_t column) override {
    return add(state, update); // s_v + u_v
  }

  mp_val post_mp(mp_val &state,size_t column) override{
    return clip(state);
  }
};
```

Listing 1. Implementation of BFS in PPGA.

4 Benchmarks

We benchmark BFS computation with PPGA on three servers, each with an Intel Core i9-7960X CPU @ 2.8 GHz (32 threads) and 128 GB DDR4 RAM, connected in LAN with 1 ms RTT and 1 GBit/s bandwidth. As PPGA does not leak any information about the graph topology, performance only depends on the public parameters, i.e., the numbers of nodes and edges, bits required to represent the nodes ($\approx \log_2 |\mathsf{V}|$), and the iteration depth, which we set to $d = 10$ here.

The results are provided in Fig. 4 and Table 1. Even for a graph size $|\mathsf{V}| + |\mathsf{E}| = 30$ million, PPGA runs in 23 minutes with total communication of 110 GiB and 105 GiB per-server RAM utilization. For a graph size of one million, it only requires 32 s run time, 3 GiB communication, and 4 GiB RAM utilization. We also implement BFS in MultiCent [BKKS25] for comparison. The main bottleneck of [BKKS25] is RAM utilization, limiting it to a graph size of 5 million where it uses 6× more RAM (103 GiB) and requires 60% higher run time than PPGA.

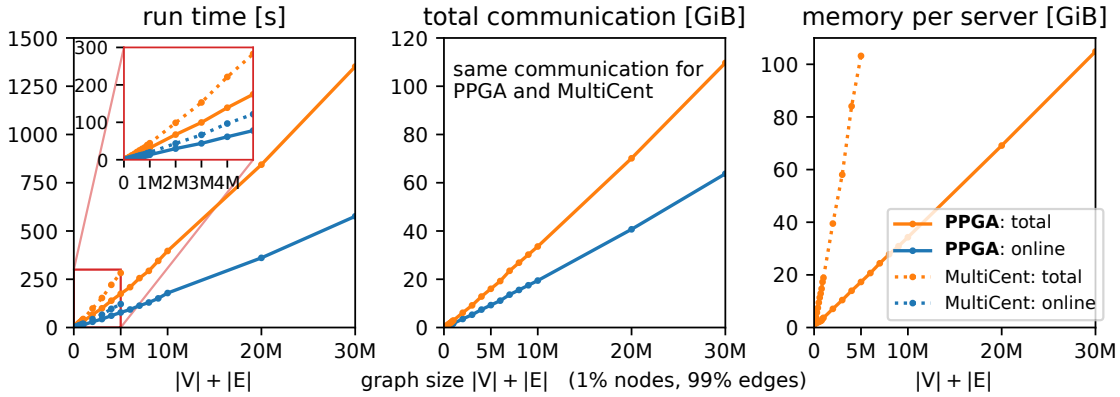


Fig. 4. BFS performance for different graph sizes when evaluated using PPGA compared to the original MultiCent implementation [BKKS25]. Run time and communication given as total as well as only online cost.

Table 1. BFS performance using PPGA or the original MultiCent implementation [BKKS25], including improvement factors of PPGA over MultiCent. Run time and communication given as total values over preprocessing + online; online only values given in brackets. The communication of PPGA and MultiCent is the same as both use the MPC protocol from MultiCent.

graph size	run time						communication		memory utilization per server		
$\|V\| + \|E\|$	**PPGA**		MultiCent [BKKS25]		improvement		**PPGA**/MultiCent		**PPGA**	MultiCent	impr.
100k	2.83 s	(1.25 s)	3.58 s	(1.57 s)	1.26×	(1.25×)	0.23 GiB	(0.13 GiB)	1.39 GiB	2.33 GiB	1.67×
200k	5.86 s	(2.54 s)	7.37 s	(3.32 s)	1.26×	(1.31×)	0.49 GiB	(0.28 GiB)	1.64 GiB	3.94 GiB	2.40×
300k	8.98 s	(3.75 s)	11.68 s	(5.36 s)	1.30×	(1.43×)	0.78 GiB	(0.45 GiB)	1.91 GiB	5.70 GiB	2.99×
400k	11.89 s	(5.02 s)	15.49 s	(7.10 s)	1.30×	(1.41×)	1.05 GiB	(0.60 GiB)	1.97 GiB	7.46 GiB	3.79×
500k	15.80 s	(6.66 s)	20.49 s	(9.41 s)	1.30×	(1.41×)	1.38 GiB	(0.79 GiB)	2.20 GiB	9.46 GiB	4.29×
600k	19.05 s	(8.11 s)	24.87 s	(11.36 s)	1.31×	(1.40×)	1.66 GiB	(0.95 GiB)	2.30 GiB	11.35 GiB	4.93×
700k	21.75 s	(9.14 s)	28.86 s	(13.02 s)	1.33×	(1.42×)	1.93 GiB	(1.11 GiB)	2.51 GiB	12.92 GiB	5.15×
800k	24.47 s	(10.34 s)	32.67 s	(14.74 s)	1.33×	(1.42×)	2.21 GiB	(1.27 GiB)	2.68 GiB	14.73 GiB	5.51×
900k	28.80 s	(12.12 s)	39.15 s	(17.74 s)	1.36×	(1.46×)	2.62 GiB	(1.51 GiB)	3.34 GiB	17.31 GiB	5.18×
1M	31.65 s	(13.30 s)	44.20 s	(19.82 s)	1.40×	(1.49×)	2.91 GiB	(1.68 GiB)	3.67 GiB	18.90 GiB	5.15×
2M	67.18 s	(29.79 s)	98.63 s	(43.54 s)	1.47×	(1.46×)	6.12 GiB	(3.53 GiB)	7.08 GiB	39.49 GiB	5.58×
3M	99.57 s	(43.76 s)	152.99 s	(66.45 s)	1.54×	(1.52×)	9.18 GiB	(5.30 GiB)	10.42 GiB	58.06 GiB	5.57×
4M	2.32 min	(1.02 min)	3.69 min	(1.61 min)	1.59×	(1.58×)	12.83 GiB	(7.42 GiB)	13.96 GiB	84.07 GiB	6.02×
5M	2.91 min	(1.30 min)	4.71 min	(2.03 min)	1.62×	(1.56×)	16.04 GiB	(9.28 GiB)	17.30 GiB	103.19 GiB	5.96×
6M	3.49 min	(1.56 min)	—	—	—	—	19.25 GiB	(11.14 GiB)	20.69 GiB	—	—
7M	4.26 min	(1.89 min)	—	—	—	—	23.50 GiB	(13.62 GiB)	24.40 GiB	—	—
8M	4.89 min	(2.16 min)	—	—	—	—	26.86 GiB	(15.56 GiB)	27.83 GiB	—	—
9M	5.75 min	(2.52 min)	—	—	—	—	30.22 GiB	(17.51 GiB)	30.90 GiB	—	—
10M	6.61 min	(2.98 min)	—	—	—	—	33.58 GiB	(19.45 GiB)	34.29 GiB	—	—
20M	14.06 min	(6.01 min)	—	—	—	—	70.13 GiB	(40.70 GiB)	69.13 GiB	—	—
30M	22.53 min	(9.60 min)	—	—	—	—	109.67 GiB	(63.73 GiB)	104.85 GiB	—	—

Acknowledgments

This project received funding from the ERC under the EU's research and innovation programs Horizon Europe (PRIVTOOLS/101124778) and Horizon 2020 (PSOTI/850990). It was co-funded by the DFG within SFB 1119 CROSSING/236615297, and supported by BMFTR and HMWK within ATHENE.

References

AFO+21. Toshinori Araki, Jun Furukawa, Kazuma Ohara, Benny Pinkas, Hanan Rosemarin, and Hikaru Tsuchida. Secure Graph Analysis at Scale. In *CCS*, 2021.

BDST22. Lennart Braun, Daniel Demmler, Thomas Schneider, and Oleksandr Tkachenko. MOTION – A Framework for Mixed-Protocol Multi-Party Computation. *TOPS*, 25(2), 2022.

BKKS25. Andreas Brüggemann, Nishat Koti, Varsha Bhat Kukkala, and Thomas Schneider. MultiCent: Secure and Scalable Computation of Centrality Measures on Multilayer Graphs. *PoPETs*, 2025(4), 2025.

DHKS25. Henri Dohmen, Robin Hundt, Nora Khayata, and Thomas Schneider. SEEC: Memory Safety Meets Efficiency in Secure Two-Party Computation. In *ASIACCS*, 2025.

Kat53. Leo Katz. A New Status Index Derived from Sociometric Analysis. *Psychometrika*, 18(1), 1953.

KKK+25. Siddharth Kapoor, Nishat Koti, Varsha Bhat Kukkala, Arpita Patra, and Bhavish Raj Gopal. GraSP: Secure Collaborative Graph Processing Made Scalable. Cryptology ePrint Archive, Paper 2025/590, 2025.

KKPG24. Nishat Koti, Varsha Bhat Kukkala, Arpita Patra, and Bhavish Raj Gopal. Graphiti: Secure Graph Computation Made More Scalable. In *CCS*, 2024.

NWI+15. Kartik Nayak, Xiao Shaun Wang, Stratis Ioannidis, Udi Weinsberg, Nina Taft, and Elaine Shi. GraphSC: Parallel Secure Computation Made Easy. In *S&P*, 2015.

OR02. Evelien Otte and Ronald Rousseau. Social Network Analysis: A Powerful Strategy, also for the Information Sciences. *Journal of Information Science*, 28, 2002.

SvA+19. Alex Sangers, Maran van Heesch, Thomas Attema, Thijs Veugen, Mark Wiggerman, Jan Veldsink, Oscar Bloemen, and Daniël Worm. Secure Multiparty PageRank Algorithm for Collaborative Fraud Detection. In *FC*, 2019.

ZLS+24. Zhenhua Zou, Zhuotao Liu, Jinyong Shan, Qi Li, Ke Xu, and Mingwei Xu. CoGNN: Towards Secure and Efficient Collaborative Graph Learning. In *CCS*, 2024.
ZLS+25. Zhenhua Zou, Zhuotao Liu, Jinyong Shan, Qi Li, Ke Xu, and Mingwei Xu. RingSG: Optimal Secure Vertex-Centric Computation for Collaborative Graph Processing. In *CCS*, 2025.