Privacy-Preserving Breadth-First-Search and Maximal Flow

[Vincent Ehrmanntraut](https://orcid.org/0009-0005-0031-8988) RWTH Aachen University Aachen, Germany ehrmanntraut@itsec.rwth-aachen.de

Abstract

We present novel Secure Multi-Party Computation (SMPC) protocols to perform Breadth-First-Searches (BFSs) and determine maximal flows on dense secret-shared graphs. In particular, we introduce a novel BFS protocol that requires only $O(\log n)$ communication rounds on graphs with n nodes, which is a big step from prior work that requires $O(n \log n)$ rounds. This BFS protocol is then used in a maximal flow protocol based on the Edmonds-Karp algorithm, which requires $O(n^3 \log n)$ rounds. We further optimize the protocol for cases where an upper bound U on the capacities is publicly known by using a capacity scaling approach. This yields a new protocol which requires $O\left(n^2\log n\log U\right)$ rounds. Finally, we introduce a novel max flow protocol based on algorithms by Dinic and Tarjan with round complexity $O(n^3)$.

All protocols presented in this paper use SMPC primitives as a black-box, allowing our protocols to be used as building blocks in a wide range of settings and applications. We evaluate our protocols with semi-honest and malicious security in different network settings. Our logarithmic BFS protocol is up to 69 times faster than prior protocols on small graphs with less than 100 nodes, but is outperformed by protocols with lower computational complexity on graphs with thousands of nodes. Further, we find our Dinic-Tarjan protocol to be faster than the Edmonds-Karp and capacity scaling protocols in our evaluation, albeit trends indicating capacity scaling protocols to be faster on graph sizes not reached in our evaluation.

CCS Concepts

• Security and privacy \rightarrow Privacy-preserving protocols.

Keywords

secure multi-party computation; breadth first search; max flow

ACM Reference Format:

Vincent Ehrmanntraut and Ulrike Meyer. 2024. Privacy-Preserving Breadth-First-Search and Maximal Flow. In Proceedings of the 23rd Workshop on Privacy in the Electronic Society (WPES '24), October 14–18, 2024, Salt Lake City, UT, USA. ACM, New York, NY, USA, [25](#page-24-0) pages. [https://doi.org/10.1145/](https://doi.org/10.1145/3689943.3695041) [3689943.3695041](https://doi.org/10.1145/3689943.3695041)

1 Introduction

Secure Multi-Party Computation (SMPC) allows multiple parties to jointly compute functions with private inputs such that each party learns no more than it could have from just the private input and output. SMPC protocols have been used successfully for private

[Ulrike Meyer](https://orcid.org/0000-0002-2569-1042) RWTH Aachen University Aachen, Germany meyer@itsec.rwth-aachen.de

biometric identification [\[10\]](#page-12-0), privately finding bartering opportunities [\[41\]](#page-12-1), auctions with secret prices [\[12\]](#page-12-2), or even to prevent satellite collisions [\[25\]](#page-12-3). To further extend the frontier of SMPC applications, we investigate two well-known primitives of graph-theory, namely Breadth-First-Search (BFS) and maximal flow (also known as max flow). BFS is a widely used primitive to determine reachable nodes and shortest paths, which is used by many algorithms including the maximal flow algorithms used in this paper. Maximal flows are useful for transportation problems [\[23\]](#page-12-4), airline scheduling [\[18\]](#page-12-5), or solving further problems like bipartite matching [\[37\]](#page-12-6) or the closure problem [\[26\]](#page-12-7). Appendix [A](#page-12-8) presents an example application, where the maximal flow problem is used to compensate variations in the production quantities of factories.

We present novel data-oblivious SMPC protocols to perform BFSs and determine maximal flows of dense graphs. In particular, we present a BFS protocol which requires only $O(\log n)$ communication rounds on graphs with n nodes, a big theoretical improvement from previous protocols that require $O(n \log n)$ rounds [\[4\]](#page-12-9). This is important as the round complexity usually has a high impact on runtimes [\[30\]](#page-12-10). The logarithmic BFS protocol is used to improve upon prior maximal flow protocols based on the Edmonds-Karp algorithm [\[21\]](#page-12-11), yielding a maximal flow protocol that requires $O\left(n^3\log n\right)$ rounds. Assuming a publicly known upper bound U on the capacities, we further reduce the round complexity to $O(n^2 \log n \log U)$, using capacity scaling techniques, which were ignored by prior works. Similarly, Dinic's algorithm [\[20\]](#page-12-12) has also been ignored by prior works. We combine it with Tarjan's algorithm [\[38\]](#page-12-13), and introduce novel tricks that result in a protocol that is fast in practice even with an asymptotically worse round complexity of $O(n^3)$.

Due to the general nature of BFSs and maximal flows, we envision our protocols to be used as building blocks in larger, applicationdependent protocols. Therefore, our protocols accept graphs that are already secret-shared as input rather then specifying private inputs and how they are used to build the graph shared amoung the compute peers. As output, our protocols produce secret-shared BFS trees and secret-shared flows. To allow future protocols to build these secret-shared graphs from private data without revealing the number of edges, we focus on dense graphs. We further improve the applicability of our protocols by using SMPC primitives in a black box manner, and only assume that primitives are based on secret shares, which allows using our protocols in semi-honest as well as malicious security settings depending on the chosen SMPC primitive. The benefit of this flexibility is displayed in the example application in Appendix [A,](#page-12-8) which permits different setups with individual advantages and drawbacks.

We evaluate our protocols with semi-honest and malicious security in different network settings. We find our logarithmic BFS protocol to be very efficient on small graphs with less than 100

WPES '24, October 14–18, 2024, Salt Lake City, UT, USA

^{© 2024} Copyright held by the owner/author(s). Publication rights licensed to ACM. This is the author's version of the work. It is posted here for your personal use. Not for redistribution. The definitive Version of Record was published in Proceedings of the 23rd Workshop on Privacy in the Electronic Society (WPES '24), October 14–18, 2024, Salt Lake City, UT, USA, [https://doi.org/10.1145/3689943.3695041.](https://doi.org/10.1145/3689943.3695041)

nodes. On large graphs with thousands of nodes, the logarithmic BFS protocol is outperformed by other protocols with lower computational complexity. Nevertheless, the reduction from $O(n \log n)$ to $O(\log n)$ rounds is a noteworthy theoretical advancement on its own. Further, we show that our Dinic-Tarjan maximal flow protocol to be up to seven times faster than the improved Edmonds-Karp protocol on all evaluated graph sizes. We also show that our capacity scaling protocols are expected to eventually outperform the Dinic-Tarjan protocol on large graphs, but we do not reach such graph sizes in our evaluation.

2 Preliminaries

In this section, we briefly present relevant terminology of SMPC (Section [2.2\)](#page-1-0) protocols and graph theory (Section [2.3\)](#page-1-1), as well as the notation used in this paper (Section [2.1\)](#page-1-2) and the square-andmultiply algorithm (Section [2.4\)](#page-2-0).

2.1 Notation

Toft's notation [\[39\]](#page-12-14) is used as base for our protocol descriptions: $\lbrack x\rbrack$ denotes a secret sharing of the value $x.$ Publicly known values that are not secret-shared are written without brackets. The addition of two secret-shares is denoted as $[a] + [b]$ and the multiplication analogously as $[a] \cdot [b]$.

Further, the *i*'th element of a secret-shared vector $\left[\overrightarrow{V}\right]$ is denoted as $\left[\vec{v}_i\right]$. Analogously, the entry in the *i*'th row and *j*'th column of a secret-shared matrix $[\![M]\!]$ is denoted as $[\![M_{i,j}]\!].\,[\![M_{i,\bullet}]\!]$ is the i 'th row of the secret-shared matrix [M]. The transpose of [M] is written as $\lceil \textit{M}^{\mathsf{T}} \rceil$. The vector and matrix notation is analogous for publicly known vectors and matrices.

Finally, $\overrightarrow{E^{i,n}}$ is the *i*'th unit vector of length *n*, that is a vector of length n that only contains zeroes except a single 1 at position i .

2.2 Secure Multi-Party Computation

SMPC protocols allow multiple parties to compute a functionality on private inputs such that each party only learns the output of the functionality, and what can be deduced from the private input and the output. Such protocols can be built using different primitives such as [\[6,](#page-12-15) [19\]](#page-12-16). The protocols presented in this paper do not depend on specific SMPC primitives, but instead consider the used primitive to be a black box that operates on secret-shares over \mathbb{Z}_{2^λ} for sufficiently large λ^1 λ^1 .

We assume that the addition of secret-shares can be done efficiently using local computation only, and that the multiplication of secret-shares requires $O(1)$ rounds and $O(\lambda)$ communication. Further, we assume these operations are universally composable as defined in [\[13\]](#page-12-17), i.e., they can be securely performed in parallel.

These conditions are fulfilled by multiple SMPC primitives, for example [\[6,](#page-12-15) [19,](#page-12-16) [22\]](#page-12-18). These primitives offer different levels of security, which can be categorized into the semi-honest or the malicioussecurity adversary model. The former assumes that corrupted parties correctly follow the protocol, whereas the latter model allows corrupted parties to deviate arbitrarily. Further, the levels of security can be differentiated by the assumed number of corrupted

parties. Most commonly, primitives assume that only a minority of the parties is corrupted, which is called an honest majority setting, or that all parties but one are corrupted, which is called a dishonest majority setting. Due to the composition theorem [\[13\]](#page-12-17), the protocols presented in this paper "inherit" the level of security from the used primitives. In other words, our protocols achieve the same level of security as the used SMPC primitives^{[2](#page-1-4)}.

The complexity of protocols can be expressed using different metrics. We mainly focus on the (communication) round complexity, which is determined by the number of communication steps performed during a protocol execution. The round complexity usually has a big impact on the runtime of protocols since each round usually leads to idle time [\[30\]](#page-12-10). Other metrics are the communication complexity, which is determined by the total amount of data sent, and computational complexity, which is determined by the total number of computation steps the protocol requires.

The protocols presented in this paper use several existing gates for elementary operations, such as comparisons of secret-shared values. As shown by [\[33\]](#page-12-19), such comparisons can be realized in constant ($O(1)$) communication rounds and $O(\lambda)$ communication. A closely related gate is the bit decomposition, that returns secretshares of the individual bits of a secret-shared value. It too can be realized in $O(1)$ rounds and $O(\lambda)$ communication ^{[3](#page-1-5)}.

Further, we use a gate G_{MIN} to determine the smallest value in a secret-shared vector. Anagreh et al. [\[4\]](#page-12-9) introduced a gate with $O(\log m)$ rounds and $O(m)$ communication, where m is the size of the vector.

Finally, we occasionally need to select one of two values based on a secret-shared condition. In that case, we use $[c]$? $[a]$: $[b]$ as shorthand notation for $[b] + [c] \cdot ([a] - [b])$. This results in $[a]$ if $c = 1$ and $[b]$ if $c = 0$.

2.3 Graph theory

A graph *G* consists of a set of nodes *V* and a set of edges $E \subseteq V \times V$. Further, $n = |V|$ denotes the number of nodes in the graph. We assume (without loss of generality) that $V = \{1, \ldots, n\}$. The graph can also be expressed as adjacency matrix $A \in \{0,1\}^{n \times n}$ where

 $A_{i,j} =$ $\begin{cases} 1 & \text{if } (i, j) \in E \end{cases}$ $\frac{1}{1}$ $\frac{1}{1}$, $\frac{1}{1}$, $\frac{1}{2}$ $\frac{1}{2}$. When using the adjacency matrix, it might otherwise

be useful to encode nodes as vectors to express some operations as vector-matrix-multiplications. In that case, a node $v \in V$ is encoded $\overline{E^{v,n}}$, that is a vector that only contains zeroes except as singe one at position $v \in V = \{1, \ldots, n\}.$

A path is a tuple (v_1, \ldots, v_l) such that $(v_i, v_{i+1}) \in E$ for all *i* from 1 to $l - 1$. A path is called a shortest path when there is no path in G that connects v_1 and v_l while visiting fewer nodes. Furthermore, the subgraph G' of G, i.e., $G' = (V, E')$ with $E' \subseteq E$, that only contains all shortest paths starting at a node v is called the layered subgraph of G starting on v .

Further, edge capacities can be defined for a graph as a function $c: V \times V \rightarrow \mathbb{N}$. All edges have a non-negative capacity, i.e.,

¹We set $\lambda = 64$ in our evaluation.

 2 There are SMPC primitives for both security setting and all number of parties $\geq 2.$ ³We use the stated complexities for the theoretical analysis. In our evaluation, we use different gates from [\[33\]](#page-12-19) with $O(\log \lambda)$ rounds and $O(\lambda)$ communication, as they are faster in practice for $\lambda = 64$.

 $c(i, j) \geq 0$ for all $(i, j) \in E$, while $c(i, j) = 0$ for all $(i, j) \notin E$. Similarly to the adjacency matrix, the capacities can also be expressed as matrix $C \in \mathbb{N}^{n \times n}$ with $C_{i,j} = c(i,j)$.

A flow sends "flow units" from a source node $s \in V$ to a sink node $t \in V$ and is defined as functions $f: V \times V \rightarrow \mathbb{Z}$ that satisfy the following constraints: First, the capacities have to be respected, i.e., $f(i, j) \leq c(i, j)$ for all $i, j \in V$. Further, the skew symmetry has to hold, i.e., $f(i, j) = -f(j, i)$ for all $i, j \in V$. Finally, flow conservation is required for all nodes except the source and the sink, i.e., for all nodes except source and sink, the inflow equals the outflow. Formally, $\sum_{j \in V} f(i, j) = 0$ for $i \in V \setminus \{s, t\}$ [\[11\]](#page-12-20). Flows can also be represented using matrices $Flow \in \mathbb{Z}^{n \times n}$ with $Flow_{i,j} = f(i,j)$.

2.4 Square-and-multiply algorithm

The square-and-multiply algorithm [\[32\]](#page-12-21) calculates of b^e in O (log e) steps. It first computes $b^{(2^0)}, \ldots, b^{\left(2^{\lfloor \log e \rfloor}\right)}$ by repeated squaring, and assembles these values into b^e . To do so, let $J \subseteq \{0, \ldots, \lfloor \log e \rfloor\}$ be such that $e = \sum_{i \in J} 2^i$. Then $b^e = b^{\left(\sum_{i \in J} 2^i\right)} = \prod_{i \in J} b^{\left(2^i\right)}$. Note that the algorithm also is applicable to compute B^e when B is a square matrix.

3 Related work

In this section we provide an overview of algorithms for finding maximal flows (Section [3.1\)](#page-2-1) and discuss which approaches introduce which challenges when turning an algorithm into a SMPC protocol. Further, we present privacy-preserving protocols that perform breadth-first-searches and find maximal flows on private graphs (Section [3.2\)](#page-2-2).

3.1 Non-privacy-preserving maximal flow algorithms

The maximal flow problem is a classical and well studied graph theoretic problem, and many different approaches to solve it have been developed. One branch of research focussed on iteratively increasing intermediary flows using various techniques such as augmenting paths [\[21,](#page-12-11) [23\]](#page-12-4), blocking flows [\[20\]](#page-12-12), and scaling [\[24\]](#page-12-22). These algorithms inspire the protocols in this paper.

A different branch of combinatorial maximal flow algorithms are push-relabel algorithms such as [\[1\]](#page-12-23). Such algorithms require dynamic data structures to guarantee good runtimes, making SMPC adaptions challenging.

A breakthrough paper by Christiano et al. [\[16\]](#page-12-24) lead to significant developments in the past decade. The paper applies Interior Point Methods (IPMs) and efficient solvers for Laplacian equation systems to achieve an approximation algorithm with complexity $O\left(mn^{1/3}\epsilon^{-11/3}\right)$, where ϵ is an approximation factor. Kathuria [\[27\]](#page-12-25) presented the first exact algorithm based on this approach, which has complexity $O\left(m^{4/3+o\left(1\right)}U^{1/3}\right)$, where U is the biggest capacity. Further development culminated in an algorithm with complexity $O\left(m^{1+o\left(1\right)}\right)$, which was introduced by Chen et al. [\[15\]](#page-12-26). This is the current (asymptotically) fastest maximal flow algorithm.

Algorithms based on IPMs heavily rely on non-linearities like inverses and logarithms, while simultaneously requiring high numerical precision and stability. SMPC implementations of such

non-linearities are expensive compared to the non-privacy preserving implementations. Further, algorithms like [\[15\]](#page-12-26) require dynamic data-structures. As SMPC protocol must hide memory access patterns to avoid leakages, such dynamic data-structures are hard to realize efficiently in SMPC protocols. For these reasons, we focus on more SMPC-friendly algorithms in this paper, and leave privacypreserving implementations of IPM-based graph algorithms open for future work.

3.2 Protocols on private graphs

Privacy-preserving protocols either focussed on dense graphs, where the number of edges *m* is close to n^2 , or on sparse graphs with fewer edges. We discuss these separately, starting with the former:

Dense graphs: To the best of our knowledge, the only dataoblivious BFS protocol on dense graphs was introduced by Blanton et al. [\[11\]](#page-12-20). However, BFS can be interpreted as a special case of the Dijkstra algorithm on graphs with uniform edge lengths. Therefore, the Dijkstra-based protocols such as [\[2–](#page-12-27)[4\]](#page-12-9) can also be used to perform Breadth-First-Searches. Most of these protocols [\[2,](#page-12-27) [4,](#page-12-9) [11\]](#page-12-20) first permute the nodes and then iteratively visit each node. The permutation allows these protocols to securely reveal the node that is being visited, yielding protocols requiring $O(n^2)$ rounds in the case of [\[2,](#page-12-27) [11\]](#page-12-20), and $O(n \log n)$ in [\[4\]](#page-12-9). One protocol [\[3\]](#page-12-28) uses a construction similar to bubble-sort instead of secret permutations, which requires $O(n^3)$ rounds. Compared to those protocols, the BFS protocol presented in this paper requires only $O(\log n)$ communication rounds. Finally, Anagreh et al. [\[5\]](#page-12-29) also presented a BFS protocol that is not data-oblivious, as it leaks the number of iterations. Thus, a fair comparisons between [\[5\]](#page-12-29) and data-oblivious protocols is impossible.

To the best of our knowledge, Blanton et al. [\[11\]](#page-12-20) also introduced the only SMPC protocol to find a maximal flow on dense graphs. It is based on the Edmonds-Karp algorithm [\[21\]](#page-12-11) and requires $O(n^5)$ communication rounds. We modify their protocol to be compatible with our BFS, thus achieving a protocol with round complexity $O(n^3 \log n)$. Further, we present two completely new protocols with round complexities $O(n^3)$ respectively $O(n^2 \log n \log U)$, where U is an upper bound on the capacities.

Sparse graphs: Privacy-preserving protocols that operate on sparse graphs are either based on message-push-algorithms or on Oblivious Random Access Memories (ORAMs). Message-pushalgorithms are a special class of algorithms, where each node repeatedly sends a message to its neighbors and updates its own state based on received messages. Such algorithms can be adapted to SMPC protocols, which introduces an overhead of $O(\log(n+m))$ rounds for each step, where m is the number of edges [\[7,](#page-12-30) [34\]](#page-12-31). Araki et al. [\[7\]](#page-12-30) use this approach for contract tracing, which includes a limited-depth BFS with $O(d \log(n + m))$ rounds for depth d. We are not aware of any maximal flow protocol using this approach.

Another approach to privacy-preserving computations on sparse graphs is based on ORAMs, which allow arbitrary read and write accesses while hiding access patterns. This has been used to find shortest paths [\[4,](#page-12-9) [29,](#page-12-32) [35\]](#page-12-33). Hence, these protocols can be used to emulate BFS. The round complexity of these protocols mainly depends on the ORAM construction, the lowest round complexity is $O(n \log n \log \log \log n)$ [\[35\]](#page-12-33). This approach was also used in trusted WPES '24, October 14–18, 2024, Salt Lake City, UT, USA Vincent Ehrmanntraut and Ulrike Meyer

Figure 1: The main steps of the logarithmic BFS protocol presented on an example graph, starting on node 1.

execution environments [\[14\]](#page-12-34), but with higher reported runtimes than our BFS protocol^{[4](#page-3-0)}.

The ORAM approach was used by a SMPC protocol based on the Edmonds-Karp algorithm [\[40\]](#page-12-35), which only works on planar graphs.

All protocols on sparse graphs have in common that they not only require the number of nodes to be public, but also the number of edges. However, in use cases where the graph is built from private data, leaking the number of edges might not be acceptable. In contrast, protocols on dense graphs, like the protocols presented in this paper, do not have this issue since the number of edges remains private.

Appendix [B](#page-14-0) provides an overview of all complexity measures of the mentioned, and our new, protocols.

4 Breadth-First-Search with logarithmic round complexity

In this section, we present our novel protocol ΠBFS-LOG that computes the BFS tree with a logarithmic round complexity.

4.1 Breadth-First-Search Trees

BFS is an algorithm to determine all nodes that are reachable from a starting node. It iteratively picks a (previously unvisited) node with the least distance to the starting node, which is often realized using a queue, and marks all neighboring nodes as seen. BFS can also be used to find the shortest paths from the starting node to the other nodes by keeping track of the first node that "sees" a previously unseen node. This results in tree, which only contains shortest paths from the starting node to all other (reachable) nodes. Such trees are called BFS trees [\[8\]](#page-12-36). An example of a BFS tree can be found in Figure [1e](#page-3-1).

4.2 The protocol

The logarithmic BFS protocol follows a fundamentally different approach compared to the classical BFS algorithm. It does not visit each node individually, which would result in a protocol with $O(n)$ rounds like the one described in Appendix [E.1.](#page-16-0)

Instead, the protocol operates as demonstrated in Figure [1:](#page-3-1) First, the nodes that are reachable in 0 to $n - 1$ steps from the start node are determined. By using the square-and-multiply algorithm, this is done in $O(\log n)$ rounds. Next, the nodes are filtered such that only the layers of the BFS tree remain, which is done in $O(1)$ communication rounds. This then allows the protocol to determine the layered subgraph in $O(1)$ communication rounds. The layered subgraph is already very similar to the BFS tree, with the exception that a node might have multiple incoming edges. In the final step of the protocol, the edges in the layered subgraph are filtered such that only the BFS tree remains, again in $O(1)$ communication rounds.

In greater detail, the first step, i.e., determining the nodes that are reached in 0 to $n - 1$ steps, is done by calculating exponents of the adjacency matrix. This is based on the intuitive understanding that $\overline{E^{v,n}}$. A yields a vector indicating all nodes that are reachable from v in one step^{[5](#page-3-2)}. It follows that $(E^{v,n} \cdot A) \cdot A = \overline{E^{v,n}} \cdot A^2$ yields a vector indicating the nodes that are reachable from v in exactly two steps. In general, the nodes reached from v in exactly i steps are indicated by $\overline{E^{0,n}}$. A^i , which is proven formally in Appendix [C.1.](#page-14-1)

Protocol [1](#page-4-0) assumes that the starting node of the search is encoded as vector *Start*. In Lines [1](#page-4-0) to [8,](#page-4-0) it uses the square-and-multiply method [\[17\]](#page-12-37) to calculate $\overrightarrow{Start} \cdot A^0, \dots, \overrightarrow{Start} \cdot A^{n-1}$ in parallel, which requires $O(\log n)$ communication rounds. An implementation detail is that the intermediary values of the computation are no longer restricted to {0, 1} but might be larger natural numbers. For this reason, Lines [9](#page-4-0) to [10](#page-4-0) of Protocol [1](#page-4-0) "reset" all non-zero values to 1, which simplifies further computations. Another consequence is that integer overflows might occur during the square-and-multiply

 $^{4}[14]$ $^{4}[14]$ also evaluated on dense graphs. The largest evaluated dense graph has roughly 500 nodes (in [\[14\]](#page-12-34) graph size is $|V| + |E|$). On this graph size, our logarithmic BFS protocol is 142 times faster in the slowest evaluated network setting.

^{5"}Indicating" means that a non-zero entry indicates that the corresponding node is reachable, and zero indicates that the node is not reachable.

procedure. This can cause the protocol to ignore edges and nodes, affecting the correctness. However, overflows can be prevented by keeping track of an upper bound on intermediary values and resetting non-zero values to 1 before an overflow can occur. Appendix [D](#page-15-0) discusses the effect of overflows and their mitigation in-depth.

Protocol 1 $\Pi_{\text{BFS-LOG}}([A], \overrightarrow{\text{Start}}])$ Find the nodes reachable in $0, \ldots, n-1$ steps: 1: $\overline{[B] \leftarrow [A]}$ 2: for $1 \le i \le n$ in parallel do $3:$ $AfterSteps_{i, \bullet} \rightarrow \sqrt{Start}$ 4: for $0 \leq i \leq \lceil \log n \rceil$ do 5: for $1 \le j \le n$ in parallel do 6: **if** $((j - 1) \gg i) \& 1 = 1$ then 7: $\begin{bmatrix} AfterSteps_{j,\bullet} \end{bmatrix} \leftarrow \begin{bmatrix} AfterSteps_{j,\bullet} \end{bmatrix} \cdot [B]$ 8: $[B] \leftarrow [B]^2$ 9: for $1 \le i, j \le n$ in parallel do 10: $[AfterSteps_{i,j}] \leftarrow [AfterSteps_{i,j}]$ Filter to layers: 11: **for** $1 \leq i, j \leq n$ do $12:$ $\text{PrefixSum}_{i,j} \left[\left. \left. \left. \left. \left. \right\rangle \right. \right. \right. \right) \left. \left. \left(\left. \left. \right\rangle \right. \right) \right] \left. \left(\left. \left. \right\rangle \right. \right) \right]$ 13: for $1 \le i, j \le n$ in parallel do $14:$ $Cmp_{i,j}$ $\left| \leftarrow \right|$ $PrefixSum_{i,j}$ > 0 15: $\left[\textit{Layers}_{1,\bullet} \right] \leftarrow \left[\textit{Cmp}_{1,\bullet} \right]$ 16: for $2 \le i \le n, 1 \le j \le n$ do 17: $\left[\textit{Layers}_{i,j} \right] \leftarrow \left[\textit{Cmp}_{i,j} \right] - \left[\textit{Cmp}_{i-1,j} \right]$ Build the layered subgraph: 18: for $1 \le i, j \le n$ in parallel do 19: for $1 \leq k \leq n-1$ in parallel do 20: $\left[\overrightarrow{Tmp_k}\right]$ ← $\left[Layer_{k,i}\right] \cdot \left[Layer_{k+1,j}\right]$ $21:$ Layered $A_{i,j}$] \leftarrow $[A_{i,j}] \cdot \sum_{k=1}^{i-1} \left\lceil \overrightarrow{Tmp_k} \right\rceil$ Filter to BFS tree: 22: for $1 \leq i, j \leq n$ do 23: $[PrefixSum_{i,j}] \leftarrow \sum_{k=1}^{i} [LayerA_{k,j}]$ 24: for $1 \le i, j \le n$ in parallel do $25:$ $Cmp_{i,j}$ $\left| \leftarrow \right|$ $PrefixSum_{i,j}$ > 0 26: $\left[\operatorname{Tree}_{1,\bullet} \right] \leftarrow \left[\operatorname{Cmp}_{1,\bullet} \right]$ 27: for $2 \le i \le n, 1 \le j \le n$ do 28: $[\mathit{Tree}_{i,j}] \leftarrow [\mathit{Cmp}_{i,j}] - [\mathit{Cmp}_{i-1,j}]$ 29: return [Tree]

Once the nodes that can be reached from the starting node are calculated, the layers of the BFS tree can be determined. This is done by filtering the $AfterSteps$ -matrix, such that a node only remains on the step where it was seen first. For example, in Figure [1](#page-3-1) node 3 can be reached by one and two steps. After the filtering, node 3 is kept only on the second layer, which corresponds to the nodes that were reached after one step. When looking at the corresponding encoded sets in Figure [1b](#page-3-1) and c, it can be seen that this filtering step is equivalent to finding the first 1 in each column.

Lines [11](#page-4-0) to [17](#page-4-0) of Protocol [1](#page-4-0) perform this filter step by first calculating the prefix sum for each entry, i.e., each element is the sum of the preceding elements in the column. This prefix sum is zero for positions prior to the first 1 of each column, and larger than zero for all following elements. Therefore, the first 1 can be found by identifying the element that is larger than zero, but where the previous prefix-sum is zero. After comparing all prefix-sums with zero in parallel, this can be done by taking the difference between each comparison result and the prior comparison result. Since calculating the prefix sums and the differences do not require any communication, and the comparisons are performed in parallel, the filter step is performed in $O(1)$ communication rounds.

After the nodes have been partitioned into layers, the layered subgraph is determined. This is done by checking whether an edge is part of the layered subgraph for each edge in parallel. For that to be the case, an edge (i, j) has to fulfill two conditions: It has to be part of the input graph, i.e., $A_{i,j} = 1$. And it has to transition from one layer to the next, i.e., there has to be a k such that i is in layer k and j is in layer $k + 1$. The latter check is performed in parallel for each possible k , and the result of the checks is multiplied with $A_{i,j}$ in Line [21](#page-4-0) of Protocol [1.](#page-4-0) It follows that building the subgraph is done in $O(1)$ communication rounds.

As shown in Figure [1d](#page-3-1), the layered subgraph is similar to the BFS tree, but nodes might still have multiple incoming edges. For such nodes, the corresponding column in the adjacency matrix contains multiple ones. To reduce the layered subgraph to a BFS tree, the columns of the adjacency matrix must be filtered such that only one 1 remains, which can be done with same column-wise filtering used to determine the layers of the graph. Thus, the final step needed to produce the BFS tree also is performed in $O(1)$ rounds.

We want to note that filtering the layered subgraph indeed results in a correct BFS tree. Recall that the BFS tree only contains shortest paths from the start node to all other (reachable) nodes. Since the protocol builds the BFS tree by removing edges from the layered subgraph, which contains all shortest paths starting at the start node, this property is given. Furthermore, the final filtering step ensures that the result of the protocol is indeed a tree while ensuring that all reachable nodes remain reachable. Hence, the result of the protocol is a valid secret-shared BFS tree.

Finally, since the protocol only calls secure gates, never reveals the values of secret-shares, and follows a publicly known control flow that only depends on the number of nodes, it can be simulated by applying the composition theorem [\[13\]](#page-12-17).

5 Maximal flows

Recall from Section [2.3](#page-1-1) that a flow sends "flow units" from a source node s to a sink node t in a graph with capacity constraints. A well-studied problem in graph-theory is finding a flow that sends the maximal number of units from the source to the sink, which is also called the maximal flow problem [\[1,](#page-12-23) [15,](#page-12-26) [16,](#page-12-24) [20,](#page-12-12) [21,](#page-12-11) [23,](#page-12-4) [24,](#page-12-22) [27\]](#page-12-25). We first non-trivially adapt a previous maximal flow protocol to our logarithmic BFS protocol in Section [5.1,](#page-5-0) yielding a maximal flow protocol with round complexity $O(n^3 \log n)$. We further identify capacity scaling techniques as a gap in prior works, which

allows reducing the round complexity to $O\left(n^2\log n\log U\right)$, assuming an upper bound U on the capacities. Finally, we present a completely novel protocol inspired by the algorithms of Dinic [\[20\]](#page-12-12) and Tarjan [\[38\]](#page-12-13). This protocol employs novel tricks to achieve $O\left(n^3\right)$ rounds. Section [6](#page-8-0) shows that it is faster than the Edmonds-Karp and capacity scaling protocols on evaluated graph sizes.

5.1 Edmonds-Karp

The Edmonds-Karp algorithm iteratively increases the flow sent from the source to the sink by finding a path from the source to sink that still has some capacity left, and sending as much flow as possible along that path [\[21\]](#page-12-11). In greater detail, a residual graph is built that only contains edges that have capacity left, i.e., all edges (i, j) with $f(i, j) < c(i, j)$. BFS is then used to find a shortest path from the source to sink in the residual graph. This path is also called an augmenting path. Once a path has been found, the flow is augmented by sending as much flow as possible along the path. If no path can be found, the flow is maximal [\[23\]](#page-12-4). It has been shown that by using BFS to select augmenting paths, the algorithm requires at most n^3 iterations [\[21\]](#page-12-11).

Blanton et al. [\[11\]](#page-12-20) already implemented the Edmonds-Karp as a SMPC protocol. However, their implementation uses a BFS protocol with a round complexity of $O(n^2)$. We will use the protocol presented in Section [4](#page-3-3) instead. However, just using our BFS protocol is not sufficient to optimize prior protocols. In order to adequatly profic from our BFS protocol, the extraction of the path from the source to the sink and finding the capacity of the augmenting path must be done in $O(\log n)$ rounds as well. Prior protocols require $O(n)$ rounds for both steps [\[11\]](#page-12-20).

We improve the extraction of the augmenting path from the BFS tree by introducing two phases in Protocol [2.](#page-5-1) First, the protocol determines the nodes on the path by iteratively stepping from the sink node to its parent in the BFS tree, then to its parent, and so on. By using the square-and-multiply algorithm in Lines [8](#page-5-1) to [15,](#page-5-1) this is done in $O(\log n)$ communication rounds. Next, the corresponding edges of the path are determined, which is done in $O(1)$ communication rounds by checking whether an edge transitions from one visited node to its parent for each edge in parallel (Lines [16](#page-5-1) to [19\)](#page-5-1).

The implementation of the path extraction in Lines [8](#page-5-1) to [19](#page-5-1) of Protocol [2](#page-5-1) is very similar to the first two steps of the logarithmic BFS protocol. However, the extraction is slightly optimized due to a special property of the transposed BFS tree matrix, which contains only one 1 in each row. This is also true for the $AfterSteps$ matrix. As shown in Appendix [C.2,](#page-14-2) this property will propagate to new intermediary values. It follows that all intermediary states remain in {0, 1}. Therefore, no overflow mitigations and no comparisons are required during the square-and-multiply procedure.

Once a shortest augmenting path has been found, the flow needs to be augmented accordingly, which is done in Lines [20](#page-5-1) to [23.](#page-5-1) First, the minimal residual capacity along the path is found. To do so, a vector is built that for each edge either contains the residual capacity of the edge if it is part of the path, or contains the residual capacity of the edge plus a large constant ∞ if the edge is not part of the path. This does not require any communication. The residual capacity of the augmenting path is then determined by calling the \mathcal{G}_{MIN} gate on the vector, which requires $O(\log n)$ communication rounds. Finally,

the flow is updated in Line [23.](#page-5-1) Note that the transpose of the path is subtracted to satisfy the skew symmetry constraints.

Hence, each iteration requires only $O(\log n)$ communication rounds. To achieve data-obliviousness, Protocol [2](#page-5-1) always performs n^3 iterations, resulting in O $(n^3 \log n)$ overall rounds.

It should be noted that always executing n^3 iterations, instead of terminating as soon as no augmenting path exists, does not affect the correctness of the protocol. If the protocol finds a maximal flow early, it will not perform any further changes to the flow, because $Tree_{Sink, \bullet}$ will only contain zeroes when no augmenting path exists. Therefore, *AfterSteps*_{i.} will also only contain zeroes after the square-and-multiply procedure for all $2 \le i \le n$. It follows that the Path matrix will only contain zeroes, and no changes are made to the flow. Thus, the correctness of the protocol follows from the correctness of the Edmonds-Karp algorithm.

Finally, since the protocol does not reveal the value of any secretshares, and only calls secure gates, and the control flow of the protocol only depends on the number of nodes, Protocol [2](#page-5-1) can be simulated by applying the composition theorem [\[13\]](#page-12-17).

5.2 Capacity scaling

Other than reducing the round complexity of each Edmonds-Karp iteration, we also reduce the number of iterations by identifying a gap in prior works, namely capacity scaling: Capacity scaling algorithms solve the maximal flow problem by reducing it to a series of "smaller" maximal flow problems [\[24,](#page-12-22) [36\]](#page-12-38). For example, a small problem for the (non-privacy-preserving) Edmonds-Karp algorithm is a graph where the maximal flow value F is small $(< n³)$. In that case, the algorithm terminates after at most F iterations, since each iteration adds at least one flow unit.

In particular, imagine a graph whose capacities are constrained in $\{0, 1\}$, i.e., all capacities are just one bit long. Then $F \leq n^2$ and the Edmonds-Karp algorithm terminates after at most n^2 iterations.

This is exploited by Protocol [3,](#page-6-0) which iteratively "adds" one bit to the intermediary capacity matrix C' , starting from the most significant bit. In each iteration, the flow created in the previous iteration is "updated" using n^2 iterations of the $\Pi_{\text{MAX-FLOW-EDMONDS-KARP}}$ protocol. It follows that Protocol [3](#page-6-0) requires $O\left(n^2\log n\log U\right)$ communication rounds, where U is a publicly known upper bound on the edge capacities.

The protocol is inspired by an early work of Garbow [\[24\]](#page-12-22), who used Dinic's algorithm [\[20\]](#page-12-12) to solve each sub-problem, as it is faster than Edmonds-Karp algorithm in the non-privacy-preserving setting. In the privacy-preserving setting, Dinic's algorithm would require $O\left(n^3\right)$ communication rounds for each subproblem, as shown in Section [5.3.](#page-6-1) Therefore, we use our Edmonds-Karp protocol, which requires only $O(n^2 \log n)$ communication rounds per subproblem. Note that this does not affect the correctness of the protocol, as Garbow showed that updating the flow requires at most n^2 augmenting paths [\[24\]](#page-12-22), i.e., at most n^2 Edmonds-Karp iterations.

Protocol [3](#page-6-0) only uses secure gates and its control flow is publicly known. Hence, it can be simulated by applying the composition theorem [\[13\]](#page-12-17). A slight caveat of this protocol is that an upper bound on capacities must be publicly known. However, this does not really constitute a leakage, as 2^{λ} (the field size) can be used as upper bound.

5.3 Dinic's maximal flow algorithm

Finally, we present a protocol based on Dinic's algorithm [\[20\]](#page-12-12) which solves the maximal flow problem with round complexity $O(n^3)$. This protocol has a worse round complexity than the capacity scaling protocol. However, in Section [6](#page-8-0) we show experimentally that it is faster than the capacity scaling protocol on small graphs.

Similarly to Edmonds-Karp algorithm, Dinic's algorithm iteratively increases the flow sent from the source to the sink. But instead of augmenting the flow along a singular augmenting path at a time, Dinic's algorithm augments the flow along all shortest augmenting paths simultaneously. This ensures that the shortest augmenting paths in the next iteration (also called phase) will be longer. Therefore, this algorithm terminates after at most n phases, where each phase augments the flow along all shortest paths.

Dinic presented an efficient algorithm for finding all shortest augmenting paths. However, implementing this algorithm as SMPC protocol would introduce a significant overhead due to the dynamic manner in which nodes are visited. Therefore, we use a different algorithm introduced by Tarjan [\[38\]](#page-12-13) to augment the flow along all shortest paths.

This algorithm resembles a wave that bounces between the source and the sink of the graph. First, all outgoing edges of the source are saturated, i.e., the source sends as much flow as possible to its neighboring nodes. These neighbors then have excess flow, which they try to send to their neighbors in the next layer of the layered subgraph of the residual graph. This propagation of flow is repeated until the sink node is reached. All nodes that were not able to propagate all incoming flow, i.e., all nodes that have some remaining excess flow, are then marked as blocked. Starting from the bottommost layer, those blocked nodes then send the excess flow backwards towards the direction of the source, which increases the excess flow at previous nodes. In the next iteration of the algorithm, the nodes with new excess capacity again attempt to propagate it towards the sink and become blocked if they fail to do so. Blocked nodes then again return their excess flow towards the direction of the source.

This "bouncing" of the wave from source to sink and back is repeated n times. Then, no node will have an excess capacity and the resulting flow is blocking in the layered subgraph of the residual graph [\[38\]](#page-12-13). In other words, all shortest paths from the source to sink have at least one edge without any remaining capacity, which is equivalent to finding all shortest augmenting paths.

Adapting this algorithm to a SMPC protocol poses some challenges: Propagating flows would require $O(n)$ rounds per visited node, resulting in $O(n^4)$ rounds overall. This is because Tarjan's algorithm iteratively looks at the edges of the visited node. It first sends as much excess as possible along the first edge, then as much remaining excess along the second edge, and so on. As a consequence, the flow sent along an edge non-linearly depends on the flows sent along the previous edges. However, we notice that in each iteration, the algorithm wants to send at most the initial excess minus the capacities of the previous edges. This wanted outflow may exceed the capacity of the current edge if the excess is too large, and may be negative if the excess has already been distributed along the previous edges. By clamping the wanted outflow to the

range between 0 and the edge's capacity, the wanted outflow is converted into the actual outflow along the edge.

This new perspective on Tarjan's algorithm allows a significant optimization of our SMPC protocol: After determining the capacities of the edges (Lines 2 and 3 of Gate [1](#page-7-0) and [2\)](#page-8-1), the wanted outflows can be computed using local computation only, i.e., without any communication (Lines 4 and 5). The wanted outflows can then be clamped in parallel (Lines 6–10). Hence, our new sum-and-clamp trick allows Gate [1](#page-7-0) and [2](#page-8-1) to propagate the excess of a node towards the sink, respectively towards the source, in $O(1)$ rounds.

Another challenge of adapting Tarjan's algorithm to a SMPC protocol is that the algorithm either requires stacks or a topological order of the nodes to be efficient. Both, simulating the stack, and determining the topological order would introduce a significant overhead, even with our BFS protocol. Instead, we notice that all nodes of a layer can be visited in parallel. Since each node only propagates flows to the next / previous layer, and not to nodes in the same layer, this does not affect the correctness. As Gate [1](#page-7-0) and [2](#page-8-1) need to hide the visited node anyways, they can visit a whole layer of nodes in parallel without any further overhead. Further, the layers of the graph can be determined efficiently by partially executing our BFS protocol, which also yields the layered subgraph of the residual graph. This results in Protocol [4,](#page-7-1) which calls Gate [1](#page-7-0) and [2](#page-8-1) O (n^3) times in total and hence requires O (n^3) rounds.

The correctness of our protocol mainly follows from the correctness of [\[20\]](#page-12-12) and [\[38\]](#page-12-13). However, Protocol [4](#page-7-1) might perform more iterations than these algorithms. This does not affect the correctness as the surplus iterations of the blocking flow algorithm have no effect since all excesses are zero. The correctness of Dinic's algorithm is also not affected by surplus iterations as in those iterations there is no augmenting path from the source to the sink, and hence the blocking flow is zero.

The security of Gate [1](#page-7-0) and [2](#page-8-1) as well as Protocol [4](#page-7-1) follows from the composition theorem [\[13\]](#page-12-17) as they all only use secure gates and a publicly known control flow that only depends on the number of nodes.

6 Evaluation

In this section, we present and discuss the runtimes of the protocols presented in this paper.

6.1 Setup

We evaluate the protocols presented in this paper using semi-honest and malicious-security SMPC primitives. Concretely, the primitives replicated-ring [\[6\]](#page-12-15) and ps-rep-ring [\[22\]](#page-12-18) as implemented by the MP-SPDZ [\[28\]](#page-12-39) framework version 0.3.8 are used. As these primitives only support three parties, we evaluate with three parties.

An important detail of these primitives is that they require preprocessed data, like random bits or Beaver triplets. The primitive specifications assume that all required pre-processed data is generated at once in a so-called offline phase, and then later used in a so-called online phase. MP-SPDZ, however, departs from that approach and generates batches of pre-processed data on-the-fly when they are needed. By default, the batch size is 10 000, e.g., when a random bit is needed, ten thousand random bits are generated^{[6](#page-8-2)}. However, our protocols might require more than ten thousand bits at once due to their high parallelism. Therefore, we increase the batch size to one million in our evaluation.

We use NEON [\[31\]](#page-12-40) to simulate three different network settings: A LAN setting with one millisecond delay and one gigabit per second bandwidth, a WAN setting with ten milliseconds delay and 100 megabits per second bandwidth, and an unrestricted setting without artificial delay or bandwidth restrictions.

Figure 2: Runtimes of the semi-honest BFS protocols on small graphs by number of nodes in the WAN setting.

This setup runs on different LXC containers for the BFS protocols and for the maximal flow protocols, with the BFS container having access to 128GB $RAM⁷$ $RAM⁷$ $RAM⁷$ and the maximal flow container having access to 16GB RAM. All containers had access to four cores of an AMD 7702P CPU clocked at 2GHz. We execute each protocol five times in each setting in a random order.

Protocol implementations and runtimes of all individual computations are availableat<https://doi.org/10.5281/zenodo.13619649> .

6.2 Breadth-First-Search

We not only evaluate the logarithmic BFS protocol presented in Section [4,](#page-3-3) but also two protocols presented in Appendix [E.](#page-16-1) One of these protocols, called the linear BFS protocol, has a round complexity of $O(n)$, but does not require any overflow mitigation. The other protocol, called the comparison-free protocol, does not require any comparisons, which can be expensive in the malicious-security case, but requires $O(n \log n)$ rounds. We also include the BFS protocol introduced by Blanton et al. [\[11\]](#page-12-20) in our evaluation as baseline. As this protocol was not optimized for secret-sharing based SMPC primitives, we slightly optimize the protocol by using the gate presented in [\[4\]](#page-12-9) to find the minimum of a secret-shared array. This reduces the round complexity of [\[11\]](#page-12-20) to $O(n \log n)^8$ $O(n \log n)^8$. All complexity measures of the evaluated protocols are presented in Appendix [B.](#page-14-0)

The protocols were evaluated on graphs of up to 2600 nodes, with 200 node increments, in the semi-honest setting^{[9](#page-8-5)}. In the malicious security setting, the protocols were evaluated only up to 400 nodes, with 100 node increments, since the total RAM required for all parties exceeded 128GB RAM on larger graphs. For protocols whose runtime exceeds one hour, the evaluation was aborted such that the first graph size whose runtime is above one hour is still included.

⁶Excess bits are stored in a cache, and the next batch is only generated once the cache is empty.

 ${\rm ^7}$ High amounts of RAM were only used during malicious-security computations. Note that the RAM is shared between parties, i.e., each party had access to 42 GB RAM. ${}^{8}{\rm The}$ resulting protocol is very similar to the Dijkstra protocol by Anagreh et al. [\[4\]](#page-12-9) when executed on a graph with uniform edge lengths.

⁹Only the number of nodes is of importance as the protocols are data-oblivious.

Further, we evaluated the protocols on graphs of up to 100 nodes with 10 node increments. As shown in Figures [2](#page-8-6) and [3,](#page-9-0) the logarithmic protocol generally is the fastest protocol on these small graphs. The exceptions are in the malicious security setting, where the linear BFS protocol (Protocol [7\)](#page-17-0) is faster on graphs with less than 30 nodes. In this case, the simpler structure appears to outweigh the higher round complexity. Another exception occurs in the malicious security setting with an unrestricted network, where the comparison-free protocol (Protocol [8\)](#page-18-0) is the fastest on graph with less than 80 nodes. In all other settings, the protocol is one of the slowest protocols. This suggests that avoiding comparisons by having more communication rounds is only beneficial in the malicious security setting with very small network delays.

Finally, the overflow mitigation has only a small impact on the runtime of the logarithmic BFS protocol. On graphs with 100 nodes, it increases the runtime by 3.1% in the malicious security WAN setting.

This is similar on large graphs, where the biggest increase of runtime caused by the overflow mitigation is 5.4% in the semi-honest setting with 2600 nodes. The (previously slow) optimized verison of [\[11\]](#page-12-20) is also still slower than the logarithmic BFS protocol in the semi-honest WAN setting (Figure [5\)](#page-10-0) for graphs with up to 2400 nodes, and is faster on graphs with 2600 nodes. In the other presented settings, i.e., the semi-honest setting with an unrestricted network (Figure [4\)](#page-9-1) and the malicious security WAN setting (Figure [6\)](#page-10-1), the optimized version of [\[11\]](#page-12-20) is faster on graphs larger than 400, respectively 200, nodes. Further, the linear and comparison-free protocols are evaluated only up to 1600, respectively 600, nodes due to the runtime exceeding one hour.

The steep increase of the logarithmic BFS protocol's runtime cannot be attributed to the number of communication rounds, as Figure [7](#page-10-2) shows that it performs significantly fewer rounds than the other protocols. Instead, it can be attributed to the computation time of the protocol. This is shown by the runtime in the unrestricted network setting, where the protocol runs for roughly 3350 seconds on graphs with 2600 nodes. In the WAN setting, it runs for

Figure 4: Runtimes of semi-honest BFS protocols by number of nodes in the unrestricted network setting.

3740 seconds, which in an increase of only 11%. For comparison, it increases by 1800% (from 0.1 seconds to 1.8 seconds) on graphs with 100 nodes. Our investigation into this phenomenon revealed that the majority of the computation time is spent on matrix multiplications, whose total computation time over the execution of the protocol is in $O(n^3 \log n)$. In contrast, the protocol by Blanton et al. [\[11\]](#page-12-20) requires only $O(n^2)$ computation, allowing it to scale better to large graphs. This protocol in turn is bottlenecked by its high number of communication rounds, as its runtimes are roughly equal to the number of communication rounds (as presented in Figure [7\)](#page-10-2) times the delays of the evaluated network settings.

The difference in runtime is further increased in the malicious security setting by the amounts of data sent by the protocols. The logarithmic BFS protocol sends 100 GB in total on a graph of 400 nodes, whereas our optimized version of [\[11\]](#page-12-20) sends only 3,5 GB. In comparison, the logarithmic protocol sends only 190MB in the semihonest setting. Therefore, the semi-honest setting is not affected to a noticeable extent by the amounts of communication.

6.3 Maximal flow

In general, the maximal flow protocols are significantly slower than the BFS protocols. Therefore, we evaluate the protocols with up to 30 nodes in the semi-honest unrestricted and LAN settings and up to 25 nodes in the malicious security unrestricted and LAN settings. In the WAN settings, we evaluated the protocols with up to 15 nodes for the semi-honest and the malicious security setting.

Further, we evaluate the capacity scaling protocol (Protocol [3\)](#page-6-0) with capacity bit-lengths 63 , 10 10 10 $\overline{3}$ and 16 , as it is the only protocol in this paper whose round-complexity scales with the bit-length of the capacities. Finally, we want to note that we use our Edmonds-Karp protocol as baseline instead of [\[11\]](#page-12-20) since Section [6.2](#page-8-7) has showed that our logarithmic BFS protocol generally is faster than the BFS

 10 Note that SMPC primitives used in this evaluation use signed 64-bit words internally. Therefore, 63 bits is the maximal input capacity bit-length.

Figure 5: Runtimes of semi-honest BFS protocols by number of nodes in the WAN network setting.

Figure 6: Runtimes of malicious security BFS protocols by the number of nodes in the WAN network setting.

from [\[11\]](#page-12-20) on the evaluated graph sizes. Appendix [B](#page-14-0) presents an overview of all complexity measures of the evaluated protocols.

The resulting runtimes are very consistent in the sense that trends are very consistent across network and security settings. In other words, the runtimes mostly differ by a constant factor depending on the evaluated setting, as shown by Figures [8](#page-10-3) to [10.](#page-11-0) To avoid redundancy, we only present a representative selection of figures in this section. The full data can be found in Appendix [F.](#page-17-1)

This also allows us to describe the observed trends independently of the setting: In our evaluation, the Dinic-Tarjan protocol (Protocol [4\)](#page-7-1) generally is the fastest protocol. On the other hand, 63-bit capacity scaling protocol was the slowest protocol. Further, the 16-bit capacity scaling protocol consistently is twice as fast as the 32-bit capacity scaling protocol, which in turn consistently is roughly twice as fast as the 63-bit version. Finally, the Edmonds-Karp protocol is faster than the 16-bit capacity scaling protocol on

Figure 7: Communication rounds of the semi-honest BFS protocols by the number of nodes.

Figure 8: Runtime of semi-honest maximal flow protocols by number of nodes in the LAN network setting.

graphs with up to 15 nodes. It is slower than the 16-bit capacity scaling protocol on larger graphs, but still faster than the 32-bit capacity protocol. However, Figures [8](#page-10-3) and [9](#page-11-1) show that Edmonds-Karp scales worse than the 32-bit protocol. In the semi-honest LAN setting, its runtime is only slightly less than the runtime of the 32-bit capacity scaling protocol.

This is expected behaviour. The capacity scaling protocol performs xn^2 Edmonds-Karp iterations on x-bit capacities, whereas the Edmonds-Karp protocol performs n^3 iterations. Therefore, we expect the Edmonds-Karp protocol to be faster on graphs where the number of nodes is less than the capacity bit length. This can be observed in for the 16-bit capacity scaling protocol and can almost be observed for the 32-bit capacity scaling protocol in Figure [8.](#page-10-3) Therefore, we expect the Edmonds-Karp protocol to be slower than the 63-bit capacity scaling protocol on graphs larger than 63 nodes.

WPES '24, October 14–18, 2024, Salt Lake City, UT, USA Vincent Ehrmanntraut and Ulrike Meyer

Figure 9: Runtime of semi-honest maximal flow protocols by number of nodes in the WAN network setting.

Figure 10: Runtime of malicious security maximal flow protocols by number of nodes in the LAN network setting.

Finally, we also evaluated the capacity scaling protocol on threebit capacities and compare it with the Dinic-Tarjan protocol, which results in Figure [11.](#page-11-2) The figure shows that in the semi-honest LAN setting, the Dinic-Tarjan protocol is faster for five and ten nodes. On graphs of 15 nodes, the runtime is roughly equivalent, and the Dinic-Tarjan protocol is slower afterward. This is explained by the higher $O(n^3)$ round complexity of the Dinic-Tarjan protocol, compared to $O(n^2 \log n)$ of the three-bit capacity scaling protocol^{[11](#page-11-3)}. Therefore, we expect the capacity scaling protocols to eventually outperform the Dinic-Tarjan protocol, even with larger capacity bit lengths, as the runtime of the capacity scaling protocols increases only linearly with regard to the capacity bit length.

 20

Figure 11: Runtime of Protocol [3](#page-6-0) with 3 bit capacities and Protocol [4](#page-7-1) with semi-honest security in the LAN setting.

number of nodes

 $\overline{15}$

7 Conclusion and future work

 10

1000

800

400

200

 Ω

untime [s] 600

We presented novel protocols to perform BFS searches and determine maximal flows on dense graphs. On the BFS front, we presented a protocol with logarithmic round complexity, which is a big theoretical advancement from the $O(n \log n)$ round complexity of prior work. This protocol was shown to be efficient on small graphs, only being outperformed on large graphs by protocols with lower computational complexity.

By using the logarithmic BFS protocol, we were able to construct a maximal flow protocol based on the Edmonds-Karp algorithm where each iteration requires only $O(\log n)$ rounds. In total, this protocol requires $O(n^3 \log n)$ rounds.

We further reduced the number of iterations by introducing a capacity scaling protocol, which requires $O(n^2\log n\log U)$ rounds, where U is an upper bound on the capacities. The evaluation unsurprisingly showed that the capacity scaling protocol is faster than the Edmonds-Karp protocol when $\log U < n$.

Finally, we presented a novel maximal flow protocol based on algorithms by Dinic [\[20\]](#page-12-12) and Tarjan [\[38\]](#page-12-13). This protocol has a worse asymptotic round complexity of $O(n^3)$, but our evaluation showed that the protocol is up to seven times faster than the Edmonds-Karp protocol. It also outperformed the capacity scaling protocol, except when the capacities' upper bound is small.

One direction for future work on BFS protocols is to find further trade-offs between the round complexity and the computation complexity. In other words, the practical runtime of privacy-preserving BFS may be further improved by future protocols that have lower computational complexity, although such protocols are likely to have a higher round complexity than our logarithmic BFS protocol.

For maximal flow protocols, one direction of future work is the development of protocols based on recent advancements in nonprivacy-preserving graph algorithms like [\[15,](#page-12-26) [27\]](#page-12-25). These algorithms have small asymptotic complexities. However, they rely on nonlinearities like logarithms while requiring high numerical precision and stability, as well as dynamic data structures, making SMPC implementations of such protocols challenging.

 $2₅$

 30

¹¹The general round complexity of capacity scaling protocols is $O(n^2 \log n \log U)$, but in the case of the 3-bit protocol " $\log U$ " is fixed to 3.

References

- [1] R. K. Ahuja and James B. Orlin. 1989. A Fast and Simple Algorithm for the Maximum Flow Problem. Operations Research 37, 5 (1989). [https://doi.org/10.](https://doi.org/10.1287/opre.37.5.748) [1287/opre.37.5.748](https://doi.org/10.1287/opre.37.5.748)
- [2] Abdelrahaman Aly and Sara Cleemput. 2022. A Fast, Practical and Simple Shortest Path Protocol for Multiparty Computation. In Computer Security - ESORICS 2022 (Lecture Notes in Computer Science). Springer Nature Switzerland, Cham. https://doi.org/10.1007/978-3-031-17143-7_37
- [3] Abdelrahaman Aly and Mathieu Van Vyve. 2015. Securely Solving Classical Network Flow Problems. In Information Security and Cryptology 2014 (Lecture Notes in Computer Science). Springer International Publishing, Cham. [https:](https://doi.org/10.1007/978-3-319-15943-0_13) [//doi.org/10.1007/978-3-319-15943-0_13](https://doi.org/10.1007/978-3-319-15943-0_13)
- [4] Mohammad Anagreh, Peeter Laud, and Eero Vainikko. 2021. Parallel Privacy-Preserving Shortest Path Algorithms. Cryptography 5, 4 (2021). [https://doi.org/](https://doi.org/10.3390/cryptography5040027) [10.3390/cryptography5040027](https://doi.org/10.3390/cryptography5040027)
- [5] Mohammad Anagreh, Peeter Laud, and Eero Vainikko. 2022. Privacy-Preserving Parallel Computation of Shortest Path Algorithms with Low Round Complexity.. In 8th International Conference on Information Systems Security and Privacy. SCITEPRESS.<https://doi.org/10.5220/0010775700003120>
- [6] Toshinori Araki, Jun Furukawa, Yehuda Lindell, Ariel Nof, and Kazuma Ohara. 2016. High-Throughput Semi-Honest Secure Three-Party Computation with an Honest Majority. In 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16). ACM, New York, NY, USA. [https://doi.org/10.1145/](https://doi.org/10.1145/2976749.2978331) [2976749.2978331](https://doi.org/10.1145/2976749.2978331)
- [7] Toshinori Araki, Jun Furukawa, Kazuma Ohara, Benny Pinkas, Hanan Rosemarin, and Hikaru Tsuchida. 2021. Secure Graph Analysis at Scale. In 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS '21). ACM, New York, NY, USA.<https://doi.org/10.1145/3460120.3484560>
- [8] B. Awerbuch and R. Gallager. 1987. A New Distributed Algorithm to Find Breadth First Search Trees. Transactions on Information Theory 33, 3 (1987). [https:](https://doi.org/10.1109/TIT.1987.1057314) [//doi.org/10.1109/TIT.1987.1057314](https://doi.org/10.1109/TIT.1987.1057314)
- [9] Alessandro Baccarini, Marina Blanton, and Shaofeng Zou. 2024. Understanding Information Disclosure from Secure Computation Output: A Study of Average Salary Computation. In Proceedings of the Fourteenth ACM Conference on Data and Application Security and Privacy (CODASPY '24). Association for Computing Machinery, New York, NY, USA, 187–198.<https://doi.org/10.1145/3626232.3653280>
- [10] Marina Blanton and Siddharth Saraph. 2015. Oblivious Maximum Bipartite Matching Size Algorithm with Applications to Secure Fingerprint Identification. In Computer Security – ESORICS 2015 (Lecture Notes in Computer Science). Springer International Publishing, Cham. https://doi.org/10.1007/978-3-319-24174-6_20
- [11] Marina Blanton, Aaron Steele, and Mehrdad Alisagari. 2013. Data-Oblivious Graph Algorithms for Secure Computation and Outsourcing. In 8th ACM SIGSAC Symposium on Information, Computer and Communications Security (ASIA CCS '13). ACM, New York, NY, USA.<https://doi.org/10.1145/2484313.2484341>
- [12] Peter Bogetoft, Dan Lund Christensen, Ivan Damgård, Martin Geisler, Thomas Jakobsen, Mikkel Krøigaard, Janus Dam Nielsen, Jesper Buus Nielsen, Kurt Nielsen, Jakob Pagter, Michael Schwartzbach, and Tomas Toft. 2009. Secure Multiparty Computation Goes Live. In Financial Cryptography and Data Security. Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-642-03549-4_20 [13] R. Canetti. 2001. Universally Composable Security: A New Paradigm for Crypto-
- graphic Protocols. In 42nd IEEE Symposium on Foundations of Computer Science.
IEE. <https://doi.org/10.1109/SFCS.2001.959888>
- [14] Javad Ghareh Chamani, Ioannis Demertzis, Dimitrios Papadopoulos, Charalampos Papamanthou, and Rasool Jalili. 2023. GraphOS: Towards Oblivious Graph Processing. Proceedings of the VLDB Endowment 16, 13 (2023). <https://doi.org/10.14778/3625054.3625067>
- [15] Li Chen, Rasmus Kyng, Yang P. Liu, Richard Peng, Maximilian Probst Gutenberg, and Sushant Sachdeva. 2022. Maximum Flow and Minimum-Cost Flow in Almost-Linear Time. In 2022 IEEE 63rd Annual Symposium on Foundations of Computer Science. IEEE.<https://doi.org/10.1109/FOCS54457.2022.00064>
- [16] Paul Christiano, Jonathan A. Kelner, Aleksander Madry, Daniel A. Spielman, and Shang-Hua Teng. 2011. Electrical Flows, Laplacian Systems, and Faster Approximation of Maximum Flow in Undirected Graphs. In 43rd Annual ACM Symposium on Theory of Computing. ACM, San Jose, California, USA. [https:](https://doi.org/10.1145/1993636.1993674) [//doi.org/10.1145/1993636.1993674](https://doi.org/10.1145/1993636.1993674)
- [17] Christophe Clavier, Benoit Feix, Georges Gagnerot, Mylène Roussellet, and Vincent Verneuil. 2011. Square Always Exponentiation. In Progress in Cryptology – INDOCRYPT 2011. Springer, Berlin, Heidelberg. [https://doi.org/10.1007/978-3-](https://doi.org/10.1007/978-3-642-25578-6_5) [642-25578-6_5](https://doi.org/10.1007/978-3-642-25578-6_5)
- [18] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. 2001. Introduction to Algorithms, Second Edition. The MIT Press and McGraw-Hill Book Company.
- [19] Ronald Cramer, Ivan Damgård, Daniel Escudero, Peter Scholl, and Chaoping Xing. 2018. SPDZ2k Efficient MPC Mod 2^k for Dishonest Majority. In Advances in $Cryptology - CRYPTO 2018$. Springer International Publishing, Cham. [https:](https://doi.org/10.1007/978-3-319-96881-0_26) [//doi.org/10.1007/978-3-319-96881-0_26](https://doi.org/10.1007/978-3-319-96881-0_26)
- [20] Yefim Dinic. 1970. Algorithm for Solution of a Problem of Maximum Flow in Networks with Power Estimation. Soviet Math. Doklady 11 (1970).
- [21] Jack Edmonds and Richard M Karp. 1972. Theoretical Improvements in Algorithmic Efficiency for Network Flow Problems. J. ACM 19, 2 (1972).
- [22] Hendrik Eerikson, Marcel Keller, Claudio Orlandi, Pille Pullonen, Joonas Puura, and Mark Simkin. 2020. Use Your Brain! Arithmetic 3PC for Any Modulus with Active Security: 1st Conference on Information-Theoretic Cryptography, ITC 2020. In 1st Conference on Information-Theoretic Cryptography (Leibniz International Proceedings in Informatics). Schloss Dagstuhl–Leibniz-Zentrum für Informatik.<https://doi.org/10.4230/LIPIcs.ITC.2020.5>
- [23] Lester R. Ford and Delbert R. Fulkerson. 1956. Maximal Flow Through a Network. Canadian Journal of Mathematics 8 (1956). [https://doi.org/10.4153/CJM-1956-](https://doi.org/10.4153/CJM-1956-045-5) [045-5](https://doi.org/10.4153/CJM-1956-045-5)
- [24] Harold N. Garbow. 1985. Scaling Algorithms for Network Problems. J. Comput. System Sci. 31, 2 (1985). [https://doi.org/10.1016/0022-0000\(85\)90039-X](https://doi.org/10.1016/0022-0000(85)90039-X)
- [25] Brett Hemenway, Steve Lu, Rafail Ostrovsky, and William Welser IV. 2016. High-Precision Secure Computation of Satellite Collision Probabilities. In Security and Cryptography for Networks. Springer International Publishing, Cham. [https:](https://doi.org/10.1007/978-3-319-44618-9_9) [//doi.org/10.1007/978-3-319-44618-9_9](https://doi.org/10.1007/978-3-319-44618-9_9)
- [26] Dorit S. Hochbaum. 2001. A New—Old Algorithm for Minimum-Cut and Maximum-Flow in Closure Graphs. Networks 37, 4 (2001), 171–193. [https:](https://doi.org/10.1002/net.1012) [//doi.org/10.1002/net.1012](https://doi.org/10.1002/net.1012)
- [27] Tarun Kathuria. 2020. A Potential Reduction Inspired Algorithm for Exact Max Flow in Almost O(M^(4/3)) Time. arxiv.org. [https://doi.org/10.48550/arXiv.2009.](https://doi.org/10.48550/arXiv.2009.03260) [03260](https://doi.org/10.48550/arXiv.2009.03260) arXiv[:2009.03260](https://arxiv.org/abs/2009.03260) [cs]
- [28] Marcel Keller. 2020. MP-SPDZ: A Versatile Framework for Multi-Party Computation. In 2020 ACM SIGSAC Conference on Computer and Communications Security (CCS '20). ACM, New York, NY, USA.<https://doi.org/10.1145/3372297.3417872>
- [29] Marcel Keller and Peter Scholl. 2014. Efficient, Oblivious Data Structures for MPC. In Advances in Cryptology – ASIACRYPT 2014 (Lecture Notes in Computer Science).
- Springer, Berlin, Heidelberg. https://doi.org/10.1007/978-3-662-45608-8_27 [30] Marcel Keller, Peter Scholl, and Nigel P. Smart. 2013. An Architecture for Practical Actively Secure MPC with Dishonest Majority. In 2013 ACM SIGSAC Conference on Computer & Communications Security (CCS '13). ACM, New York, NY, USA. <https://doi.org/10.1145/2508859.2516744>
- [31] Andreas Klinger, Vincent Ehrmanntraut, and Ulrike Meyer. 2024. Estimating the Runtime and Global Network Traffic of SMPC Protocols. In Fourteenth ACM Conference on Data and Application Security and Privacy (CODASPY '24). ACM, New York, NY, USA.<https://doi.org/10.1145/3626232.3653258>
- [32] Donald E. Knuth. 1981. Section 4.6.3. Evaluation of Powers. In Seminumerical Algorithms (second ed.). The Art of Computer Programming, Vol. 2. Addison-Wesley
- [33] Payman Mohassel and Peter Rindal. 2018. ABY3: A Mixed Protocol Framework for Machine Learning. In 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18). ACM, New York, NY, USA. [https:](https://doi.org/10.1145/3243734.3243760) [//doi.org/10.1145/3243734.3243760](https://doi.org/10.1145/3243734.3243760)
- [34] Kartik Nayak, Xiao Shaun Wang, Stratis Ioannidis, Udi Weinsberg, Nina Taft, and Elaine Shi. 2015. GraphSC: Parallel Secure Computation Made Easy. In 2015 IEEE Symposium on Security and Privacy. IEEE.<https://doi.org/10.1109/SP.2015.30>
- [35] Benjamin Ostrovsky. 2024. Privacy-Preserving Dijkstra. In Advances in Cryptology – CRYPTO 2024. Springer Nature Switzerland, Cham. [https://doi.org/10.1007/978-](https://doi.org/10.1007/978-3-031-68400-5_3) [3-031-68400-5_3](https://doi.org/10.1007/978-3-031-68400-5_3)
- [36] Antonio Sedeño-Noda, Carlos González-Martín, and Sergio Alonso. 2004. A Generalization of the Scaling Max-Flow Algorithm. Computers & Operations Research 31, 13 (2004). [https://doi.org/10.1016/S0305-0548\(03\)00171-0](https://doi.org/10.1016/S0305-0548(03)00171-0)
- [37] Steven L. Tanimoto, Alon Itai, and Michael Rodeh. 1978. Some Matching Problems for Bipartite Graphs. J. ACM 25, 4 (1978).<https://doi.org/10.1145/322092.322093>
- [38] Robert Endre Tarjan. 1984. A Simple Version of Karzanov's Blocking Flow Algorithm. Operations Research Letters 2, 6 (1984). [https://doi.org/10.1016/0167-](https://doi.org/10.1016/0167-6377(84)90076-2) [6377\(84\)90076-2](https://doi.org/10.1016/0167-6377(84)90076-2)
- [39] Tomas Toft. 2009. Solving Linear Programs Using Multiparty Computation. In Financial Cryptography and Data Security. Springer, Berlin, Heidelberg. [https:](https://doi.org/10.1007/978-3-642-03549-4_6) [//doi.org/10.1007/978-3-642-03549-4_6](https://doi.org/10.1007/978-3-642-03549-4_6)
- [40] Xiao Shaun Wang, Kartik Nayak, Chang Liu, T-H. Hubert Chan, Elaine Shi, Emil Stefanov, and Yan Huang. 2014. Oblivious Data Structures. In 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS '14). ACM, New York, NY, USA.<https://doi.org/10.1145/2660267.2660314>
- [41] Stefan Wüller. 2018. Privacy-Preserving Electronic Bartering. Ph. D. Dissertation. RWTH Aachen University.<https://doi.org/10.18154/RWTH-2018-222969>

A Example application

This section presents a toy example application of the maximal flow protocols presented in this paper. For this application, consider N

Figure 12: Example of the graph construction used to solve the chemical plants' problem. Edges with zero capacity are not displayed.

chemical plants that produce a chemical, for example, acetylsali-cylic acid, also known as Aspirin^{[12](#page-13-0)}. These plants produce Aspirin of the same quality, i.e., the output from the first plant is not distinguishable from the second's plant output. However, they suffer from variations in the input chemicals, which leads to variations in the yield of their production and therefore in the output quantities. Similarly, the demand for Aspirin varies. This leads to situations in which factories produced more Aspirin than they are able to sell, or in which factories failed to produce enough Aspirin to meet contractual obligations.

Therefore, the plants decide to cooperate and trade Aspirin amongst each other. This allows overproducing plants with excess capacities to sell their remaining stock, and under-producing plants to meet contractual obligations. To maximize the total welfare, they agree that they want to maximize the amount of Aspirin traded amongst the plants. Note that the maximizing trades need not to fully meet the demands and excesses.

Such maximizing trades can be found using maximal flows: Plants that overproduce Aspirin are connected to the source, with the corresponding edge's capacities being the excesses of the plants. Similarly, plants that need Aspirin are connected to the sink. The capacities of these edges correspond to the plants' demands. Finally, overproducing plants are connected to underproducing plants. These edges have capacity ∞ , which is an upper bound on the plants' excesses. Further, these edges may be added only conditionally, e.g., only when the asking price of the overproducing plant is lower than the bidding price of the underproducing plant.

Figure [12](#page-13-1) displays an example graph construction. The overproducing plants P_2 , P_4 , P_5 , identified by having a positive amount on the left side of Figure [12,](#page-13-1) are connected to the source node s on the right side of Figure [12.](#page-13-1) The underproducing plants P_1 , P_3 , identified by having a negative amount, are connected to the sink node t . Further, each plant has a minimal price at which it is willing to sell Aspirin, or a maximal price at which it is willing to buy Aspirin. An overproducing plant is only connected to an underproducing plant when its selling price is lower than the buying price of the underproducing plant. For example, P_5 is not connected to P_3 because P_3 is willing to pay at most 73 000\$ per tonne, whereas P_5 asks for at least 75 000\$ per tonne.

Trades that maximize the total Aspirin sold can be found by finding a maximal flow from s to t : The flow from an overproducing

plant to an underproducing plant indiciates the amount of Aspirin the overproducing plant sells to the underproducing plant. We assume that two plants are able to negotiate a price once they were assigned a trade by this system.

Traditionally, using such a system would require a trusted (third) party. All plants must be willing to share their over-/underproductions and prices with the third party. However, as the chemical plants consider their production quantities and the prices at which they are able to produce and sell Aspirin important trade secrets, it is unlikely that all plants agree on a trusted third party. Therefore, Protocol [5](#page-13-2) is used to find the trades.

The protocol first receives the secret-shared inputs from the plants and builds the corresponding graph in Lines [1](#page-13-2) to [8.](#page-13-2) Lines [9](#page-13-2) to [11](#page-13-2) then build the unit-vector representations of the source and sink nodes. After, any of the maximal flow protocols presented in this paper can be used to find the maximal flow in the graph, and the resulting trades are revealed to the plants.

Note that the chemical plants only must be able to secret-share their amounts and prices, and be able to receive the shares of the computed flow. This decouples the plants from the computation, as they do not have to actively take part in the computation of the maximal flow. For example, the actual computations could be carried out by only three of the plants, or two plants and a regulatory body. Using the same primitives as the evaluation in Section [6,](#page-8-0) this would be considered secure as long the plants trust that no two computing parties collude. This has the advantage that only the computing parties are burdened with performing the computation and the associated IT overheads.

Other SMPC primitives and computing party setups lead to different security guarantees. For example, when each party actively

 12 Note that we know next to nothing about chemical production and even less about the production of Aspirin. The scenario presented in this section is purely hypothethical.

takes part in the computation, and [\[19\]](#page-12-16) is used as SMPC primitive, then each plant only has to trust itself. In other words, a plant can assume that all other plants have colluded against it, and even actively deviate from the protocol, and still be confident that the protocol does not leak its inputs.

Finally, it should be stressed that SMPC in general aims to replace a trusted third party. Hence, if a functionality can be used to infer private inputs when a trusted third party is used, then this is also possible when using SMPC protocols. For example, if all but one chemical plant collude against one plant, they may be able to craft malicious amounts and prices such that they learn the target plant's inputs from the resulting maximal flow. This attack could be performed independently of the SMPC protocol used to compute the maximal flow, it can even be used in a trusted third party setting. The leakage is not caused by the protocol but by the computed functionality. The curious reader is referred to [\[9\]](#page-12-41) for an in-depth discussion on the leakages caused by functionalities.

B Protocol complexities

Tables [1](#page-15-1) and [2](#page-15-2) present the asymptotic complexities of the protocols evaluated in this paper. The communication and computation complexities ignore polynomial dependencies on the number of parties, since different SMPC primitives induce different overheads. The communication complexity also ignores polynomial dependencies on λ . We also separate the shuffling of the adjacency matrix, as the primitives that can be used for shuffling are dependent on the security setting and the number of parties.

C Additional proofs

This section contains a proof that the reachable nodes in a graph can be calculated by exponentiating the adjacency matrix (Appen- dix C.1 , as well as a proof that intermediary values remain in $\{0, 1\}$ in the Edmonds-Karp protocol (Appendix [C.2\)](#page-14-2).

C.1 Powers of the adjacency matrix yield reachable nodes

The logarithmic BFS protocol is based on the observation that when $G = (V = \{1, \ldots, n\}, E)$ is a graph and A the corresponding adjacency matrix, then $\overline{E^{v,n}}$. A^i indicates the nodes that are reachable in exactly i steps from v . This observation is formally proven in this section.

We start by formally defining what "indicates" means in this context by defining an encoding from sets of nodes to vectors.

Definition C.1. Let $V = \{1, \ldots, n\}$ be the set of nodes of a graph, and $S \subseteq V$. We say that $\overrightarrow{E} \in \mathbb{N}^n$ is a relaxed node set encoding of S if and only if $\overrightarrow{E}_i \neq 0$ for all $i \in S$ and $\overrightarrow{E}_i = 0$ for all $i \in V \setminus S$.

For the sake of completeness, we also formally define the set of nodes reachable from v in exactly l steps in Definition [C.2.](#page-14-3)

Definition C.2. Let $G = (V, E)$ be a graph and $l \in \mathbb{N}$. Then $R(v, l)$ is the set of the nodes that can be reached from v in exactly l steps, defined as

$$
R(G, v, 0) = \{v\}
$$

$$
R(G, v, l) = \{w \in V \mid \exists u \in R(v, l-1) : (u, w) \in E\}
$$

Using these definitions, we can properly formalize the observation in Lemma [C.3,](#page-14-4) which is proven by induction.

LEMMA C.3. Let $G = (V = \{1, ..., n\}, E)$ be a graph and $A \in \{0, 1\}^{n \times n}$ the corresponding adjacency matrix. Then $\overline{E^{v,n}} \cdot A^i$ is a relaxed node set encoding of $R(G, v, i)$ for all $v \in V, i \in \mathbb{N}$.

PROOF. First, we note that for $i = 0$, $\overrightarrow{E^{v,n}} \cdot A^i = \overrightarrow{E^{v,n}} \cdot A^0 = \overrightarrow{E^{v,n}}$. which is a relaxed node set encoding of $\{v\} = R(G, v, 0) = R(G, v, i)$.

Next, we assume that Lemma [C.3](#page-14-4) holds for $i \in \mathbb{N}$ and show that it also holds for $i + 1$. To this end, let $\vec{X} = \vec{E}^{v,n} \cdot \vec{A}^i$ be (by induction assumption) a relaxed node set encoding of the $R(G, v, i)$. Then by Δ associativity $\overrightarrow{Y} = \overrightarrow{E^{v,n}} \cdot A^{i+1} = (\overrightarrow{E^{v,n}} \cdot A^i) \cdot A = \overrightarrow{X} \cdot A$. Hence, we only need to show that $\overrightarrow{X} \cdot A$ is a relaxed node set encoding of $R(G, v, i + 1).$

We first show that $w \in R(G, v, i + 1)$ for all $w \in V$ such that $\overrightarrow{Y}_w \neq 0$.

$$
\overrightarrow{Y}_{w} = \sum_{u=1}^{n} \overrightarrow{X}_{u} A_{u,w} \neq 0
$$

\n
$$
\Rightarrow \exists u \in V : \overrightarrow{X}_{u} A_{u,w} \neq 0
$$

\n
$$
\Rightarrow \exists u \in V : \overrightarrow{X}_{u} \neq 0 \land A_{u,w} \neq 0
$$

\n
$$
\Rightarrow \exists u \in V : u \in R(G, v, i) \land (u, w) \in E
$$

\n
$$
\Rightarrow w \in R(G, v, i+1)
$$

Therefore $\overrightarrow{Y}_w = 0$ for all $w \notin R(G, v, i+1)$.

Next, we show that $\overrightarrow{Y}_w \neq 0$ for all $w \in R(G, v, i + 1)$.

$$
\overrightarrow{Y}_{w} = \sum_{u=1}^{n} \overrightarrow{X}_{u} A_{u,w}
$$

$$
= \sum_{u \in R(G,v,i)} \overrightarrow{X}_{u} A_{u,w}
$$

$$
= \sum_{u \in R(G,v,i): A_{u,w}=1} \overrightarrow{X}_{u}
$$

Since $\overrightarrow{X} \in \mathbb{N}^n$ and $\overrightarrow{X}_u \neq 0$ for all $u \in R(G, v, i)$ by definition, $\overrightarrow{X}_u > 0$ for all $u \in R(G, v, i)$. It follows that $\overrightarrow{Y}_w > 0$ and therefore $\overrightarrow{Y}_w \neq 0$ for all $w \in R(G, v, i+1)$.

Therefore, $\overrightarrow{Y}_w \neq 0$ if and only if $w \in R(G, v, i+1)$, which implies that $\overrightarrow{Y}_w = 0$ for all $w \notin R(G, v, i + 1)$. Hence, $\overrightarrow{Y} = \overrightarrow{E^{v,n}} \cdot A^{i+1}$ is a relaxed node set encoding of $R(G, v, i + 1)$, which concludes the induction.

C.2 Edmonds-Karp does not need overflow mitigation

The maximal flow protocol presented in Section [5.1](#page-5-0) uses the squareand-multiply algorithm to extract an augmenting path from a BFS tree matrix. In contrast to the logarithmic BFS protocol, this protocol does not need to worry about integer overflows, as the (transposed) BFS tree matrix contains at most one 1 in each row. Lemma [C.4](#page-14-5) shows that this property propagates through matrix multiplications. As argued in Section [5.1,](#page-5-0) this means that all intermediary values in the protocol execution will remain in {0, 1}, and hence no overflows occur.

LEMMA C.4. Let $A \in \{0,1\}^{x \times y}, B \in \{0,1\}^{y \times z}$ such that each row contains at most one 1. Then $C = AB$ is in $\{0, 1\}^{x \times z}$ and each row of C contains at most one 1.

• Protocol implements the Dijkstra algorithm, BFS is simulated by uniform edge weights.

 \dagger Protocol designed for spare graphs. m denotes the number of edges.

 \star S denotes the effort required to permute the adjacency matrix.

When using replicated or Shamir's secret sharing, less communication is required in the semi-honest setting.

 \star S denotes the effort required to permute the adjacency matrix.

 \dagger Protocol only works on planar graphs. m denotes the number of edges and dim the doubling dimension.

 $^{\diamond}$ When using replicated or Shamir's secret sharing, less communication is required in the semi-honest setting.

PROOF. First, we show that $C_{i,j} \in \{0,1\}$ for all $1 \leq i \leq x, 1 \leq j$ $j \leq z$. To do so, let 1 ≤ $i \leq x$, 1 ≤ $j \leq z$ be arbitrary, but fixed. We now differentiate between two cases:

(1) The row $A_{i, \bullet}$ does not contain any ones. Then $C_{i, j}$ is zero.

(2) The row $A_{i,\bullet}$ does contain a 1. Let u be the corresponding index such that $A_{i,u} = 1$. Then:

$$
C_{i,j} = \sum_{k=1}^{y} A_{i,k} B_{k,j} = \sum_{k=1, k \neq u}^{y} A_{i,k} B_{k,j} + A_{i,u} B_{u,j}
$$

=
$$
\sum_{k=1, k \neq u}^{y} 0 \cdot B_{k,j} + 1 \cdot B_{u,j} = B_{u,j} \in \{0, 1\}
$$

Next, we prove by contradiction that each row of C contains at most one 1. Assume that there is a row in C that contains multiple ones, i.e., there are i , j_1 , j_2 such that $C_{i,j_1} = C_{i,j_2} = 1$. We know that $A_{i,\bullet}$ contains only one 1. Let u be the corresponding index

$$
(A_{i,u} = 1)
$$
. Then:

$$
C_{i,j_1} = \sum_{k=1}^{y} A_{i,k} B_{k,j_1} = A_{i,u} B_{u,j_1} = B_{u,j_1} = 1
$$

$$
C_{i,j_2} = \sum_{k=1}^{y} A_{i,k} B_{k,j_2} = A_{i,u} B_{u,j_2} = B_{u,j_2} = 1
$$

Hence, the row $B_{u,\bullet}$ contains multiple ones, which is a contradiction. It follows that each row of C can contain at most one 1. \Box

D Overflows in the logarithmic BFS protocol

Recall that during the logarithmic BFS protocol from Section [4,](#page-3-3) the adjacency matrix is exponentiated. As a consequence, the intermediary values are not limited to $\{0, 1\}$, but might be arbitrary natural numbers, which leads to the potential danger of overflows in practice. These protocols can be handled in two ways: They can either be ignored, which leads to a risk of incorrect results, or they can be prevented. Appendix [D.1](#page-16-2) discusses the consequences of ignoring overflows in detail, whereas Appendix [D.2](#page-16-3) presents a way to prevent overflows.

D.1 Impact of overflows

The overflows that occur during the protcol execution can be classified into two categories, "bad" overflows which convert non-zero values to zero, and "good" overflows, where non-zero values remain non-zero. Good overflows do not affect the semantic interpretation of the value and hence to not affect the correctness of the protocol. However, bad overflows cause the protocol to ignore nodes and edges, which can affect the correctness of the protocol.

The probability that a bad overflow occurs depends on the input graph. For example, Figure [13](#page-16-4) displays a graph structure that produces powers of two when the adjacency matrix is exponentiated during the protocol. When the underlying SMPC primitives operate modulo a power of two, like the primitives used in Section [6,](#page-8-0) bad overflows are very likely to occur.

Figure 13: A graph structure that produces powers of two when the adjacency matrix is exponetiated.

Furthermore, the impact of a bad overflow can vary. To demonstrate this, we use Figure [14](#page-16-5) as an example input graph. If the protocol ignores node 2 because of a bad overflow, then nodes 2, 3, and 4 would be considered to be unreachable and the resulting BFS tree is empty. A different form of incorrectness occurs if the protocol ignores the edge (2, 4) because of a bad overflow. In that case, the protocol would return an incorrect "shortest" path from node 1 to node 4. However, not all bad overflows affect the correctness of the protocol, if the edge (3, 4) would be ignored, the protocol would still return a correct BFS tree since the edge is not part of the result.

Due to these reasons, it is hard to determine the impact of ignored overflows. Hence, we generally suggest using the overflow mitigation presented in the following section. Section [6](#page-8-0) shows that its impact on the runtime should be acceptable in most cases. Otherwise, we recommend a thorough evaluation of the potential impact of non-correct results on the protocol that uses the BFS protocol.

Figure 14: An example graph.

D.2 Preventing overflows

Overflows during the square-and-multiply procedure could be prevented by "resetting" non-zero values to 1 after each multiplication. However, since every reset step consists of $O\left(n^2\right)$ parallel comparisons, this would introduce an unreasonable overhead. Instead, the overflow-mitigating comparisons are only performed as needed in the worst-case scenario, which is a fully connected graph. On such graphs, the powers of the adjacency matrix match the upper-bound given by Lemma [D.1.](#page-16-6)

LEMMA D.1. Let the enties of $A \in \mathbb{R}^{x \times y}$, $B \in \mathbb{R}^{y \times z}$ be upperbounded by \overline{a} and \overline{b} , i.e., $A_{i,j} \leq \overline{a}$ for all $1 \leq i \leq x, 1 \leq j \leq y$ and $B_{i,j} \leq \overline{b}$ for all $1 \leq i \leq y, 1 \leq j \leq z$. Then the entries of $C = AB$ are upper-bounded by $\overline{c} = y \overline{a} b$.

PROOF. For all $1 \le i \le x, 1 \le j \le z$ it holds that:

$$
C_{i,j} = \sum_{k=1}^{y} A_{i,k} B_{k,j} \le \sum_{k=1}^{y} \overline{ab} = y \overline{ab} = \overline{c}
$$

 \Box

Protocol [6](#page-17-3) keeps track of these upper-bounds. Before each matrix multiplication, it is checked whether an overflow might be possible. If that is the case, the overflow-mitigating comparisons are performed and non-zero values are reset to 1, thus preventing the overflow.

An important aspect of this mitigation strategy is that the control flow of the protocol is independent of the actual input graph and only depends on the number of nodes. In other words, our protocol remains data-oblivious. Further, the protocol still only uses secure gates and does not reveal any intermediary values, and therefore can be simulated using the composition theorem [\[13\]](#page-12-17).

E Other BFS protocols

The evaluation in Section [6](#page-8-0) not only presents the runtime of the logarithmic BFS protocol described in Section [4,](#page-3-3) but also presents the runtimes of further BFS protocol with worse round complexities. These protocols are presented in this section.

E.1 BFS with linear round complexity

One of the additionally evaluated BFS protocols has a linear round complexity. In this protocol, all values remain bounded to {0, 1}, so while this protocol does have a worse round complexity than the logarthmic BFS protocol, one does not need to worry about overflows when using this protocol.

The linear round complexity is achieved by iteratively visiting the layers of the graph. Each iteration first marks the currently visited layer as seen and then finds all edges to unseen nodes in Lines [7](#page-17-0) to [9](#page-17-0) of Protocol [7.](#page-17-0) These edges point towards the next layer to be visited. However, there might be multiple edges pointing towards the same node, similar to the layered graph in the logarithmic BFS protocol. Therefore, the same filter step as in the logarithmic BFS protocol is applied to the edges in Lines [10](#page-17-0) to [13.](#page-17-0) Those edges are then added to the resulting BFS tree and the next layer to be visited is determined.

This protocol follows a control flow that only depends on the number of nodes, never reveals the values of secret-shares, and WPES '24, October 14–18, 2024, Salt Lake City, UT, USA Vincent Ehrmanntraut and Ulrike Meyer

 $P_{rotocal}$ 6 Π_{spec}

only calls secure gates. Therefore, it can be simulated using the composition theorem [\[13\]](#page-12-17).

E.2 BFS without comparisons

Another BFS protocol used in the evaluation does not require any comparisons, which are expensive in malicious-security settings, but requires $O(n \log n)$ communication rounds. It operates similar to the classical (non-privacy-preserving) BFS in the sense that it iteratively fetches a node to visit from a queue and adds previously unseen neighbors to the queue.

The queue is implemented as a big vector of size n^2 , where $\overline{Queue}_i = 1$ means that the node *i* mod *n* is currently enqueued. This allows efficiently adding the neighbors of the current node by writing to the corresponding entries of the queue vector, as done in Line [15](#page-18-0) of Protocol [8.](#page-18-0) Extracting the next node to be visited is then equivalent to finding the first one in the queue vector. This could be done using the same filtering method used in Π_{BFS-LOG} and ΠBFS-linear, which would result in another linear-round protocol, but that approach would require comparisons which we want to avoid in this protocol. Therefore, the sub-protocol SELECTFIRstOne is used. This sub-protocol iteratively builds a new temporary vector that is half as big as the input vector, recursively find the first one in the temporary vector, and then expands the sub-result

to the full size. In total, this approach requires $2\log n^2 = 4\log n$ communication rounds.

Once the node to be visited been fetched from the queue, all neighbors are visited in parallel, which is done in $O(1)$ communication rounds in Lines [13](#page-18-0) to [18](#page-18-0) of Protocol [8.](#page-18-0) This results in a total of O ($n \log n$) communication rounds.

Like the linear protocol, this protocol also follows a control flow that only depends on the number of nodes, never reveals the values of secret-shares, and only calls secure gates. Therefore, it can be simulated using the composition theorem [\[13\]](#page-12-17).

F Numerical evaluation results

This section presents the runtimes of the BFS protocol evaluation (Tables [3](#page-19-0)to [10\)](#page-22-0) and of the maximal flow protocol evaluation (Tables [11](#page-22-1)to [18\)](#page-24-1). The tables present the runtime in seconds, the gray cells indicate the fastest protocol for each row, and ⊥ means that a protocol was not evaluated in that setting. The number of rounds and the total data sent by the parties does not depend on the network setting, these informations are only provided per protocol and used SMPC primitive in Tables [6,](#page-20-0) [10,](#page-22-0) [14,](#page-23-0) and [18.](#page-24-1)

| Nodes | Logarithmic | Logarithmic w/o overflow mitigation | Optimized [11] | Linear | Comparison-free |
|-------|---------------|-------------------------------------|----------------|---------------|-----------------|
| 10 | 0.01 | 0.01 | 0.04 | 0.02 | 0.02 |
| 20 | 0.01 | 0.01 | 0.09 | 0.04 | 0.07 |
| 30 | 0.02 | 0.01 | 0.17 | 0.08 | 0.13 |
| 40 | 0.02 | 0.02 | 0.24 | 0.14 | 0.27 |
| 50 | 0.03 | 0.03 | 0.31 | 0.20 | 0.48 |
| 60 | 0.04 | 0.04 | 0.40 | 0.29 | 0.76 |
| 70 | 0.06 | 0.05 | 0.47 | 0.42 | 1.22 |
| 80 | 0.07 | 0.06 | 0.56 | 0.57 | 1.86 |
| 90 | 0.09 | 0.08 | 0.61 | 0.80 | 2.72 |
| 100 | 0.11 | 0.10 | 0.67 | 1.05 | 3.75 |
| 200 | 0.58 | 0.55 | 1.52 | 7.41 | 45.66 |
| 400 | 3.82 | 3.64 | 3.51 | 56.00 | 706.71 |
| 600 | 14.35 | 13.75 | 5.87 | 189.65 | 3826.38 |
| 800 | 40.75 | 38.89 | 8.47 | 432.70 | ⊥ |
| 1000 | 69.45 | 69.36 | 11.11 | 773.58 | 丄 |
| 1200 | 150.83 | 145.96 | 14.59 | 1438.97 | ⊥ |
| 1400 | 244.31 | 241.59 | 17.48 | 2062.51 | \bot |
| 1600 | 446.23 | 430.48 | 20.32 | 3 3 0 1 . 1 2 | ⊥ |
| 1800 | 970.91 | 957.00 | 23.24 | 丄 | 丄 |
| 2000 | 1 267.27 | 1259.51 | 25.60 | T | ⊥ |
| 2200 | 1963.17 | 1946.01 | 31.72 | \perp | \perp |
| 2400 | 2 2 8 3 . 3 7 | 2 2 7 3 . 0 5 | 36.03 | T | 丄 |
| 2600 | 3 3 0 9 . 3 6 | 3292.33 | 39.89 | 丄 | 丄 |

Table 3: Runtimes (in seconds) of the BFS protocols in the semi-honest unrestricted network setting.

Table 4: Runtimes (in seconds) of the BFS protocols in the semi-honest LAN setting.

| Nodes | Logarithmic | Logarithmic w/o overflow mitigation | Optimized [11] | Linear | Comparison-free |
|-------|-------------|-------------------------------------|----------------|---------|-----------------|
| 10 | 0.07 | 0.07 | 0.53 | 0.20 | 0.28 |
| 20 | 0.08 | 0.07 | 1.27 | 0.43 | 0.86 |
| 30 | 0.09 | 0.08 | 1.92 | 0.69 | 1.69 |
| 40 | 0.11 | 0.09 | 2.95 | 0.99 | 2.81 |
| 50 | 0.12 | 0.11 | 3.71 | 1.33 | 4.26 |
| 60 | 0.14 | 0.12 | 4.47 | 1.71 | 5.93 |
| 70 | 0.18 | 0.15 | 5.86 | 2.20 | 8.07 |
| 80 | 0.21 | 0.17 | 6.75 | 2.72 | 10.50 |
| 90 | 0.24 | 0.20 | 7.64 | 3.30 | 13.32 |
| 100 | 0.27 | 0.23 | 8.47 | 3.93 | 16.88 |
| 200 | 0.90 | 0.78 | 18.89 | 16.24 | 95.77 |
| 400 | 4.61 | 4.27 | 42.12 | 99.18 | 908.04 |
| 600 | 16.06 | 15.00 | 69.49 | 324.83 | 4 205.87 |
| 800 | 43.18 | 40.41 | 93.42 | 741.36 | 丄 |
| 1000 | 71.74 | 71.41 | 117.37 | 1394.57 | 丄 |
| 1200 | 158.57 | 149.92 | 153.24 | 2479.91 | 丄 |
| 1400 | 251.82 | 243.82 | 179.99 | 3700.48 | 丄 |
| 1600 | 454.48 | 434.40 | 207.19 | 5708.77 | \perp |
| 1800 | 967.92 | 959.33 | 234.79 | 丄 | 丄 |
| 2000 | 1285.91 | 1268.25 | 262.31 | 丄 | 丄 |
| 2200 | 1979.29 | 1993.97 | 311.90 | 丄 | 丄 |
| 2400 | 2307.24 | 2286.88 | 341.73 | 丄 | 丄 |
| 2600 | 3361.94 | 3300.99 | 372.20 | 丄 | 丄 |

| Nodes | Logarithmic | Logarithmic w/o overflow mitigation | Optimized [11] | Linear | Comparison-free |
|-------|---------------|-------------------------------------|----------------|---------|-----------------|
| 10 | 0.65 | 0.66 | 5.10 | 1.88 | 2.65 |
| 20 | 0.78 | 0.68 | 12.15 | 4.07 | 8.19 |
| 30 | 0.83 | 0.74 | 18.39 | 6.37 | 16.09 |
| 40 | 0.93 | 0.80 | 28.20 | 8.87 | 26.52 |
| 50 | 1.01 | 0.87 | 35.41 | 11.64 | 39.69 |
| 60 | 1.06 | 0.95 | 42.59 | 14.67 | 54.36 |
| 70 | 1.31 | 1.06 | 56.11 | 18.03 | 72.77 |
| 80 | 1.38 | 1.14 | 64.22 | 21.72 | 92.41 |
| 90 | 1.48 | 1.22 | 72.36 | 25.84 | 114.48 |
| 100 | 1.56 | 1.27 | 80.56 | 30.33 | 141.25 |
| 200 | 3.15 | 2.59 | 180.20 | 110.23 | 546.49 |
| 400 | 11.02 | 9.27 | 399.58 | 638.57 | 2700.68 |
| 600 | 33.07 | 25.76 | 657.35 | 2006.15 | 8 2 6 8 . 5 4 |
| 800 | 73.51 | 59.28 | 880.06 | 4686.42 | ⊥ |
| 1000 | 121.10 | 97.93 | 1 1 0 4 . 9 3 | 9165.68 | ⊥ |
| 1200 | 229.10 | 190.61 | 1441.05 | 丄 | 丄 |
| 1400 | 349.96 | 301.19 | 1687.98 | 丄 | 丄 |
| 1600 | 574.24 | 508.15 | 1940.02 | 丄 | 丄 |
| 1800 | 1 1 2 6 . 3 7 | 1050.20 | 2 191.96 | 丄 | 丄 |
| 2000 | 1471.28 | 1382.14 | 2441.58 | 丄 | 丄 |
| 2200 | 2 2 1 5 . 7 7 | 2095.09 | 2904.16 | T | T |
| 2400 | 2583.31 | 2448.98 | 3 179.02 | 丄 | 丄 |
| 2600 | 3662.68 | 3508.02 | 3452.78 | 工 | \perp |

Table 5: Runtimes (in seconds) of the BFS protocols in the semi-honest WAN setting.

Table 6: Rounds and total data sent (C., in megabyte) of the BFS protocols in the semi-honest setting.

| Nodes | | Logarithmic | Logarithmic with overflows | | Optimized [11] | | Linear | | Comparison-free | |
|-------|--------|------------------|----------------------------|----------------|----------------|------------------|---------|---------|-----------------|----------|
| | Rounds | C. [MB] | Rounds | C. [MB] | Rounds | C. [MB] | Rounds | C. [MB] | Rounds | C. [MB] |
| 10 | 65 | $\boldsymbol{0}$ | 65 | θ | 588 | $\mathbf{0}$ | 184 | 0 | 260 | Ω |
| 20 | 75 | $\boldsymbol{0}$ | 67 | Ω | 1359 | $\boldsymbol{0}$ | 384 | 1 | 800 | 1 |
| 30 | 75 | $\boldsymbol{0}$ | 67 | θ | 2049 | $\mathbf{0}$ | 584 | 5 | 1560 | 3 |
| 40 | 77 | 1 | 69 | 1 | 3090 | $\mathbf{0}$ | 784 | 13 | 2560 | 7 |
| 50 | 77 | \overline{c} | 69 | 1 | 3870 | $\mathbf{1}$ | 984 | 27 | 3800 | 15 |
| 60 | 77 | 3 | 69 | $\overline{2}$ | 4650 | 1 | 1184 | 47 | 5 1 6 0 | 26 |
| 70 | 87 | 4 | 71 | 3 | 6051 | 2 | 1384 | 75 | 6860 | 41 |
| 80 | 87 | 6 | 71 | $\overline{4}$ | 6921 | $\boldsymbol{2}$ | 1584 | 113 | 8640 | 62 |
| 90 | 87 | 7 | 71 | 6 | 7791 | 3 | 1784 | 162 | 10 6 20 | 88 |
| 100 | 87 | 9 | 71 | 7 | 8661 | $\overline{4}$ | 1984 | 223 | 13 000 | 121 |
| 200 | 89 | 40 | 73 | 31 | 19152 | 17 | 4012 | 1801 | 46 800 | 965 |
| 400 | 91 | 169 | 75 | 132 | 41943 | 67 | 8236 | 14486 | 175 200 | 7703 |
| 600 | 117 | 475 | 81 | 309 | 68334 | 151 | 12844 | 48 9 57 | 384 000 | 25971 |
| 800 | 125 | 848 | 81 | 553 | 91 1 34 | 268 | 18024 | 116098 | 丄 | 丄 |
| 1000 | 133 | 1332 | 85 | 872 | 113934 | 417 | 23976 | 226867 | ⊥ | 丄 |
| 1200 | 163 | 2 1 3 0 | 95 | 1290 | 147529 | 601 | 30884 | 392124 | 丄 | ⊥ |
| 1400 | 179 | 2896 | 99 | 1761 | 172129 | 817 | 38 9 44 | 622800 | ⊥ | ⊥ |
| 1600 | 199 | 3794 | 107 | 2308 | 196733 | 1068 | 48344 | 929778 | \perp | \perp |
| 1800 | 219 | 4797 | 115 | 2930 | 221337 | 1351 | ⊥ | 丄 | ⊥ | ⊥ |
| 2000 | 243 | 5934 | 123 | 3632 | 245 937 | 1665 | 丄 | 丄 | 丄 | 丄 |
| 2200 | 273 | 7303 | 137 | 4522 | 290 332 | 2020 | 丄 | 丄 | 丄 | ⊥ |
| 2400 | 305 | 8706 | 149 | 5386 | 316736 | 2399 | 丄 | 丄 | 丄 | 丄 |
| 2600 | 337 | 10 222 | 161 | 6326 | 343 140 | 2819 | ⊥ | ⊥ | ⊥ | \perp |

| Nodes | Logarithmic | Logarithmic w/o overflow mitigation | Optimized [11] | Linear | Comparison-free |
|-------|-------------|-------------------------------------|----------------|---------------|-----------------|
| 10 | 1.50 | 1.53 | 0.11 | 1.52 | 0.02 |
| 20 | 1.57 | 1.53 | 0.30 | 1.64 | 0.10 |
| 30 | 1.68 | 1.56 | 1.83 | 1.82 | 0.28 |
| 40 | 1.82 | 1.79 | 2.17 | 2.25 | 0.57 |
| 50 | 2.86 | 2.75 | 2.35 | 2.94 | 1.06 |
| 60 | 3.82 | 3.84 | 2.57 | 5.42 | 1.69 |
| 70 | 4.99 | 4.92 | 3.32 | 7.45 | 2.66 |
| 80 | 7.07 | 6.81 | 3.50 | 10.60 | 3.95 |
| 90 | 9.65 | 9.50 | 3.91 | 15.37 | 5.92 |
| 100 | 12.00 | 11.86 | 4.16 | 20.86 | 8.41 |
| 150 | 43.45 | 42.72 | 8.42 | 69.36 | 34.29 |
| 200 | 101.66 | 100.54 | 10.65 | 164.03 | 88.33 |
| 250 | 200.60 | 198.89 | 12.92 | 319.91 | 201.58 |
| 300 | 372.57 | 373.03 | 27.11 | 554.61 | 368.53 |
| 350 | 592.53 | 591.65 | 30.88 | 863.90 | 667.89 |
| 400 | 899.89 | 881.67 | 36.26 | 1 2 9 2 . 1 8 | 1051.21 |

Table 7: Runtimes (in seconds) of the BFS protocols in the malicious security unrestricted network setting.

Table 8: Runtimes (in seconds) of the BFS protocols in the malicious security LAN setting.

| Nodes | Logarithmic | Logarithmic w/o overflow mitigation | Optimized [11] | Linear | Comparison-free |
|-------|---------------|-------------------------------------|----------------|---------|-----------------|
| 10 | 2.38 | 2.35 | 1.31 | 2.50 | 0.28 |
| 20 | 2.50 | 2.54 | 3.15 | 2.84 | 0.95 |
| 30 | 2.53 | 2.41 | 6.75 | 3.27 | 2.05 |
| 40 | 2.70 | 2.66 | 9.52 | 3.94 | 3.53 |
| 50 | 3.98 | 3.87 | 11.27 | 4.96 | 5.49 |
| 60 | 5.34 | 5.10 | 13.11 | 8.40 | 7.95 |
| 70 | 6.75 | 6.71 | 17.30 | 10.96 | 11.18 |
| 80 | 9.42 | 9.21 | 19.44 | 15.06 | 15.07 |
| 90 | 12.68 | 12.54 | 21.60 | 21.12 | 19.66 |
| 100 | 15.71 | 15.60 | 23.73 | 28.25 | 25.58 |
| 150 | 56.55 | 55.87 | 42.96 | 89.14 | 67.80 |
| 200 | 130.71 | 129.76 | 56.12 | 208.50 | 151.09 |
| 250 | 259.07 | 257.43 | 69.93 | 400.26 | 298.76 |
| 300 | 477.92 | 474.13 | 107.62 | 694.33 | 521.53 |
| 350 | 762.55 | 762.10 | 124.79 | 1079.18 | 870.05 |
| 400 | 1 1 4 4 . 9 9 | 1 1 38.13 | 143.72 | 1619.51 | 1344.37 |

Table 9: Runtimes (in seconds) of the BFS protocols in the malicious security WAN setting.

Table 10: Rounds and total data sent (C., in megabyte) of the BFS protocols in the malicious security setting.

| Nodes | | Optimized [11] Logarithmic with overflows Linear Logarithmic | | | | Comparison-free | | | | |
|-------|--------|---|--------|---------|---------|-----------------|---------|------------|---------|--------------|
| | Rounds | C. [MB] | Rounds | C. [MB] | Rounds | C. [MB] | Rounds | C. [MB] | Rounds | C. [MB] |
| 10 | 86 | 240 | 86 | 240 | 1168 | 3 | 211 | 240 | 260 | $\mathbf{0}$ |
| 20 | 97 | 243 | 88 | 243 | 2814 | 8 | 421 | 243 | 820 | 5 |
| 30 | 97 | 250 | 88 | 249 | 4 1 5 4 | 247 | 631 | 253 | 1640 | 19 |
| 40 | 99 | 266 | 90 | 265 | 6560 | 260 | 841 | 272 | 2750 | 45 |
| 50 | 104 | 375 | 95 | 374 | 8 1 8 0 | 265 | 1051 | 304 | 4 1 7 0 | 88 |
| 60 | 109 | 494 | 100 | 493 | 9800 | 271 | 1269 | 592 | 5755 | 153 |
| 70 | 125 | 634 | 107 | 631 | 13 2 46 | 318 | 1484 | 736 | 7790 | 243 |
| 80 | 130 | 871 | 112 | 866 | 15 1 26 | 329 | 1702 | 1064 | 9835 | 362 |
| 90 | 145 | 1 1 9 7 | 127 | 1 1 9 2 | 17006 | 341 | 1925 | 1497 | 12415 | 516 |
| 100 | 145 | 1486 | 127 | 1480 | 18886 | 353 | 2151 | 2 1 1 9 | 15495 | 708 |
| 150 | 192 | 5 1 9 0 | 174 | 5 1 7 5 | 32212 | 769 | 3364 | 7044 | 27 400 | 2363 |
| 200 | 197 | 12 16 3 | 179 | 12 13 6 | 42912 | 899 | 4711 | 16559 | 47035 | 5594 |
| 250 | 205 | 24 2 36 | 187 | 24 194 | 53622 | 1111 | 6258 | 32 289 | 71465 | 10953 |
| 300 | 225 | 44 900 | 199 | 44 600 | 72188 | 2600 | 8047 | 55815 | 101545 | 18932 |
| 350 | 225 | 71331 | 207 | 71 250 | 84 208 | 3043 | 10 1 23 | 88639 | 136 265 | 30 1 55 |
| 400 | 233 | 107 283 | 207 | 106 936 | 96228 | 3495 | 12531 | 132 308 | 177 055 | 45 002 |

Table 11: Runtimes (in seconds) of the max flow protocols in the semi-honest unrestricted network setting.

Table 12: Runtimes (in seconds) of the max flow protocols in the semi-honest LAN setting.

Table 13: Runtimes (in seconds) of the max flow protocols in the semi-honest WAN setting.

| Nodes | | Edmonds-Karp Capacity-Scaling (63 bits) Capacity-Scaling (32 bits) Capacity-Scaling (16 bits) Dinic-Tarjan | | | |
|-------|---------|--|---------------|-------------|---------------|
| | 157.57 | 1573.81 | 799.42 | 400.75 | 28.26 |
| 10 | 1477.63 | 8367.28 | 4 2 5 1 . 2 0 | 2 1 3 6 .98 | 278.29 |
| | 5317.61 | 20855.06 | 10 591.23 | 5 300.05 | 1 0 1 6 . 4 1 |

Table 14: Rounds and total data sent (C., in megabyte) of the max flow protocols in the semi-honest setting.

Table 15: Runtimes (in seconds) of the max flow protocols in the malicious security unrestricted network setting.

Table 16: Runtimes (in seconds) of the max flow protocols in the malicious security LAN setting.

Table 17: Runtimes (in seconds) of the max flow protocols in the malicious security WAN setting.

Table 18: Rounds and total data sent (C., in megabyte) of the max flow protocols in the malicious security setting.

| Nodes | Edmonds-Karp | | Capacity-Scaling (63 bits) | | Capacity-Scaling (32 bits) | | Capacity-Scaling (16 bits) | | Dinic-Tarjan | |
|-------|---------------|---------|----------------------------|-----------|----------------------------|---------|----------------------------|-----------|--------------|---------|
| | Rounds | [MB] | Rounds | C. [MB] | Rounds | C. [MB] | Rounds | $C.$ [MB] | Rounds | C. [MB] |
| | 16397 | 314 | 165 096 | 689 | 83870 | 503 | 41 949 | 448 | 3017 | 244 |
| 10 | 155 103 | 2559 | 879368 | 13544 | 446 676 | 7040 | 223 348 | 3640 | 29 4 4 0 | 647 |
| 15 | 557847 | 24 6 51 | 2 186 742 | 96 283 | 110741 | 49 159 | 555 374 | 24 5 40 | 106 363 | 4 3 5 1 |
| 20 | 1582062 | 151 474 | 4 7 3 4 3 0 4 | 452814 | 2404739 | 230 201 | 1202381 | 115 258 | 261 057 | 18090 |
| 25 | 3 2 5 7 0 5 4 | 561714 | 7879462 | 1 358 763 | 4 002 284 | 690432 | 2001148 | 345 216 | 520 901 | 55 5 29 |