Going Faster: Privacy-Preserving Shortest Paths from Start to End (Extended Version)*

Vincent Ehrmanntraut
RWTH Aachen University
Aachen, Germany
ehrmanntraut@itsec.rwth-aachen.de

Ulrike Meyer
RWTH Aachen University
Aachen, Germany
meyer@itsec.rwth-aachen.de

Abstract

Finding shortest paths in graphs is a fundamental combinatorial optimization problem with numerous applications, e. g., in graph centrality measures, computational biology, and business process analysis. To address privacy constraints in such applications, Secure Multi-Party Computation (SMPC) protocols can compute the lengths of shortest paths on secret-shared graphs. Many such protocols have been proposed in the past, but they only compute the lengths of the shortest paths and not the paths themselves.

In this paper, we close this gap by proposing two SMPC protocols that output secret-shared shortest paths. The first composes sub-protocols to calculate predecessors, to follow them, and to reorder wrongly positioned padding. Our second protocol optimizes the predecessor-following by decomposing it into sub-functionalities and proposing new sub-protocols for each. Further, the padding-reorder is replaced by a graph transformation.

We evaluate our protocols across multiple network and security settings. Our optimized predecessor-following is up to 23 times faster than prior art, and our graph transformation is orders of magnitude faster than the padding-reorder. Overall, our second protocol is 5% to 49% faster than the first protocol due to the predecessor-calculation dominating the runtime on large graphs.

Finally, we introduce a novel demultiplex protocol that, in fast networks, outperforms prior protocols on vector sizes up to 500, and on all evaluated vector sizes in slow networks. Due to the ubiquity of demultiplexes, this protocol is of independent interest.

CCS Concepts

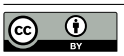
• **Security and privacy → Privacy-preserving protocols**.

Keywords

secure multi-party computation; shortest path

ACM Reference Format:
Vincent Ehrmanntraut and Ulrike Meyer. 2026. Going Faster: Privacy-Preserving Shortest Paths from Start to End (Extended Version). In *25th Workshop on Privacy in the Electronic Society (WPES '26), November 15–19, 2026, The Hague, Netherlands.* ACM, New York, NY, USA, 42 pages. https://doi.org/10.1145/3847192.3847373

*This is the extended version of the paper published at WPES '26 [31].



This work is licensed under a Creative Commons Attribution 4.0 International License.
WPES '26, The Hague, Netherlands
© 2026 Copyright held by the owner/author(s).
ACM ISBN 979-8-4007-3026-9/2026/11
https://doi.org/10.1145/3847192.3847373

1 Introduction

The shortest path problem consists of finding a path of minimal length between pairs of nodes in a weighted graph. It is a fundamental problem in computer science with various applications across fields. For example, shortest paths are used in graph centrality measures [35], in the analysis of metabolic connection networks during drug discovery [57], in the alignment of genomic sequences [39] in computational biology, and for computing differences between process models and real processes [2] in business process analysis. Beyond these direct applications, shortest paths also serve as a building block for further algorithms. For example, [19] uses shortest paths to compute min-cost flows, which in turn can be used to solve assignment problems. All the application areas naturally lead to applications of the shortest path problem with privacy constraints, e. g., collaborative biological analysis with medically sensitive or with proprietary data, or cross-organizational process analysis. Indirect applications with well-documented privacy constraints are air traffic flow slot assignment [52], the kidney donor exchange problem [17], and bartering [63].

Such privacy constraints may be addressed using Secure Multi-Party Computation (SMPC) protocols. [3, 4, 6, 8, 44, 51, 56] propose protocols that compute the length of the shortest paths between a single node and all other nodes in secret-shared graphs. [6] also proposes a protocol that computes the lengths of the shortest paths between all pairs of nodes. While these protocols call themselves single-source shortest-path protocols or all-pairs shortest-path protocols, they only compute the length of the shortest paths. None of the referenced protocols produces the path between nodes itself. Thus, they cannot be directly applied to the mentioned applications as they require the shortest paths and not just their lengths.

In this work, we close this gap by presenting two single-pair shortest-path protocols. They consume a secret-shared graph and secret-shared start and target nodes, and produce a secret-shared path from the start to the target node. Like the prior work on (length of) shortest path protocols, we assume the secret-shared graph to be created by a surrounding application-specific SMPC protocol that also uses the secret-shared path. Only the number of nodes in the graph needs to be public. All other information about the graph and the path, including the number of edges, the length of edges, or the number of traversed nodes, remains private.

Our first protocol composes three sub-functionalities that (1) calculate predecessors, i. e., pointers towards the previous node on the path, (2) follow those predecessors, and (3) move the padding that hides the number of traversed nodes to the correct position. This is already non-trivial, as the number of steps in the path must remain secret. We address (1) by modifying two prior (length of) shortest-path protocols such that they produce predecessors and in

passing fix a leakage found in one of them [6]. (2) is addressed by modifying a part of Blanton et al. [14]'s max-flow protocol, and (3) using a single radix sort iteration.

Our second protocol optimizes the first in two ways. It first decomposes the predecessor-following (2) into a demultiplex and a path-extraction sub-functionality. For each of these, multiple interchangeable sub-protocols are presented, among them a novel demultiplex and a novel path-extraction protocol. We hypothesize that no sub-protocol outperforms its alternatives across all graph sizes and computation environments. Therefore, our second protocol adaptively selects sub-protocols depending on the graph size and computation environment. It then eliminates (3) using a graph transformation that tricks the predecessor-following into creating the padding at the correct position. All protocols are proven secure in the Arithmetic Black Box (ABB)-model and thus can be instantiated using any universally composable secret-sharing SMPC primitive. This allows using our protocols with any number of parties and with any adversary model.

We evaluate our protocols in a three-party setting and with both semi-honest and malicious security on graphs with up to 1000 nodes. Taken individually, each optimization is effective. Given the predecessors, our optimized predecessor-following is 2.7 to 23 times faster than Blanton et al. [14]'s predecessor-following protocol in slow networks, and 1.4 to 15 times faster in fast networks. Our graph transformation is up to three orders of magnitude faster than the sorting-based padding-reorder. Overall, our second single-pair shortest-path protocol is 5% to 49% faster than the first protocol, with the smaller gains on large graphs where the predecessor-calculation dominates the runtime.

Finally, our novel demultiplex protocol is faster than prior approaches up to vector size 500 in fast networks, and on all evaluated vector sizes in slow networks, both with semi-honest security. This is of independent interest for other protocols using demultiplexes, including oblivious data structures [62] and softmax protocols [45] used in privacy-preserving machine-learning.

To summarize, our contributions are as follows:

- We present the, to the best of our knowledge, first single-pair shortest-path protocol that produces a path instead of lengths. This includes identifying and fixing a leakage in a prior (length of) shortest-path protocol [6].
- We optimize our protocol's predecessor-following step by decomposing it into demultiplex and path-extraction sub-functionalities, and presenting novel sub-protocols for each sub-functionality. Predecessor-following and path-extraction are of potential independent interest for SMPC protocols that internally construct paths. As the demultiplex is a common building block of SMPC protocols, our new demultiplex protocol and our evaluation of prior protocols are of independent interest as well.
- We further optimize the single-pair shortest-path protocol by introducing a graph transformation trick to efficiently ensure correctly positioned padding.
- We evaluate our protocols in multiple network settings and with semi-honest and malicious security, showing that our optimizations individually are effective, but that the predecessor-calculation dilutes the effect of the optimizations on the runtime of the overall single-pair shortest-path protocols.

2 Preliminaries

This section briefly introduces used notation in Sec. 2.1, followed by a quick overview of SMPC in Sec. 2.2 and of relevant definitions from graph theory in Sec. 2.3.

2.1 Notation

Secret-shared values are denoted by square brackets, e. g., $[x]$ is a secret-sharing of value x such that each party holds one share.

Matrices are capitalized and bold, with $\mathbf{M}_{i,j}$ being the element in the i'th row and j'th column of matrix $\mathbf{M}$. The whole i'th row is written as $\mathbf{M}_{i,\bullet}$, and the j'th column as $\mathbf{M}_{\bullet,j}$. A vector with k elements is a $1 \times k$ matrix, i. e., a matrix with a single row. For better readability, vectors are written with an arrow and are capitalized, $\overrightarrow{V}_i$ is the i'th entry of the vector $\overrightarrow{V}$. The notation is analogous for secret-shared matrices and vectors.

When a vector is shuffled using the permutation σ, $\sigma(i)$ is the new position of the element at the old position i. A matrix is shuffled by first shuffling each row individually with the same permutation, and then shuffling each column individually, again using the same permutation. This matrix shuffling procedure corresponds to shuffling the nodes of a graph.

Finally, we use $[c]\ ?\ [a]\ :\ [b]$ as shorthand notation for $[b] + [c] \cdot ([a] - [b])$, which results in $[a]$ if $c = 1$ and $[b]$ if $c = 0$.

2.2 Secure Multi-Party Computation

Classically, an SMPC protocol allows multiple parties to compute a function on the private input data in a distributed fashion such that each party learns its output of the function but nothing about the other parties' inputs beyond what can be deduced from their own input and output.

Such protocols can for example be designed using secret-sharing techniques. Each party secret-shares its own private input and distributes the shares to the computing parties, and the function is then computed on these shares. Once the computation finishes, the reconstruction of the computed shares yields the desired output. The computation of secret-shared outputs from secret-shared inputs can also be used as a building block in another larger protocol and can thus be of independent interest.

In this work, we present a building-block protocol that computes shortest paths, i. e., the protocol consumes a secret-shared graph and produces a secret-shared shortest path, while simultaneously using building-block protocols itself. A particular class of used building-blocks are the so-called SMPC primitives that implement elementary operations such as additions and multiplications.

We present our protocol as a sequence of the ABB operations presented in Fig. 1. These operations work on secret-shared values and can be realized using any universally composable SMPC primitive based on linear secret-sharings over $\mathbb{F}_p$ or $\mathbb{Z}_{2^k}$ for sufficiently large primes p or integers k, such as [9, 26, 28, 43].

Security model. Such primitives exist for different security models, commonly distinguished by the fraction of corrupted parties (honest vs. dishonest majority), and corrupted parties' behavior (semi-honest vs. malicious). The semi-honest security model (also known as passive security) assumes that corrupted parties follow

- Input (i):
  Yields secret-sharing $[v]$ of the value v which is provided by party i. This operation is not used by our protocols, and is only included for completeness.
- PublicInput (v):
  Yields secret-sharing $[v]$ of the value v.
  Shorthand notation: $[v]$ for integer v, e. g., $[4]$
- Reveal ($[v]$):
  Reveals the secret-shared value v to all parties.
- Add ($[v_1]$, $[v_2]$):
  Calculates $v_3 = v_1 + v_2$ and yields new secret-sharing $[v_3]$.
  Shorthand notation: $[v_3] \leftarrow [v_1] + [v_2]$
- Multiply ($[v_1]$, $[v_2]$):
  Calculates $r = v_1 \cdot v_2$ and yields new secret-sharing $[r]$.
  Shorthand notation: $[r] \leftarrow [v_1] \cdot [v_2]$
- MultiplyMatrices($[\mathbf{A}_{1,1}], \ldots, [\mathbf{A}_{m,l}], [\mathbf{B}_{1,1}], \ldots, [\mathbf{B}_{l,n}]$):
  Computes the matrix-product $\mathbf{C} = \mathbf{A} \cdot \mathbf{B}$ and yields new secret-sharings $[\mathbf{C}_{1,1}], \ldots, [\mathbf{C}_{m,n}]$.
  Shorthand notations: $[\mathbf{C}] \leftarrow [\mathbf{A}] \cdot [\mathbf{B}]$, and $[c] \leftarrow [\overrightarrow{A}] \cdot [\overrightarrow{B}^{\mathsf{T}}]$ ($\overrightarrow{A}$ and $\overrightarrow{B}$ are $1 \times l$ matrices)
- LessThan ($[v_1]$, $[v_2]$) / Equals ($[v_1]$, $[v_2]$):
  Yields new secret-sharing $[1]$ if $v_1 < v_2$ / $v_1 = v_2$, or new secret-sharing $[0]$ if $v_1 \geq v_2$ / $v_1 \neq v_2$.
  Shorthand notation: $[r] \leftarrow [v_1] < [v_2]$ / $[v_1] = [v_2]$
- BitDecompose ($[v], k$):
  Yields secret-sharings $[b_1], \ldots, [b_k]$ of the k least significant bits of v.
- GeneratePermutation (k):
  Generates a secret uniformly random permutation of $1, \ldots, k$, and yields the secret-shared permutation $[\sigma]$.
- Shuffle / Unshuffle ($[v_1], \ldots, [v_k], [\sigma]$):
  Shuffles the vector $[v_1], \ldots, [v_k]$ using the permutation σ / σ^{-1}, and yields new secret-sharings of the shuffled vector.

Figure 1: Operations supported by the ABB considered in this work.

the protocol, while they may deviate arbitrarily in the malicious security model (also known as active security).

As our protocol consists of a sequence of ABB operations, we prove the security of our protocol in the ABB-hybrid-model, i. e., under the assumption that the ABB is realized securely. Concretely, we show that the sequence of operations and revealed secret-shared values do not leak anything about the secret-shared inputs and outputs of the protocol beyond what can be deduced from the protocol's public input[1]. Due to the universal composability of the ABB, this is sufficient for our protocol to achieve the same level of security as the SMPC primitive used to realize the operations [49].

Complexity metrics. We consider the round, communication, and computation complexity of protocols. The round complexity is determined by the number of sequential communication steps. The communication complexity reflects the total amount of sent data, and the computation complexity the number of computation steps. As our protocols' sequences of operations do not depend on the number of parties, nor on the bit size of secret-shares, we ignore these factors in our complexities.

All three metrics influence protocol runtimes, but their relative weight depends on factors such as the network's delay and bandwidth, the parties' computational power, and the used SMPC primitive. For example, high network delays emphasize the round complexity, while low bandwidths emphasize the communication complexity. Collectively, we call these factors (other than protocol input sizes) the computation environment.

Used operations and primitives. The used ABB operations are presented in Fig. 1. Except for the Reveal operation, all operations yield new secret-sharings of the operations' results.

We optimize our protocol for linearly secret-sharing SMPC primitives, and thus assume that additions are performed without any communication between parties, while multiplying and revealing secret-shared values requires $O(1)$ rounds and $O(1)$ communication. The other operations can generally be built from the Input, Reveal, Add, and Multiply operations: The matrix multiplication of an $n \times l$ matrix by an $l \times m$ matrix is trivially achieved in $O(1)$ rounds using $O(nlm)$ parallel multiplications, while comparison and bit-decomposition are achieved in $O(1)$ rounds and communication [23, 27]. Finally, secure permutations of length-k vectors are achievable in $O(\log k)$ rounds and $O(k \log k)$ communication [44].

In our evaluation, we use MP-SPDZ [41]'s implementation of Araki et al. [9]'s semi-honest security SMPC primitive, and its malicious security variant obtained by applying [1]. These implementations apply techniques from [25] to reduce the communication of matrix multiplications to $O(nm)$[2], and from [11] to securely permute a length-k vector using $O(k)$ communication in $O(1)$ rounds. The semi-honest security implementation uses the bit-decomposition and comparison protocols from [54].

2.3 Graph theory

A weighted graph $G = (V, E, l)$ consists of a set of nodes V, a set of edges $E \subseteq V \times V$ connecting some of these nodes, and an edge length function $l : V \times V \mapsto \mathbb{Z} \cup \{\infty\}$, where ∞ is a large constant in practice. The edge length function maps edges $(u, v) \in E$ to their length, and non-edges $(u, v) \notin E$ to ∞.

We assume, without loss of generality, that $V = \{1, \ldots, n\}$, where $n = |V|$ is the number of nodes. This allows presenting the edge lengths as a matrix $\mathbf{L}$ with $\mathbf{L}_{i,j} = l(i, j)$. Further, this allows encoding nodes as vectors, where a node $v \in V$ is mapped to a vector of length n containing only zeroes except a single one at position v. These representations of the nodes and edge lengths are useful to express some graph operations as vector-matrix products.

[1] In this work, the protocol's public input is the number of nodes. Further, malicious security requires validation of values provided through Input commands [49], which we do not use in our protocols. We therefore only consider the revealed values.

[2] We refer to [18, Sec. 1] for a good explanation of the technique.

Finally, a path from $s \in V$ to $t \in V$ is a sequence of nodes $p = (v_1, \ldots, v_k)$ such that $v_1 = s, v_k = t$, and $(v_i, v_{i+1}) \in E$ for all $1 \leq i < k$. Its number of steps equals the number of traversed edges, i. e., $k - 1$, and its length is the sum of all edge lengths, i. e., $l(p) = \sum_{i=1}^{k-1} l(v_i, v_{i+1})$. A shortest path from s to t is a path whose length is minimal, i. e., there is no shorter path from s to t. There might be multiple shortest paths of equal length from s to t. The problem of finding such paths from a starting node to a target node is called the single-pair shortest path problem, which is only well-defined when the graph contains no cycles of negative total length.

3 Related work

Prior privacy-preserving (length of) shortest path protocols are presented in Sec. 3.1, followed by an overview of non-privacy-preserving algorithms in Sec. 3.2. Finally, Sec. 3.3 presents prior demultiplex protocols and Sec. 3.4 an orthogonal approach to optimizing SMPC protocols.

3.1 Privacy-preserving shortest paths

Multiple SMPC protocols that securely compute the length of shortest paths have been proposed. Table 1 presents an overview of these protocols. Generally, they can be divided into protocols that operate on sparse graphs, where the number of edges m is in the order of the number of nodes n, and protocols that operate on dense graphs, where the number of edges is close to n^2.

Protocols on sparse graphs mostly adapt Dijkstra's algorithm using Oblivious Random Access Memories (ORAMs) to hide memory access patterns [6, 44, 51, 56]. The current state-of-the-art of this research direction exploits the fact that each node is visited only once to achieve a protocol requiring $O(n \log n \log\log\log n)$ rounds [56]. However, the protocol only computes the length of shortest paths, and does not yield the paths themselves.

Alternatively, frameworks like [10, 40, 48, 55] adapt message-passing algorithms into privacy-preserving SMPC protocols operating on sparse graphs. While [10, 55] support the Bellman-Ford algorithm, we are not aware of any work using such frameworks to compute shortest paths or even the length of shortest paths.

On dense graphs, Aly and Van Vyve [4] first introduced a protocol inspired by Dijkstra's algorithm which hides memory access patterns by accessing all possible edges in each iteration. Even though some optimizations reduce the search space in later iterations, this leads to a high overhead. Aly and Cleemput [3] reduce this overhead by shuffling the nodes. This allows revealing the shuffled index of the unvisited node closest to the source, enabling efficient updates of the neighboring nodes' lengths. Anagreh et al. [6] further optimize this approach by using a more efficient method to find the closest unvisited node. While the fundamental approach can be realized securely, both protocols suffer from a small leakage. Sec. 4.1 further discusses the leakage and its prevention. Further, these protocols do not return the shortest paths. Instead, the protocol by Anagreh et al. [6] only computes the lengths of shortest paths. Aly and Van Vyve [4] and Aly and Cleemput [3] also include the predecessor nodes. As shown in Sec. 5, constructing a path from these predecessor nodes is not trivial.

Anagreh et al. [6] also adapt other shortest path algorithms to privacy-preserving protocols, including Floyd-Warshall [34] and min-addition [13] for dense graphs, and Bellman-Ford [12] for sparse graphs. On dense graphs, Anagreh et al. [6] find that the Floyd-Warshall protocol is the fastest on small graphs, and that the Dijkstra protocol is faster on larger graphs. The cross-over point depends on the network [6]. Further, these protocols also only compute the lengths of shortest paths.

To the best of our knowledge, the only two SMPC protocols [14, 30] that create paths are not general shortest path protocols, but Breadth-First-Search (BFS) protocols, which solve the shortest path problem only for the special case where all edges have length 1. Blanton et al. [14] use the predecessors from their BFS protocol to begin at the target node and iteratively step towards the start node. Fake nodes are added to hide the path's number of steps. Sec. 5.1 builds a baseline protocol upon this approach.

Ehrmanntraut and Meyer [30] follow a different approach: Their BFS encodes the predecessor information as a matrix, allowing them to use the square-and-multiply algorithm to step from the target node to the start node in $O(\log n)$ rounds. In Sec. 5.2, we re-use this as one instantiation of a sub-functionality.

Finally, differentiation from Graph Encryption Schemes, such as [32, 36, 53, 60], is needed. Such schemes operate in a fundamentally different setting where one party has full knowledge of the graph, allowing pre-computations of (partial) shortest paths and building an index. This index is encrypted and uploaded to an untrusted cloud, allowing the graph owner to later securely query shortest paths. In comparison, the protocol presented in this paper does not require any party to have any knowledge about the graph beyond the number of nodes.

3.2 Non-privacy-preserving shortest paths

Whilst Dijkstra's algorithm [29] has been shown to be an optimal sequential algorithm on general graphs [21], efficient parallel shortest path algorithms have proven themselves to be much more challenging to construct. Algorithms such as [46, 61] were able to reduce the time required to compute shortest paths, but at the cost of significant overheads in the total required work. Only recently, two independent breakthroughs [21, 59] managed to overcome these overheads by leveraging parallel approximative shortest path algorithms and iteratively improving the approximation quality until an exact solution can be determined.

However, these algorithms are not suitable as a basis for SMPC protocols for two reasons: Parallel approximative shortest path algorithms like [22, 58] dynamically create and join threads, making data-oblivious adaptations inefficient. Further, reductions of exact to approximative shortest paths require non-integer arithmetic that is able to accurately represent values like $\ln\left(\frac{3}{2}\right)$. This adds another significant overhead to potential SMPC implementations [38].

3.3 Demultiplexing

A demultiplex converts a numerical index x into the unit vector $\overrightarrow{E^{x,m}}$ of length m, i. e., a vector that only contains zeroes except a single one at position x [42, 50]. It is a common building block of SMPC protocols and has, for example, been used to realize oblivious data structures [44], in privacy-preserving kidney donor exchanges [16],

Table 1: Chronologically ordered comparison of (length of) shortest path protocols. Complexities ignore polynomial dependencies on the number of parties and field size.

Protocol	Base Algorithm	Returns Path	Rounds	Communication	Computation
Blanton et al. [14] •	BFS	●	$O\left(n^2+S\right)^\star$	$O\left(n^2+S\right)^\star$	$O\left(n^2+S\right)^\star$
Keller and Scholl [44] △	Dijkstra	○	$O\left(n\log^3 n + m\log^4 n\right)$	$O\left(n\log^3 n + m\log^4 n\right)$	$O\left(n\log^3 n + m\log^4 n\right)$
Aly and Van Vyve [4]	Dijkstra	◐	$O\left(n^3\right)$	$O\left(n^5\right)$	$O\left(n^5\right)$
Liu et al. [51] △	Dijkstra	○	$O\left((n+m)\log^2 n\right)$	$O\left((n+m)\log^2 n\right)$	$O\left((n+m)\log^2 n\right)$
Aly and Cleemput [3] †	Dijkstra	◐	$O\left(n^2+S\right)^\star$	$O\left(n^2\log n+S\right)^\star$	$O\left(n^2\log n+S\right)^\star$
Anagreh et al. [6] †	Dijkstra	○	$O\left(n\log n+S\right)^\star$	$O\left(n^2+S\right)^\star$	$O\left(n^2+S\right)^\star$
Anagreh et al. [6]	Floyd-Warshall	○	$O\left(n\right)$	$O\left(n^3\right)$	$O\left(n^3\right)$
Anagreh et al. [6]	Min-Addition	○	$O\left(\log^2 n\right)$	$O\left(n^3\log n\right)$	$O\left(n^3\log n\right)$
Anagreh et al. [6] △	Bellman-Ford	○	$O\left(n\log n\right)$	$O\left(nm\right)$	$O\left(nm\right)$
Anagreh et al. [7] ‡	Dijkstra	○	$O\left(n\log n\right)$	$O\left(n^2\right)$ / $O\left(n^3\right)^\diamond$	$O\left(n^3\right)$
Ostrovsky [56] △	Dijkstra	○	$O\left(n\log n\log\log\log n\right)$	$O\left((n+m)\log n\right)$	$O\left((n+m)\log n\right)$
Ehrmanntraut and Meyer [30] •	BFS	●	$O\left(\log n\right)$	$O\left(n^2\log n\right)$ / $O\left(n^3\log n\right)^\diamond$	$O\left(n^3\log n\right)$
This work ⊞	Varies	●	$O\left(n\right)$ to $O\left(n\log n\right)$	$O\left(n^2\right)$ to $O\left(n^3\log n\right)$	$O\left(n^2\right)$ to $O\left(n^3\log n\right)$

● Returns path from start node to target node ◐ Returns predecessors ○ Only computes lengths

• Protocol does not find shortest paths in general but only performs a Breadth-First-Search, i. e., all edges have length 1.

△ Protocol designed for sparse graphs. m denotes the number of edges.

* S denotes the effort required to shuffle the edge-length matrix and depends on the SMPC primitive. At most $O\left(\log n\right)$ rounds, $O\left(n^2\log n\right)$ comm. and comp.

$^\diamond$ The smaller communication complexity requires the primitive supporting $n \times n$ matrix multiplications with $O\left(n^2\right)$ communication, such as [1, 9, 28].

† Protocol is affected by the leakage described in Sec. 4.1 and App. A.

‡ Protocol is not data-oblivious and always leaks the number of reachable nodes.

⊞ Our protocol adapts sub-protocols to the computation's environment, and hence has different characteristics in different environments.

and in a recent protocol that evaluates softmax layers in the context of privacy-preserving machine learning [45].

Launchbury et al. [50] first introduced a demultiplex protocol that first determines the bit-decomposition of the target index x once, and then recursively demultiplexes the first half and the second half of the bits. This results in two unit vectors of size $\sqrt{m}$, which are then combined into a length-m unit vector. Overall, this approach requires $O\left(\log\log m\right)$ communication rounds and $O\left(m\right)$ communication and computation.

Keller et al. [42] introduced an alternative sub-protocol that also first decomposes the target index x into its bits, but then processes these bits one bit at a time. This approach is optimized towards primitives operating on large fields by packing multiple bits into a single field element. This significantly reduces the communication cost of this protocol, which is $O\left(m\right)$ but with a small hidden constant, at the expense of the higher round complexity of $O\left(\log m\right)$.

Finally, multiple works [14, 15] have implicitly used the trivial approach to demultiplexing consisting of m parallel comparisons.

3.4 Hybrid compilers

In this work, we evaluate different protocols for realizing sub-functionalities like demultiplexes, and switch between sub-protocols depending on the best approach for any given input size and computation environment. This is similar but orthogonal to hybrid compilers, which enable switching between SMPC primitives with the goal of using the fastest primitive for each basic operation of the evaluated protocol [20, 24, 33]. In comparison, this paper assumes a fixed underlying primitive, and instead adjusts the protocol to the input size and computation environment, i. e., our optimizations are made at a higher level. It is theoretically possible to integrate our high-level optimizations with low-level compiler optimizations.

4 Building blocks

This section presents our contributions with regard to building blocks of our single-pair shortest path protocols. Sec. 4.1 presents a leakage in prior (length of) shortest path protocols and how we fix it. The fixed protocol is used in both of our single-pair shortest-path protocols. Afterward, Sec. 4.2 presents a new demultiplex protocol to be used by our optimized shortest-path protocol.

4.1 Fixing a leakage in prior protocols

In this section, we briefly describe a subtle leakage in prior shortest path protocols based on the Dijkstra algorithm, namely the protocols by Aly and Cleemput [3] and by Anagreh et al. [6]. As described in Sec. 3.1, both protocols first shuffle the nodes in an initialization phase. In each iteration, they then find the unvisited node closest to the start node and reveal its shuffled index, which allows efficient updates of the neighbors' lengths.

It is claimed that the revealed indices do not reveal any private information due to the shuffling of the nodes [3, 6]. In a follow-up paper, Anagreh and Laud [5] specify that the revealed indices can be simulated using a random permutation of $1, \ldots, n$. However, when multiple nodes have the same length to the start node, the closest unvisited node with the lowest shuffled index is chosen, as explained in detail in App. A. As a result, the revealed indices always increase when the length to the start node does not increase. It follows that, on graphs where multiple nodes have the same length to the start node, the revealed (shuffled) node indices are not uniformly random permutations of $1, \ldots, n$. This breaks the simulator argument by Anagreh and Laud [5]. Further, some bounds on the number of layers in the graph and the size of each layer can be inferred.

This leakage can be fixed in multiple ways, e. g., by choosing the node to visit uniformly at random from the set of all nodes with

the equal smallest length to the start node. The random selection, however, would incur a significant overhead on the protocol's runtime. Instead, App. A formally proves that it is sufficient to select the node with the lowest unshuffled index when there are multiple nodes with the same length to the start node, and presents Prot. 9, an efficient implementation of this approach. This implementation will be further modified by Prot. 3 to produce predecessors.

4.2 A new demultiplex protocol

Recall from Sec. 3.3 that demultiplex protocols convert a secret-shared index $[x]$ into the unit vector $[\overrightarrow{E^{x,m}}]$. We introduce a novel demultiplex protocol, called $\Pi^{\text{PreprocessReveal}}_{\text{Demux}}$ (Prot. 1), that first generates a uniformly random unit vector by shuffling an arbitrary unit vector. It then reveals the difference between the wanted index x and the random unit vector's index, and rotates the random vector accordingly. App. B.1 proves that this does not leak anything about x as the offset calculation effectively forms a one-time pad.

Protocol 1 $\Pi^{\text{PreprocessReveal}}_{\text{Demux}}$ $([x], m)$

1: $l \leftarrow 2^{\lceil \log_2 m \rceil}$
2: $[\overrightarrow{Pre}] \leftarrow [\overrightarrow{E^{1,l}}]$; Shuffle $[\overrightarrow{Pre}]$
3: $offset \leftarrow \text{Reveal}\left(\left([x] - \sum_{i=1}^{l} i \cdot [\overrightarrow{Pre}_i]\right) \bmod l\right)$
4: **for** $1 \le i \le m$ **do**
5: $\quad [\overrightarrow{Result}_i] \leftarrow [\overrightarrow{Pre}_{i-offset-1 \bmod l}]$
6: **return** $[\overrightarrow{Result}]$

For this approach to be efficient in practice, the modulo operation in Line 3 of Prot. 1 must be performed quickly. We notice that revealing secret-shared values modulo a power of two often is faster than modulo an arbitrary modulus[3], and therefore increase the size of the pregenerated random unit vector to the next power of two. Using the primitives used in our evaluation, $\Pi^{\text{PreprocessReveal}}_{\text{Demux}}$ requires $O(1)$ rounds and $O(m)$ communication and computation. Other primitives might not shuffle as efficiently, leading to $O(\log m)$ rounds and $O(m \log m)$ communication and computation.

5 Privacy-preserving single-pair shortest-paths

In this section, we present two protocols that realize the single-pair shortest-path functionality presented in Func. 1. These protocols consume the number of nodes n as public input, and a graph's edge-length matrix as well as the start and target node as secret-shared inputs. Their single secret-shared output is a matrix that encodes a shortest path from the start to the target node where the i'th row of this matrix corresponds to the i'th visited node[4]. Any other information about the graph or the path, e. g., the number of edges or the length of the path, must not be leaked. Note that Func. 1 only requires the resulting path to be a shortest path. This means that our protocols can freely choose any shortest path when multiple shortest paths from s to t exist.

[3]Primitives over $\mathbb{Z}_{2^k}$ can use $\text{Reveal}([x] \bmod 2^l) = \text{Reveal}(2^{k-l}[x]) \gg (k-l)$.
[4]This encoding can be transformed into either the path's adjacency matrix using a matrix multiplication, or into an array of numerical indices using local computations.

$\mathcal{F}_{\text{SPSP}}$

Input known to all parties: The number of nodes n.
Secret-shared inputs:
A $n \times n$ edge-length matrix $[\mathbf{L}]$, and
a unit vector $[\overrightarrow{E^{s,n}}]$ of length n encoding the start node s, and
a unit vector $[\overrightarrow{E^{t,n}}]$ of length n encoding the target node t.

Secret-shared output:
A $n \times n$ step matrix $[\mathbf{S^p}]$ that encodes a shortest path $p = (v_1, \ldots, v_k)$ from s to t in the following way: For all $1 \le i \le k$, $\mathbf{S^p}_{i,\bullet} = \overrightarrow{E^{v_i,n}}$, and $\mathbf{S^p}_{i,\bullet} = 0^n$ for all $k < i \le n$. If no path from s to t exists, all entries of $\mathbf{S^p}$ are zero.

Functionality 1: The single-pair shortest path functionality.

To the best of our knowledge, this functionality has not been implemented before. We close that gap with an initial protocol presented in Sec. 5.1. It first computes the path's predecessors using modified prior (length of) shortest-path protocols. These predecessors are then followed, and the padding is reordered. This is already non-trivial, as the number of steps in the path must remain secret.

Sec. 5.2 then decomposes the predecessor-following into independent sub-functionalities, and presents different sub-protocols for each sub-functionality. As these sub-protocols differ in their round, communication, and computation costs, no single choice is fastest across all graph sizes and computation environments. Finally, Sec. 5.3 uses graph-theoretic trickery to eliminate our baseline's padding reorder step.

5.1 Our baseline protocol

Our baseline protocol builds the shortest path in the three distinct steps presented by Fig. 2: First, the predecessors along the path are determined. These predecessors point from each node towards the previous node on the shortest path. In the example input graph, the predecessor of node 5 is node 3 while the predecessor of node 3 is node 2. The second step then begins at the target node and follows these predecessors until the starting node has been reached, resulting in what we call the "almost step matrix". This matrix is similar to the wanted step matrix, but the padding is at the wrong position. The final padding-reorder step fixes this.

Step 1: Computing predecessors. While prior (length of) shortest-path protocols that produce predecessors exist [3, 4], the current state-of-the-art (length of) shortest-path protocols on dense graphs lack this capability. Therefore, we extend Anagreh et al. [6]'s Floyd-Warshall and Dijkstra protocols to also compute predecessors. Anagreh et al. [6] observe that the Floyd-Warshall protocol is faster on small graphs but slower on large ones, so our baseline uses whichever of the two computes the predecessors faster[5].

Modifying the Floyd-Warshall protocol is straightforward as a well-known technique to compute predecessors in the classical (non-privacy-preserving) setting can simply be applied, resulting

[5]As $\Pi^{\text{Dijkstra}}_{\text{PredCalc}}$ requires non-negative edge-lengths, only $\Pi^{\text{FloydWarshall}}_{\text{PredCalc}}$ is applicable on graphs that may contain negative edge lengths.

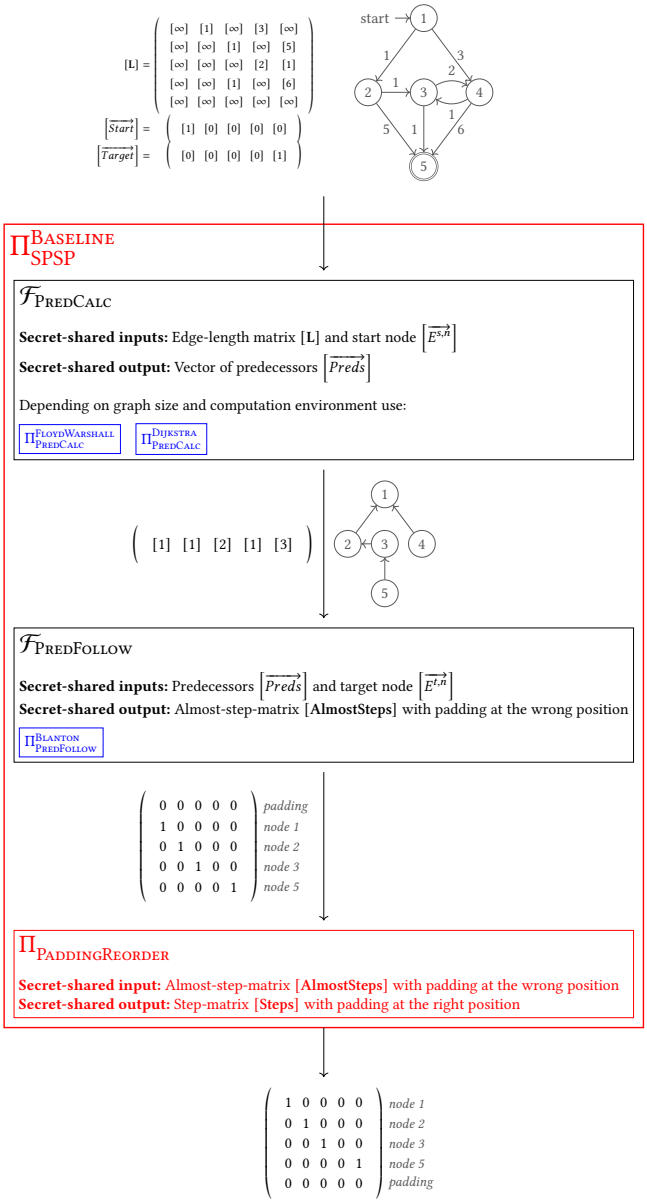


Figure 2: Overview of the baseline single-pair shortest-path protocol. Protocols newly introduced in this paper are highlighted in red and modified prior protocols in blue.

in $\Pi^{\text{FloydWarshall}}_{\text{PredCalc}}$ (Prot. 2). This technique is not applicable to the Dijkstra protocol, as the nodes are shuffled during the initialization, requiring additional measures to unshuffle the predecessors. $\Pi^{\text{Dijkstra}}_{\text{PredCalc}}$ (Prot. 3) builds on Prot. 9, our fix for the leakage in [6]'s Dijkstra protocol, which involves computing an inverse permutation. This is re-used in Line 13 to update the predecessors, thus minimizing the overall performance impact of the leakage prevention and predecessor calculation[6].

[6]This is no coincidence, as we explicitly forgo easier-to-prove leakage-suppression approaches to achieve this double-use of the inverse permutation.

Protocol 2 $\Pi^{\text{FloydWarshall}}_{\text{PredCalc}}\left([\mathbf{L}], [\overrightarrow{Start}]\right)$, i. e., the Floyd-Warshall protocol from [6] with predecessor computation. Our modifications are highlighted in blue.

1: **for** $1 \le i, j \le n$ **do**
2: $\quad[\mathbf{Dist}_{i,j}] \leftarrow \begin{cases} [\mathbf{L}_{i,j}] & \text{if } i \neq j \\ [0] & \text{if } i = j \end{cases}$
3: $\quad[\mathbf{Preds}_{i,j}] \leftarrow [i]$
4: **for** $1 \le k \le n$ **do**
5: $\quad$**for** $1 \le i, j \le n$ **do in parallel**
6: $\quad\quad[better] \leftarrow ([\mathbf{Dist}_{i,k}] + [\mathbf{Dist}_{k,j}] < [\mathbf{Dist}_{i,j}])$
7: $\quad\quad[\mathbf{Dist}_{i,j}] \leftarrow [better]\ ?\ [\mathbf{Dist}_{i,k}] + [\mathbf{Dist}_{k,j}]\ :\ [\mathbf{Dist}_{i,j}]$
8: $\quad\quad[\mathbf{Preds}_{i,j}] \leftarrow [better]\ ?\ [\mathbf{Preds}_{k,j}]\ :\ [\mathbf{Preds}_{i,j}]$
9: **return** $[\overrightarrow{Start}] \cdot [\mathbf{Preds}]$

Protocol 3 $\Pi^{\text{Dijkstra}}_{\text{PredCalc}}\left([\mathbf{L}], [\overrightarrow{Start}]\right)$, i. e., the Dijkstra protocol from [6] with predecessor computation and leakage prevention. Our modifications are highlighted in blue.

1: **for** $1 \le i \le n$ **do**
2: $\quad[\overrightarrow{Dist}_i] \leftarrow \left(1 - [\overrightarrow{Start}_i]\right) \cdot \infty$
3: $\quad\overrightarrow{Visited}_i \leftarrow 0$; $[\overrightarrow{InvPerm}_i] \leftarrow [i]$; $[\overrightarrow{Preds}_i] \leftarrow [i]$
4: Generate permutation $[\sigma]$ of size n and use it to shuffle $[\overrightarrow{Dist}]$, $[\overrightarrow{Preds}]$, $[\overrightarrow{InvPerm}]$ and $[\mathbf{L}]$.
5: **for** $1 \le iter \le n$ **do**
6: $\quad$**for** $1 \le j \le n$ **do** ▷ Prevent leakage as in Prot. 9.
7: $\quad\quad[\overrightarrow{Tmp}_j] \leftarrow (n+1) \cdot [\overrightarrow{Dist}_j] + [\overrightarrow{InvPerm}_j]$
8: $\quad i \leftarrow \text{Reveal}\left(\text{ArgMinOfUnvisited}\left([\overrightarrow{Tmp}], \overrightarrow{Visited}\right)\right)$
9: $\quad\overrightarrow{Visited}_i \leftarrow 1$
10: $\quad$**for** $1 \le j \le n$ **do in parallel**
11: $\quad\quad[isBetter] \leftarrow [\overrightarrow{Dist}_i] + [\mathbf{L}_{i,j}] < [\overrightarrow{Dist}_j]$
12: $\quad\quad[\overrightarrow{Dist}_j] \leftarrow [isBetter]\ ?\ [\overrightarrow{Dist}_i] + [\mathbf{L}_{i,j}]\ :\ [\overrightarrow{Dist}_j]$
13: $\quad\quad[\overrightarrow{Preds}_j] \leftarrow [isBetter]\ ?\ [\overrightarrow{InvPerm}_i]\ :\ [\overrightarrow{Preds}_j]$
14: Unshuffle $[\overrightarrow{Preds}]$ using $[\sigma]$ and **return** $[\overrightarrow{Preds}]$

An important detail of $\Pi^{\text{FloydWarshall}}_{\text{PredCalc}}$ and $\Pi^{\text{Dijkstra}}_{\text{PredCalc}}$ is that the predecessor of each node i is initialized as node i, i. e., with a single node loop. As the predecessors of unreachable nodes and the start node remain unchanged in both protocols, this single node loop can be used by the following path construction to detect the end of the path. Finally, these modifications do not change the asymptotic complexities of the original protocols presented in [6].

Step 2: Following the predecessors. Once the predecessors have been computed, the next step is the extraction of the path from these predecessors. It may appear that this problem is as easy as iteratively following the predecessors. However, the simplicity is thwarted by cases where the path has fewer than $n-1$ steps. As the protocol must not leak the number of steps in the path, special care is required to handle this case obliviously.

Our baseline uses a protocol by Blanton et al. [14] to follow the predecessors while hiding the path's number of steps, which, to

the best of our knowledge, is the only prior protocol capable of this. It adds a chain of fake nodes, and sets the predecessor of the start node to a fake node before following the predecessors. Once the full path has been traversed, the protocol then follows this chain of fake nodes, thus hiding the true number of steps in the path.

Compared to [14], our implementation $\Pi^{\text{Blanton}}_{\text{PredFollow}}$ (Prot. 4) introduces two minor optimizations: Fake nodes are added after the predecessor calculation, while [14] add fake nodes before their equivalent of the predecessor-calculation[7]. $\Pi^{\text{Blanton}}_{\text{PredFollow}}$ thus eliminates the fake node overhead during the predecessor calculation. The protocol also slightly changes the order of operations in the main loop resulting in more parallel computations.

Protocol 4 $\Pi^{\text{Blanton}}_{\text{PredFollow}}\left([\overrightarrow{Preds}], [\overrightarrow{Target}]\right)$, inspired by [14].

Add fake nodes:
1: **for** $1 \le i \le n$ **do in parallel**
2: $[cond] \leftarrow [\overrightarrow{Preds}_i] = i$
3: $[\overrightarrow{PredsWithFake}_i] \leftarrow [cond]\ ?\ n+1\ :\ [\overrightarrow{Preds}_i]$
4: $[\overrightarrow{PredsWithFake}_{n+i}] \leftarrow [n+i+1]$
5: $[\textbf{ResWithFake}_{1,i}] \leftarrow [\overrightarrow{Target}_i]$
Main loop:
6: **for** $1 \le i \le n-1$ **do**
7: $[nextNode] \leftarrow [\textbf{ResWithFake}_{i,\bullet}] \cdot [\overrightarrow{PredsWithFake}^{\mathsf{T}}]$
8: **for** $1 \le j \le 2n$ **do in parallel** ▷ Implicit $\Pi^{\text{ParallelComp}}_{\text{Demux}}$
9: $[\textbf{ResWithFake}_{i+1,j}] \leftarrow ([nextNode] = j)$
Reverse **ResWithFake** and remove fake nodes:
10: **for** $1 \le i, j \le n$ **do**
11: $[\textbf{Result}_{n-i+1,j}] \leftarrow [\textbf{ResWithFake}_{i,j}]$
12: **return** [Result]

Step 3: Reordering the padding. $\Pi^{\text{Blanton}}_{\text{PredFollow}}$'s output matrix is close to the wanted step matrix, but the zero-padding is at the beginning of the matrix, not the end. This is an artifact of the path being built back-to-front. Thus, the padding needs to be moved to make our baseline protocol compliant with $\mathcal{F}_{\text{SPSP}}$.

Conceptually, this is equivalent to sorting the rows of the matrix according to an "is node / is padding" key as constructed by $\Pi_{\text{PaddingReorder}}$ (Prot. 5). It is possible to construct a specialized $\log n$-depth sorting network for this particular key. We instead propose using the radix sort protocol by Hamada et al. [37], as it needs only one iteration to sort by the one-bit key. This reduces the effort of reordering the padding to a single shuffle of the step matrix.

Protocol 5 $\Pi_{\text{PaddingReorder}}$ ([AlmostSteps])

1: **for** $1 \le i \le n$ **do**
2: $[\overrightarrow{Keys}_i] \leftarrow 1 - \sum_{j=1}^{n} [\textbf{AlmostSteps}_{i,j}]$
▷ one-bit keys ⇒ only one radix sort iteration
3: **return** $\textsc{ObliviousRadixSort}\left([\textbf{AlmostSteps}], [\overrightarrow{Keys}]\right)$

[7] Blanton et al. [14] perform a BFS, not a general shortest path computation.

To summarize, the $\Pi^{\text{Baseline}}_{\text{SPSP}}$ first computes the shortest path's predecessors using $\Pi^{\text{Dijkstra}}_{\text{PredCalc}}$ or $\Pi^{\text{FloydWarshall}}_{\text{PredCalc}}$, depending on the graph size and computation environment. It then follows these predecessors using a modified version of the predecessor-following protocol by Blanton et al. [14] and finally reorders the padding. Constructing the path from the predecessors requires $O(n+S)$ rounds, and $O(n^2+S)$ communication and computation, where S is the effort of shuffling the step matrix. Finally, the $\Pi^{\text{Baseline}}_{\text{SPSP}}$'s control flow only depends on the publicly known number of nodes and computation environment, and no values are revealed. Hence, it trivially is secure in the ABB-hybrid-model.

5.2 Optimizing the predecessor-following

This section focuses on $\Pi^{\text{Baseline}}_{\text{SPSP}}$'s second step, the predecessor-following that computes the almost-step matrix from the previously created predecessors. Our baseline always uses $\Pi^{\text{Blanton}}_{\text{PredFollow}}$, which we decompose into sub-functionalities, each realizable by several interchangeable sub-protocols. As with the predecessor-calculation, the fastest sub-protocol depends on the graph size and computation environment, and our protocols adaptively select the sub-protocols accordingly[8].

This first decomposition observes that Lines 8 and 9 of $\Pi^{\text{Blanton}}_{\text{PredFollow}}$ construct a vector of length $2n$ that only contains zeroes except a single one at index $nextNode$. In other words, the protocol implicitly demultiplexes $[nextNode]$, and these lines could be replaced by other demultiplex protocols. This is made explicit in $\Pi^{\text{Blanton + A.D.}}_{\text{PredFollow}}$, presented in Fig. 3b, which can use any of the protocols presented in Sec. 3.3 or our new protocol of Sec. 4.2.

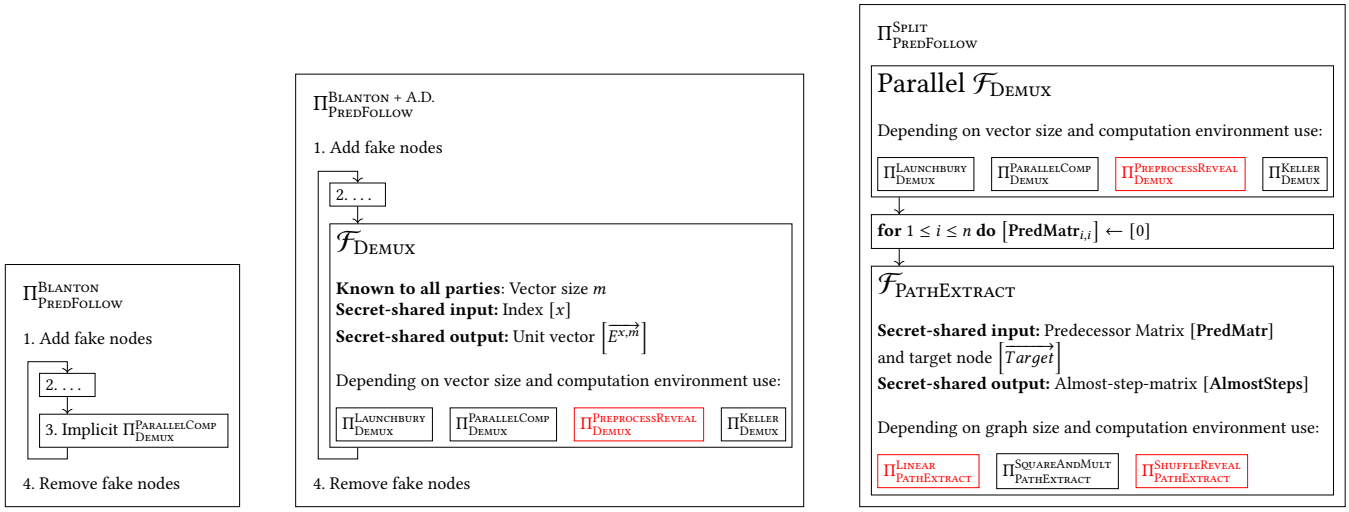
We push this decomposition further by splitting the predecessor-following into a separate "demultiplex" and "path-extraction" step, resulting in the protocol $\Pi^{\text{Split}}_{\text{PredFollow}}$ shown in Fig. 3c. The demultiplex step demultiplexes all predecessors in parallel, resulting in a matrix we call the predecessor matrix. The diagonal of this matrix is set to zero, converting the single-node loops that mark the end of the path into all-zero vectors.

Protocol 6 $\Pi^{\text{Linear}}_{\text{PathExtract}}\left([\textbf{PredMatr}], [\overrightarrow{Target}]\right)$

1: $[\textbf{Result}_{n,\bullet}] \leftarrow [\overrightarrow{Target}]$
2: **for** $1 \le i \le n-1$ **do**
3: $[\textbf{Result}_{n-i,\bullet}] \leftarrow [\textbf{Result}_{n-i+1,\bullet}] \cdot [\textbf{PredMatr}]$
4: **return** [Result]

This facilitates the creation of the padding in the following path-extraction step. For example, $\Pi^{\text{Linear}}_{\text{PathExtract}}$ (Prot. 6), which determines the predecessor of the current node using a vector-matrix product, obliviously creates the zero-padding once the end of the path has been reached without any fake nodes. Note, however, that the padding still must be reordered. In total, $\Pi^{\text{Linear}}_{\text{PathExtract}}$ requires $O(n)$ rounds, $O(n^3)$ computation, and $O(n^2)$ or $O(n^3)$ communication depending on the used primitive.

[8] We determine the fastest sub-protocol for each graph size and computation environment using microbenchmarks. Runtime estimations such as [47] could be used instead, but no runtime estimation tool is compatible with the MP-SPDZ version used in our evaluation ([47] only supports version 0.3.7, we use 0.4.2).



a) Blanton b) Blanton with adaptive demultiplex c) Split predecessor-following

Figure 3: The evolution of predecessor-following protocols. Sub-protocols newly introduced in this paper are highlighted in red.

The round complexity can be reduced using a technique introduced by Ehrmanntraut and Meyer [30]. Their BFS protocol implicitly includes a path-extraction protocol based on the observation that $\mathbf{Result}_{n-i,\bullet} = \overrightarrow{Target} \cdot \mathbf{PredMatr}^i$. They use the square-and-multiply algorithm to compute the powers of **PredMatr**, resulting in $\Pi^{\text{SquareAndMult}}_{\text{PathExtract}}$ that requires $O(\log n)$ rounds, $O(n^3 \log n)$ computation, and $O(n^2 \log n)$ or $O(n^3 \log n)$ communication.

Our final novel path-extraction protocol, $\Pi^{\text{ShuffleReveal}}_{\text{PathExtract}}$ (Prot. 7), iteratively follows the predecessors similarly to $\Pi^{\text{Linear}}_{\text{PathExtract}}$, with the difference that the nodes are shuffled first, and that the shuffled indices of the visited nodes are revealed. This facilitates the lookup of the current node's predecessor in the main loop. To ensure that the revealed node indices do not leak any information about the path, fake nodes as in $\Pi^{\text{Blanton}}_{\text{PredFollow}}$ are reintroduced, i. e., the end of the real path is routed into a chain of fake nodes. As the shuffling of nodes includes the fake nodes, the revealed shuffled indices of real nodes are not distinguishable from those of fake nodes. App. B.4 formally proves the security of $\Pi^{\text{ShuffleReveal}}_{\text{PathExtract}}$ in the ABB-hybrid-model. Overall, the protocol requires $O(n + S)$ rounds and communication, and $O(n^2 + S)$ computation, where S is the effort of shuffling a $(2n-1) \times (2n-1)$ matrix.

To summarize, the three path-extraction protocols offer different trade-offs between rounds, communication, and computation, and $\Pi^{\text{Split}}_{\text{PredFollow}}$ can use any of them. Splitting the predecessor-following, however, always incurs a computation overhead. While this overhead is non-asymptotic, i. e., hidden by the big-O notation, when $\Pi^{\text{ShuffleReveal}}_{\text{PathExtract}}$ is used, the optimized single-pair shortest-path protocol may forgo the split and use $\Pi^{\text{Blanton + A.D.}}_{\text{PredFollow}}$ instead.

5.3 Eliminating the padding-reorder

All predecessor-following protocols discussed in the previous section build the path back-to-front, and thus create the padding at the beginning of the matrix, requiring the padding-reorder step

Protocol 7 $\Pi^{\text{ShuffleReveal}}_{\text{PathExtract}}\left([\mathbf{PredMatr}], [\overrightarrow{Target}]\right)$

Add fake nodes:
1: Pad **PredMatr** to size $(2n-1) \times (2n-1)$ with $[0]$
2: **for** $1 \le i \le n$ **do** ▷ Connections to fake nodes.
3: $[\mathbf{PredMatr}_{i,n+1}] \leftarrow 1 - \sum_{j=1}^{n} [\mathbf{PredMatr}_{i,j}]$
4: **for** $1 \le i \le n-2$ **do** ▷ Connections between fake nodes.
5: $[\mathbf{PredMatr}_{n+i,n+i+1}] \leftarrow [1]$
Extract the path:
6: $[\sigma] \leftarrow$ GeneratePermutation $(2n-1)$
7: Shuffle [**PredMatr**] and $[\overrightarrow{Target}]$ using $[\sigma]$.
8: $current \leftarrow \text{Reveal}\left(\sum_{j=1}^{2n-1} j \cdot [\overrightarrow{Target}_j]\right)$
9: $[\mathbf{ResultWithFake}_{1,current}] \leftarrow [1]$
10: **for** $1 \le step \le n-1$ **do**
11: $next \leftarrow \text{Reveal}\left(\sum_{j=1}^{2n-1} j \cdot [\mathbf{PredMatr}_{current,j}]\right)$
12: $[\mathbf{ResultWithFake}_{step+1,next}] \leftarrow [1]$; $current \leftarrow next$
13: Unshuffle each row of [**ResultWithFake**] using $[\sigma]$.
Reverse **ResultWithFake** and remove fake nodes:
14: **for** $1 \le i, j \le n$ **do**
15: $[\mathbf{Result}_{n-i+1,j}] \leftarrow [\mathbf{ResultWithFake}_{i,j}]$
16: **return** [**Result**]

to move it to the correct position. This step is eliminated using a graph transformation trick that reverses all edges of the graph, and swaps the start and target node. For example, the edge $(1, 2)$ in Fig. 4's example input graph is transformed into the edge $(2, 1)$. The following predecessor calculation is then started from node 5, i. e., the target node of the initial input graph. As proven in App. B.2, the resulting predecessors are equal to the successors of the shortest path that ends in the target node in the input graph. In other words, the optimized protocol now computes pointers to the next node on

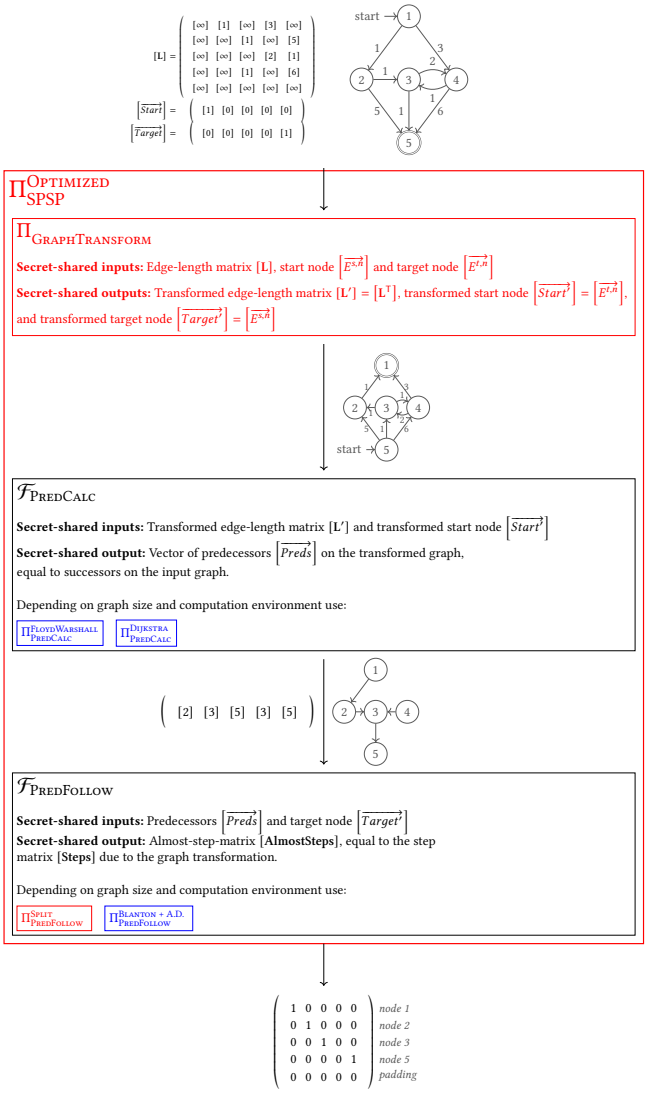


Figure 4: Overview of the optimized single-pair shortest-path protocol. Protocols newly introduced in this paper are highlighted in red and modified prior protocols in blue.

the shortest path instead of pointers to the previous node. Applying the previously discussed predecessor-following protocols on these successors directly yields step matrices with the padding at the correct position, and the padding-reorder is no longer needed.

This graph transformation is implemented by transposing the edge-length matrix of the graph and swapping start and target nodes. Thus, the graph transformation requires no communication and $O(n^2)$ computations, while eliminating the padding-reordering step that required $O(1)$ rounds and $O(n^2)$ communication.

To summarize, our optimized single-pair shortest-path protocol $\Pi^{\text{Optimized}}_{\text{SPSP}}$ combines the graph transformation with the optimized predecessor-following. It first applies the graph transformation, such that $\Pi^{\text{Dijkstra}}_{\text{PredCalc}}$ or $\Pi^{\text{FloydWarshall}}_{\text{PredCalc}}$ compute successors instead of predecessors. It then follows these successors using $\Pi^{\text{Blanton + A.D.}}_{\text{PredFollow}}$ or $\Pi^{\text{Split}}_{\text{PredFollow}}$, again picking the demultiplex and path-extraction sub-protocols fastest for the given graph size and computation environment. As all sub-protocols are secure in the ABB-hybrid-model and the selection only depends on public information, $\Pi^{\text{Optimized}}_{\text{SPSP}}$ is secure in the ABB-hybrid-model as well.

6 Evaluation

After the description of our evaluation setup in Sec. 6.1, Sec. 6.2 confirms that no demultiplex or path-extraction protocol dominates across all settings. Sec. 6.3 then evaluates the optimizations on the predecessor-following, and Sec. 6.4 compares the padding-reorder with the graph transformation. Finally, Sec. 6.5 presents start-to-end runtimes of the full single-pair shortest-path protocols.

6.1 Setup

We implement our protocols in MP-SPDZ [41] version 0.4.2. The primitive `replicated-ring` [9] is used to evaluate the protocols with semi-honest security, and `sy-rep-ring` [1] with malicious security. These primitives only support three parties. For those primitives, MP-SPDZ pre-processes material like random bits in batches on the fly. Unused material is cached, and the next batch is generated once the cache is empty. We set the maximum batch size to 1 000 000 to reflect our protocols' parallel nature.

All computations ran in containers with four cores of Intel Platinum 8160 CPUs (2.1 GHz base clock) and 50 GiB RAM shared among all three parties. Using NEON [47], three network settings are simulated: the LAN setting has a 1ms delay (i. e., 2ms "ping") and 1 Gbit/s bandwidth per party, the WAN setting a 10ms delay and 100 Mbit/s, and the unrestricted setting has no artificial limitations.

We use synthetic graphs, which does not affect runtimes as our data-oblivious protocols' control flow only depends on the number of nodes. Each computation was repeated ten times and all figures display the 95% confidence intervals as error bands, which are often so small that they are barely visible. Our implementations, runtime logs, and plotting scripts are publicly available[9].

6.2 Microbenchmarks of sub-protocols

To streamline the presentation of our results, we focus on the WAN setting with the semi-honest primitive in this section. Our evaluation's full data can be found in App. D and in the artifact.

Demultiplexing. We have evaluated the demultiplex protocol by Keller et al. [44] with and without bitpacking ($\Pi^{\text{Keller}}_{\text{Demux}}$, respectively $\Pi^{\text{Keller-NB}}_{\text{Demux}}$), the protocol by Launchbury et al. [50] ($\Pi^{\text{Launchbury}}_{\text{Demux}}$), a trivial protocol realized by parallel comparisons ($\Pi^{\text{ParallelComp}}_{\text{Demux}}$), and our novel demux protocol ($\Pi^{\text{PreprocessReveal}}_{\text{Demux}}$). Fig. 5 presents the average runtime per demultiplex for both single and parallel demultiplexes with semi-honest security in the WAN setting.

In both settings, $\Pi^{\text{Keller}}_{\text{Demux}}$ is significantly slower than $\Pi^{\text{Keller-NB}}_{\text{Demux}}$, e. g., 1.46 vs 0.34 seconds for a single demux of size 2000. As the bitpacking variant unpacks the bits via bit-decompositions, it requires both more rounds (179 vs 39) and communication (53KB vs 49KB).

For single demultiplexes, our new $\Pi^{\text{PreprocessReveal}}_{\text{Demux}}$ outperforms all other protocols across the evaluated vector sizes in the semi-honest WAN setting. It requires slightly fewer rounds (5 vs 7) than

[9] https://doi.org/10.5281/zenodo.22748227

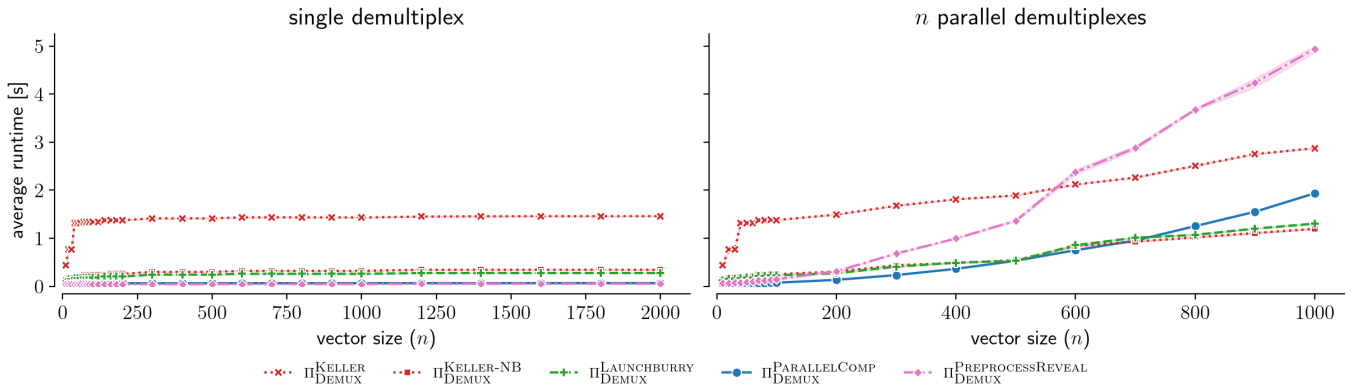


Figure 5: Runtimes of single and n parallel demultiplexes by vector size n with semi-honest security in the WAN setting.

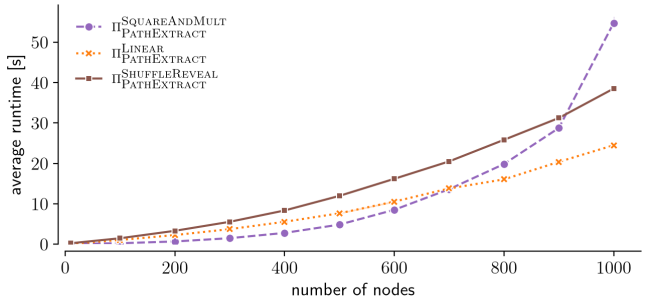


Figure 6: Runtimes of the path extraction protocols with semi-honest security in the WAN setting.

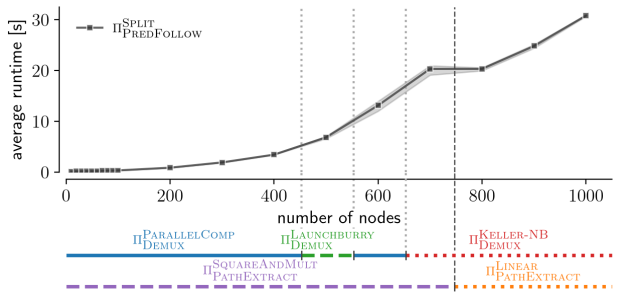


Figure 7: Runtime of $\Pi^{\text{Split}}_{\text{PredFollow}}$ with semi-honest security in the WAN setting, annotated with sub-protocol crossover points.

the next-fastest $\Pi^{\text{ParallelComp}}_{\text{Demux}}$ but more communication (e. g., 98KB vs 79KB at size 2000), making it well-suited for high-delay networks. In the semi-honest LAN setting, however, it outperforms $\Pi^{\text{ParallelComp}}_{\text{Demux}}$ only up to size 500 (App. D, Table 12), and its advantage does not generalize to other SMPC primitives as the semi-honest primitive shuffles particularly efficiently [11].

Surprisingly, for single demultiplexes, the simple $\Pi^{\text{ParallelComp}}_{\text{Demux}}$ always outperforms the more complex $\Pi^{\text{Keller-NB}}_{\text{Demux}}$ and $\Pi^{\text{Launchburry}}_{\text{Demux}}$. For parallel demultiplexes, both $\Pi^{\text{PreprocessReveal}}_{\text{Demux}}$ and $\Pi^{\text{ParallelComp}}_{\text{Demux}}$ run into a communication bottleneck. They require 49.2MB and 37.5MB, while $\Pi^{\text{Keller-NB}}_{\text{Demux}}$ and $\Pi^{\text{Launchburry}}_{\text{Demux}}$ need only 24.8MB and 26.8MB for 1000 demultiplexes of size 1000. Still, $\Pi^{\text{ParallelComp}}_{\text{Demux}}$ is the fastest up to size 400 and at size 600. On larger vectors, $\Pi^{\text{Keller-NB}}_{\text{Demux}}$ and $\Pi^{\text{Launchburry}}_{\text{Demux}}$ have comparable runtimes. Both thus provide good trade-offs, with the setting dictating whether focusing on rounds or communication is advantageous.

Path-extraction. As with demultiplexes, the fastest path-extraction protocol changes across graph sizes. In the semi-honest WAN setting presented in Fig. 6, $\Pi^{\text{SquareAndMult}}_{\text{PathExtract}}$ is fastest for up to 700 nodes and $\Pi^{\text{Linear}}_{\text{PathExtract}}$ beyond. $\Pi^{\text{SquareAndMult}}_{\text{PathExtract}}$ needs far fewer rounds (20 vs 1000 at 1000 nodes) but more communication (240MB vs 24MB) and computation ($O\left(n^3 \log n\right)$ vs $O\left(n^3\right)$). Our novel $\Pi^{\text{ShuffleReveal}}_{\text{PathExtract}}$ is never fastest here, requiring more rounds (1016) and communication (480MB), but only $O\left(n^2\right)$ computation. This is advantageous in faster networks, e. g., beyond 700 nodes in the semi-honest LAN setting (App. D, Table 28).

6.3 Predecessor following

These microbenchmarks decide which sub-protocol our predecessor-following protocols use. For example, Fig. 7 shows $\Pi^{\text{Split}}_{\text{PredFollow}}$ using $\Pi^{\text{Launchburry}}_{\text{Demux}}$ at 500 nodes, $\Pi^{\text{Keller-NB}}_{\text{Demux}}$ from 700 nodes on, and $\Pi^{\text{ParallelComp}}_{\text{Demux}}$ otherwise, and switching the path-extraction from $\Pi^{\text{SquareAndMult}}_{\text{PathExtract}}$ to $\Pi^{\text{Linear}}_{\text{PathExtract}}$ at 800 nodes. The other protocols do not adapt here as $\Pi^{\text{Blanton}}_{\text{PredFollow}}$ cannot, and $\Pi^{\text{Blanton + A.D.}}_{\text{PredFollow}}$ uses $\Pi^{\text{PreprocessReveal}}_{\text{Demux}}$, which always outperforms the other demultiplexes in this setting.

Fig. 8 shows the runtimes of the protocols with semi-honest security in the WAN setting. Due to our novel demultiplex protocol, $\Pi^{\text{Blanton + A.D.}}_{\text{PredFollow}}$ speeds up the predecessor following by a factor ranging from 1.34 to 1.59 compared to $\Pi^{\text{Blanton}}_{\text{PredFollow}}$. Surprisingly, this speedup exceeds the one observed in the demultiplex microbenchmarks. Our investigation in App. C attributes this to local computation factors such as speculative execution or caching. We thus claim microbenchmarks lead to good, but not necessarily optimal, sub-protocol choices. The effectiveness of such choices is highlighted

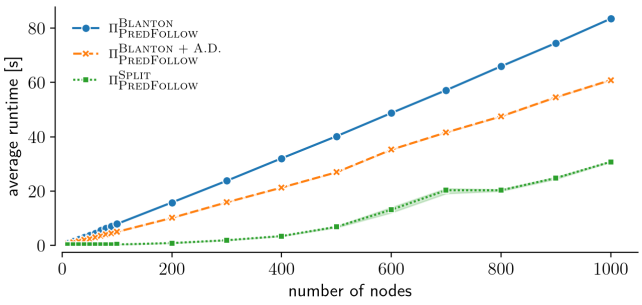


Figure 8: Runtimes of predecessor-following protocols with semi-honest security in the WAN setting.

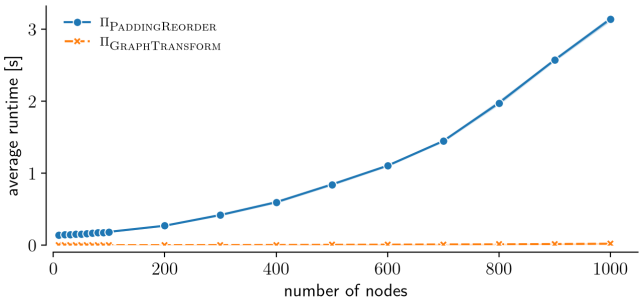


Figure 9: $\Pi_{\text{PaddingReorder}}$ and $\Pi_{\text{GraphTransform}}$'s runtimes with semi-honest security in the WAN setting.

by $\Pi^{\text{Split}}_{\text{PredFollow}}$, which further profits from the adaptive selection of path-extraction protocols. It outperforms $\Pi^{\text{Blanton}}_{\text{PredFollow}}$ by a factor of 2.73 (700 nodes) up to 23.1 (100 nodes).

In general, we observe the adaptive sub-protocol selection to be most effective in slow networks and on smaller graphs. This is because $\Pi^{\text{Blanton}}_{\text{PredFollow}}$ requires $O\left(n^2\right)$ computation, which is asymptotically optimal due to the $O\left(n^2\right)$ output size, while our alternatives trade more computation for fewer rounds or less communication. They thus help most where local computation matters least, i. e., on small graphs or in slow networks.

6.4 Padding reorder and graph transformation

Our single-pair shortest-path protocols require methods to place the zero-padding in the correct position at the end of the path. Fig. 9 displays both approaches' runtimes in the semi-honest WAN setting. Unsurprisingly, $\Pi_{\text{GraphTransform}}$ is up to three orders of magnitude faster than $\Pi_{\text{PaddingReorder}}$ on all graph sizes as it consists of local operations while $\Pi_{\text{PaddingReorder}}$ sorts the almost-step matrix.

6.5 Start-to-end runtimes

Finally, we evaluate the start-to-end runtimes of $\Pi^{\text{Baseline}}_{\text{SPSP}}$ and $\Pi^{\text{Optimized}}_{\text{SPSP}}$. Both were introduced in this paper, and $\Pi^{\text{Baseline}}_{\text{SPSP}}$ has been faithfully optimized, i. e., we are not aware of trivial ways to reduce its rounds, communication, or computation. Importantly, it already adaptively selects either $\Pi^{\text{Dijkstra}}_{\text{PredCalc}}$ or $\Pi^{\text{FloydWarshall}}_{\text{PredCalc}}$ for the predecessor-calculation, whose runtime differs by up to two orders of magnitude (App. D, Tables 3 to 8).

Table 2: Comparison of the start-to-end runtimes with semi-honest security in the WAN setting.

n	$\Pi^{\text{Baseline}}_{\text{SPSP}}$ [s]	$\Pi^{\text{Optimized}}_{\text{SPSP}}$ [s]	Speedup Absolute [s]	Speedup Relative
100	31.46	23.62	7.84	1.33
200	132.31	115.15	17.15	1.15
300	282.98	260.25	22.73	1.09
400	396.42	368.08	28.34	1.08
500	510.55	475.30	35.24	1.07
600	633.56	594.65	38.92	1.07
700	759.62	715.44	44.18	1.06
800	887.98	836.91	51.07	1.06
900	1 014.99	960.05	54.95	1.06
1000	1 139.65	1 085.60	54.05	1.05

Relative speedups are largest on small graphs and shrink as graphs grow. In the semi-honest WAN setting, a 49% speedup is observed at 50 nodes, which, as shown by Table 2, shrinks to 33% at 100 and 5% at 1000 nodes. This partly reflects our optimized predecessor-following helping most on small graphs, but mainly that the predecessor-calculation's runtime grows faster than the predecessor-following and padding-reordering. Its share of the total runtime thus grows, diluting the effects of Secs. 5.2 and 5.3.

7 Conclusion

In this work, we have presented the first single-pair shortest-path SMPC protocols that compute paths and, in contrast to prior work, not just lengths. First, a baseline was introduced that computes the predecessors using modified prior (length of) shortest-path protocols, follows them using a modified protocol by Blanton et al. [14], and reorders the padding. We then optimized the predecessor-following by decomposing it into demultiplex and path-extraction sub-functionalities, introducing new sub-protocols for each, and adaptively selecting sub-protocols. We further used a graph transformation to eliminate the padding-reorder.

The evaluation showed that individually, the optimizations are effective. Our optimized predecessor-following is up to 23 times faster than Blanton et al. [14]'s protocol, and the graph transformation is orders of magnitude faster than the padding-reorder. The overall impact of the optimizations is diluted by the predecessor-calculation, such that our optimized single-pair shortest-path protocol is 5% to 49% faster than the baseline.

Further, the decomposition into sub-functionalities promotes a transfer of results. Any SMPC protocol that explicitly uses a demultiplex can adopt our novel demultiplex protocol and adaptive selection as a drop-in replacement, making our work of interest to oblivious data structures [44], softmax protocols [45], kidney donor exchange [16], and protocols that use demultiplexes implicitly [14, 15]. Predecessor-following and path-extraction likewise benefit protocols that internally construct paths, such as [14, 30].

Acknowledgements

This work was partially funded by the German Federal Ministry of Research, Technology and Space (BMFTR) under grant number 03WIR3808A.

References

[1] Mark Abspoel, Anders Dalskov, Daniel Escudero, and Ariel Nof. 2021. An Efficient Passive-to-Active Compiler for Honest-Majority MPC over Rings. In *Applied Cryptography and Network Security (Lecture Notes in Computer Science)*, Kazue Sako and Nils Ole Tippenhauer (Eds.). Springer International Publishing, Cham. doi:10.1007/978-3-030-78375-4_6

[2] A. Adriansyah, B.F. van Dongen, and W.M.P. van der Aalst. 2011. Conformance Checking Using Cost-Based Fitness Analysis. In *2011 IEEE 15th International Enterprise Distributed Object Computing Conference*. 55–64. doi:10.1109/EDOC.2011.12

[3] 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. doi:10.1007/978-3-031-17143-7_37

[4] 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. doi:10.1007/978-3-319-15943-0_13

[5] Mohammad Anagreh and Peeter Laud. 2025. Security Proof of Single-Source Shortest Distance Protocols Built on Secure Multiparty Computation Protocols. *Cryptography* 9, 1 (2025), 1. doi:10.3390/cryptography9010001

[6] Mohammad Anagreh, Peeter Laud, and Eero Vainikko. 2021. Parallel Privacy-Preserving Shortest Path Algorithms. *Cryptography* 5, 4 (2021). doi:10.3390/cryptography5040027

[7] 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. doi:10.5220/0010775700003120

[8] Mohammad Anagreh, Eero Vainikko, and Peeter Laud. 2021. Parallel Privacy-Preserving Shortest Paths by Radius-Stepping. In *2021 29th Euromicro International Conference on Parallel, Distributed and Network-Based Processing (PDP)*. 276–280. doi:10.1109/PDP52278.2021.00051

[9] 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 *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*. ACM, New York, NY, USA. doi:10.1145/2976749.2978331

[10] Toshinori Araki, Jun Furukawa, Kazuma Ohara, Benny Pinkas, Hanan Rosemarin, and Hikaru Tsuchida. 2021. Secure Graph Analysis at Scale. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security (CCS '21)*. ACM, New York, NY, USA. doi:10.1145/3460120.3484560

[11] Gilad Asharov, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Ariel Nof, Benny Pinkas, Katsumi Takahashi, and Junichi Tomida. 2022. Efficient Secure Three-Party Sorting with Applications to Data Analysis and Heavy Hitters. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS '22)*. ACM, New York, NY, USA. doi:10.1145/3548606.3560691

[12] Richard Bellman. 1958. On a Routing Problem. *Quart. Appl. Math.* 16, 1 (1958), 87–90. doi:10.1090/qam/102435

[13] Claude Berge. 1962. *The Theory of Graphs and Its Applications*. Methuen.

[14] Marina Blanton, Aaron Steele, and Mehrdad Aliasgari. 2013. Data-Oblivious Graph Algorithms for Secure Computation and Outsourcing. In *Proceedings of the 8th ACM SIGSAC Symposium on Information, Computer and Communications Security (ASIA CCS '13)*. ACM, New York, NY, USA. doi:10.1145/2484313.2484341

[15] Enrico Bottazzi, Chan Nam Ngo, and Masato Tsutsumi. 2024. Multilateral Trade Credit Set-off in MPC via Graph Anonymization and Network Simplex. Cryptology ePrint Archive, Paper 2024/2037. https://eprint.iacr.org/2024/2037

[16] Malte Breuer, Ulrike Meyer, and Susanne Wetzel. 2024. Efficient Privacy-Preserving Approximation of the Kidney Exchange Problem. In *Proceedings of the 19th ACM Asia Conference on Computer and Communications Security (ASIA CCS '24)*. ACM, New York, NY, USA, 306–322. doi:10.1145/3634737.3645015

[17] Malte Breuer, Ulrike Meyer, Susanne Wetzel, and Anja Mühlfeld. 2020. A Privacy-Preserving Protocol for the Kidney Exchange Problem. In *Proceedings of the 19th Workshop on Privacy in the Electronic Society (WPES '20)*. ACM, New York, NY, USA, 151–162. doi:10.1145/3411497.3420213

[18] Andreas Brüggemann, Thomas Schneider, and Maximilian Stillger. 2026. Additions, Multiplications, and the Interaction In-Between: Optimizing MPC Protocols via Leveled Linear Secret Sharing. Cryptology ePrint Archive, Paper 2026/368. https://eprint.iacr.org/2026/368

[19] R. G. Busacker and P. J. Gowen. 1960. *A Procedure for Determining a Family of Minimum-Cost Network Flow Patterns*. Technical Report AD249662. Operations Research Office, Johns Hopkins Univ., Bethesda, Md.

[20] Niklas Büscher, Daniel Demmler, Stefan Katzenbeisser, David Kretzmer, and Thomas Schneider. 2018. HyCC: Compilation of Hybrid Protocols for Practical Secure Computation. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18)*. ACM, New York, NY, USA, 847–861. doi:10.1145/3243734.3243786

[21] Nairen Cao and Jeremy T. Fineman. 2023. Parallel Exact Shortest Paths in Almost Linear Work and Square Root Depth. In *Proceedings of the 2023 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*. Society for Industrial and Applied Mathematics, 4354–4372. doi:10.1137/1.9781611977554.ch166

[22] Nairen Cao, Jeremy T. Fineman, and Katina Russell. 2020. Efficient Construction of Directed Hopsets and Parallel Approximate Shortest Paths. In *Proceedings of the 52nd Annual ACM SIGACT Symposium on Theory of Computing (STOC 2020)*. ACM, New York, NY, USA, 336–349. doi:10.1145/3357713.3384270

[23] Octavian Catrina and Sebastiaan de Hoogh. 2010. Improved Primitives for Secure Multiparty Integer Computation. In *Security and Cryptography for Networks (Lecture Notes in Computer Science)*. Springer, Berlin, Heidelberg. doi:10.1007/978-3-642-15317-4_13

[24] Edward Chen, Jinhao Zhu, Alex Ozdemir, Riad S. Wahby, Fraser Brown, and Wenting Zheng. 2023. Silph: A Framework for Scalable and Accurate Generation of Hybrid MPC Protocols. In *2023 IEEE Symposium on Security and Privacy (SP)*. 848–863. doi:10.1109/SP46215.2023.10179397

[25] Koji Chida, Koki Hamada, Dai Ikarashi, Ryo Kikuchi, Daniel Genkin, Yehuda Lindell, and Ariel Nof. 2023. Fast Large-Scale Honest-Majority MPC for Malicious Adversaries. *Journal of Cryptology* 36, 3 (2023), 15. doi:10.1007/s00145-023-09453-7

[26] 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 (Lecture Notes in Computer Science)*. Springer International Publishing, Cham. doi:10.1007/978-3-319-96881-0_26

[27] Ivan Damgård, Daniel Escudero, Tore Frederiksen, Marcel Keller, Peter Scholl, and Nikolaj Volgushev. 2019. New Primitives for Actively-Secure MPC over Rings with Applications to Private Machine Learning. In *2019 IEEE Symposium on Security and Privacy (SP)*. 1102–1120. doi:10.1109/SP.2019.00078

[28] Ivan Damgård and Jesper Buus Nielsen. 2007. Scalable and Unconditionally Secure Multiparty Computation. In *Advances in Cryptology - CRYPTO 2007 (Lecture Notes in Computer Science)*, Alfred Menezes (Ed.). Springer, Berlin, Heidelberg. doi:10.1007/978-3-540-74143-5_32

[29] E. W. Dijkstra. 2022. A Note on Two Problems in Connexion with Graphs. In *Edsger Wybe Dijkstra: His Life,Work, and Legacy* (1 ed.). Vol. 45. ACM, New York, NY, USA.

[30] 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)*. ACM. doi:10.1145/3689943.3695041

[31] Vincent Ehrmanntraut and Ulrike Meyer. 2026. Going Faster: Privacy-Preserving Shortest Paths from Start to End. In *Proceedings of the 25th Workshop on Privacy in the Electronic Society (WPES '26)*. ACM, New York, NY, USA. doi:10.1145/3847192.3847373

[32] Francesca Falzon, Esha Ghosh, Kenneth G. Paterson, and Roberto Tamassia. 2024. PathGES: An Efficient and Secure Graph Encryption Scheme for Shortest Path Queries. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security (CCS '24)*. ACM. doi:10.1145/3658644.3670305

[33] Vivian Fang, Lloyd Brown, William Lin, Wenting Zheng, Aurojit Panda, and Raluca Ada Popa. 2022. CostCO: An Automatic Cost Modeling Framework for Secure Multi-Party Computation. In *2022 IEEE 7th European Symposium on Security and Privacy (EuroS&P)*. 140–153. doi:10.1109/EuroSP53844.2022.00017

[34] Robert W. Floyd. 1962. Algorithm 97: Shortest Path. *Commun. ACM* 5, 6 (1962), 345.

[35] Linton C. Freeman. 1977. A Set of Measures of Centrality Based on Betweenness. *Sociometry* 40, 1 (1977), 35–41. jstor:3033543 doi:10.2307/3033543

[36] Esha Ghosh, Seny Kamara, and Roberto Tamassia. 2021. Efficient Graph Encryption Scheme for Shortest Path Queries. In *Proceedings of the 2021 ACM Asia Conference on Computer and Communications Security (ASIA CCS '21)*. ACM, New York, NY, USA, 516–525. doi:10.1145/3433210.3453099

[37] Koki Hamada, Dai Ikarashi, Koji Chida, and Katsumi Takahashi. 2014. Oblivious Radix Sort: An Efficient Sorting Algorithm for Practical Secure Multi-party Computation. Cryptology ePrint Archive, Paper 2014/121. https://eprint.iacr.org/2014/121

[38] 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 (Lecture Notes in Computer Science)*. Springer International Publishing, Cham. doi:10.1007/978-3-319-44618-9_9

[39] Pesho Ivanov, Benjamin Bichsel, Harun Mustafa, André Kahles, Gunnar Rätsch, and Martin Vechev. 2020. AStarix: Fast and Optimal Sequence-to-Graph Alignment. In *Research in Computational Molecular Biology (Lecture Notes in Computer Science)*, Russell Schwartz (Ed.). Springer International Publishing, Cham, 104–119. doi:10.1007/978-3-030-45257-5_7

[40] Siddharth Kapoor, Nishat Koti, Varsha Bhat Kukkala, Arpita Patra, and Bhavish Raj Gopal. 2025. GraSP: Secure Collaborative Graph Processing Made Scalable. Cryptology ePrint Archive, Paper 2025/590. https://eprint.iacr.org/2025/590
[41] Marcel Keller. 2020. MP-SPDZ: A Versatile Framework for Multi-Party Computation. In *Proceedings of the 2020 ACM SIGSAC Conference on Computer and Communications Security (CCS '20)*. ACM, New York, NY, USA. doi:10.1145/3372297.3417872
[42] Marcel Keller, Emmanuela Orsini, Dragos Rotaru, Peter Scholl, Eduardo Soria-Vazquez, and Srinivas Vivek. 2017. Faster Secure Multi-party Computation of AES and DES Using Lookup Tables. In *Applied Cryptography and Network Security (Lecture Notes in Computer Science)*, Dieter Gollmann, Atsuko Miyaji, and Hiroaki Kikuchi (Eds.). Springer International Publishing, Cham. doi:10.1007/978-3-319-61204-1_12
[43] Marcel Keller, Emmanuela Orsini, and Peter Scholl. 2016. MASCOT: Faster Malicious Arithmetic Secure Computation with Oblivious Transfer. In *Proceedings of the 2016 ACM SIGSAC Conference on Computer and Communications Security (CCS '16)*. ACM, New York, NY, USA. doi:10.1145/2976749.2978357
[44] 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. doi:10.1007/978-3-662-45608-8_27
[45] Marcel Keller and Ke Sun. 2026. On MPC-friendly Softmax. Cryptology ePrint Archive, Paper 2026/944. https://eprint.iacr.org/2026/944
[46] Philip N. Klein and Sairam Subramanian. 1997. A Randomized Parallel Algorithm for Single-Source Shortest Paths. *Journal of Algorithms* 25, 2 (1997), 205–220. doi:10.1006/jagm.1997.0888
[47] Andreas Klinger, Vincent Ehrmanntraut, and Ulrike Meyer. 2024. Estimating the Runtime and Global Network Traffic of SMPC Protocols. In *Proceedings of the Fourteenth ACM Conference on Data and Application Security and Privacy (CODASPY '24)*. ACM, New York, NY, USA. doi:10.1145/3626232.3653258
[48] Nishat Koti, Varsha Bhat Kukkala, Arpita Patra, and Bhavish Raj Gopal. 2024. Graphiti: Secure Graph Computation Made More Scalable. In *Proceedings of the 2024 on ACM SIGSAC Conference on Computer and Communications Security (CCS '24)*. ACM, New York, NY, USA. doi:10.1145/3658644.3670393
[49] Peeter Laud. 2015. Stateful Abstractions of Secure Multiparty Computation. In *Applications of Secure Multiparty Computation*. Cryptology and Information Security Series, Vol. 13. IOS Press, Amsterdam, 26–42.
[50] John Launchbury, Iavor S. Diatchki, Thomas DuBuisson, and Andy Adams-Moran. 2012. Efficient Lookup-Table Protocol in Secure Multiparty Computation. In *Proceedings of the 17th ACM SIGPLAN International Conference on Functional Programming (ICFP '12)*. ACM, New York, NY, USA, 189–200. doi:10.1145/2364527.2364556
[51] Chang Liu, Xiao Shaun Wang, Kartik Nayak, Yan Huang, and Elaine Shi. 2015. ObliVM: A Programming Framework for Secure Computation. In *2015 IEEE Symposium on Security and Privacy*. doi:10.1109/SP.2015.29
[52] Thomas Loruenser, Florian Wohner, and Stephan Krenn. 2022. A Verifiable Multiparty Computation Solver for the Assignment Problem and Applications to Air Traffic Management. In *Proceedings of the 2022 on Cloud Computing Security Workshop (CCSW '22)*. ACM, 41–51. doi:10.1145/3560810.3564263
[53] Xianrui Meng, Seny Kamara, Kobbi Nissim, and George Kollios. 2015. GRECS: Graph Encryption for Approximate Shortest Distance Queries. In *Proceedings of the 22nd ACM SIGSAC Conference on Computer and Communications Security (CCS '15)*. ACM. doi:10.1145/2810103.2813672
[54] Payman Mohassel and Peter Rindal. 2018. ABY3: A Mixed Protocol Framework for Machine Learning. In *Proceedings of the 2018 ACM SIGSAC Conference on Computer and Communications Security (CCS '18)*. ACM, New York, NY, USA. doi:10.1145/3243734.3243760
[55] 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. doi:10.1109/SP.2015.30
[56] Benjamin Ostrovsky. 2024. Privacy-Preserving Dijkstra. In *Advances in Cryptology – CRYPTO 2024 (Lecture Notes in Computer Science)*. Springer Nature Switzerland, Cham. doi:10.1007/978-3-031-68400-5_3
[57] Jelili Oyelade, Itunuoluwa Isewon, Olufemi Aromolaran, Efosa Uwoghiren, Titilope Dokunmu, Solomon Rotimi, Oluwadurotimi Aworunse, Olawole Obembe, and Ezekiel Adebiyi. 2019. Computational Identification of Metabolic Pathways of Plasmodium Falciparum Using the K-Shortest Path Algorithm. *International Journal of Genomics* 2019, 1 (2019), 1750291. doi:10.1155/2019/1750291
[58] Václav Rozhoň, Michael Elkin, Christoph Grunau, and Bernhard Haeupler. 2022. Deterministic Low-Diameter Decompositions for Weighted Graphs and Distributed and Parallel Applications. In *2022 IEEE 63rd Annual Symposium on Foundations of Computer Science (FOCS)*. 1114–1121. doi:10.1109/FOCS54457.2022.00107
[59] Václav Rozhoň, Bernhard Haeupler, Anders Martinsson, Christoph Grunau, and Goran Zuzic. 2023. Parallel Breadth-First Search and Exact Shortest Paths and Stronger Notions for Approximate Distances. In *Proceedings of the 55th Annual ACM Symposium on Theory of Computing (STOC 2023)*. ACM, New York, NY, USA. doi:10.1145/3564246.3585235
[60] Meng Shen, Baoli Ma, Liehuang Zhu, Rashid Mijumbi, Xiaojiang Du, and Jiankun Hu. 2018. Cloud-Based Approximate Constrained Shortest Distance Queries Over Encrypted Graphs With Privacy Protection. *IEEE Transactions on Information Forensics and Security* 13, 4 (2018), 940–953. doi:10.1109/TIFS.2017.2774451
[61] Thomas H. Spencer. 1997. Time-Work Tradeoffs for Parallel Algorithms. *J. ACM* 44, 5 (1997), 742–778. doi:10.1145/265910.265923
[62] Xiao Shaun Wang, Kartik Nayak, Chang Liu, T-H. Hubert Chan, Elaine Shi, Emil Stefanov, and Yan Huang. 2014. Oblivious Data Structures. In *Proceedings of the 2014 ACM SIGSAC Conference on Computer and Communications Security (CCS '14)*. ACM, New York, NY, USA. doi:10.1145/2660267.2660314
[63] Stefan Wüller. 2018. *Privacy-Preserving Electronic Bartering*. Ph. D. Dissertation. RWTH Aachen University. doi:10.18154/RWTH-2018-222969

A Leakages in prior (length of) shortest-path protocols

This section discusses a small and subtle leakage in (length of) shortest path protocols by Aly and Cleemput [3] and by Anagreh et al. [6]. Both protocols are inspired by Dijkstra's algorithm and follow the same approach presented in Sec. 3.1, which shuffles the nodes and reveals the shuffled index of the unvisited node closest to the start node. This enables efficient updates of the node's neighbors' lengths. It is claimed that this approach is secure since the shuffling hides the real order in which the nodes are visited, and that this approach can be simulated by choosing a random permutation of $1, \ldots, n$ as revealed node order. However, [3, 6] both protocols use the blinding permutation as tie-breaker when multiple nodes have the same length to the starting node, which is the source of the leakage discussed in this section.

Consider, for example, Prot. 8, the ArgMin sub-protocol used by Aly and Cleemput [3]. It performs a straightforward linear search to find the node with the minimal length to the start node. However, if there are multiple nodes that all have the same (minimal) length to the start node, it will just return the first of these nodes it encounters, which is the node with the lowest shuffled index. This is also the case for the ArgMin sub-protocol used by Anagreh et al. [6], which splits the input vector in two halves and recursively finds the minimum and the corresponding index in both halves. If the minima of both halves are equal, it always returns the lower shuffled index.

Protocol 8 ArgMin sub-protocol used by Aly and Cleemput [3].

function ArgMin $\left([\overrightarrow{Dist}], i\right)$
  ▷ i is used to reduce the search space in the i'th Dijkstra
    ▷ iteration. This is not relevant for the leakage.
  $[d] \leftarrow [\infty]$
  $[v] \leftarrow [\bot]$
  for $i \leq j \leq n$ **do**
    $[cond] \leftarrow [\overrightarrow{Dist}_j] < [d]$
    $[v] \leftarrow [cond]\ ?\ j\ :\ [v]$
    $[d] \leftarrow [cond]\ ?\ [\overrightarrow{Dist}_j]\ :\ [d]$
  return $[v]$

This behavior becomes problematic when multiple nodes have the same length to the start node. In that case, the gates will always select the node with the lowest shuffled index. In the following iteration, the node with the second-lowest shuffled index will be selected, and so on. As a result, the revealed shuffled node indices are strictly increasing as long as the corresponding nodes have the same length to the starting node.

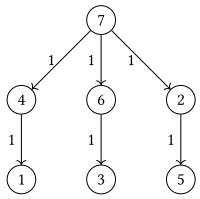


Figure 10: An example graph on which the leakage occurs. The node labels are the shuffled node indices.

Consider, for example, the graph presented in Fig. 10. No matter which permutation is used during the protocol executions, the second revealed node index will always be smaller than the third revealed node index. As a consequence, some traces will never occur, for example "7 6 4 2 1 3 5". This breaks the security arguments by Aly and Cleemput [3] and Anagreh et al. [6] that claim that the revealed trace is a uniformly random permutation of $1, \ldots, n$.

Beyond the academic discussion of the leakage, we want to provide some intuition on the leakage: Each time that the revealed shuffled node index decreases, it is revealed that the length to the starting node increases. On the other hand, it is possible for the length to the starting node to increase without a decrease in the shuffled node indices. This allows deriving further information about the graph, e. g., a lower bound on the number of layers in the graph, and an upper bound for the size of each layer.

Depending on the application of the protocols, the practical implications of this leakage vary significantly. If it is known a priori from the use-case that each node has a distinct length to the start node, this leakage becomes irrelevant. On the other hand, when these protocols are used as part of larger protocols, e. g., to find min-cost-max-flows, the leakage might reveal information about the state of the larger protocol, which ultimately could reveal more sensitive information.

Luckily, the leakage can be avoided by making the node choice independent of the shuffling permutation. To do so, we propose the original, unshuffled, node index as tie-breaker for the case that two nodes have the same length to the starting node. As shown in Prot. 9, this is done by first shuffling $[\overrightarrow{InvPerm}]$, a vector containing the values $1, \ldots, n$, which turns it into the inverse of the shuffling permutation $[\sigma]$. Afterward, the tie-breaker is incorporated into the node selection by first multiplying all lengths to the start node by $n + 1$, and then adding the tie-breaker in Line 8. This allows selecting the nearest node with any ArgMin gate, i. e., this fix is compatible with the protocol by Aly and Cleemput [3] as well as the protocol by Anagreh et al. [6]. It is assumed that the ArgMin procedure ignores already visited nodes.

The performance penalty of this fix should be minimal. This is because the only additional communication introduced by the fix is the shuffling of $[\overrightarrow{InvPerm}]$, which can be done in parallel to the shuffling of the edge-length matrix and the length vector. Building the temporary vector during the iterations of the protocols is done using local computations only, i. e., no communication overhead occurs during the iterations compared to the leaky Dijkstra protocols. Furthermore, $[\overrightarrow{InvPerm}]$ is useful when computing predecessors, as shown in Sec. 5.1.

Protocol 9 Dijkstra protocol without leakage.

1: **function** Dijkstra $\left([\mathbf{L}], [\overrightarrow{Start}]\right)$
2: **for** $1 \le i \le n$ **do**
3: $[\overrightarrow{Dist}_i] \leftarrow \left(1 - [\overrightarrow{Start}_i]\right) \cdot \infty$
4: $\overrightarrow{Visited}_i \leftarrow 0$; $[\overrightarrow{InvPerm}_i] \leftarrow [i]$
5: Generate permutation $[\sigma]$ of size n and use it to shuffle $[\overrightarrow{Dist}]$, $[\overrightarrow{InvPerm}]$ and $[\mathbf{L}]$.
6: **for** $1 \le \text{iter} \le n$ **do**
7: **for** $1 \le j \le n$ **do**
8: $[\overrightarrow{Tmp}_j] \leftarrow (n+1) \cdot [\overrightarrow{Dist}_j] + [\overrightarrow{InvPerm}_j]$
9: $i \leftarrow$ Reveal $\left(\text{ArgMinOfUnvisited}\left([\overrightarrow{Tmp}], \overrightarrow{Visited}\right)\right)$
10: $\overrightarrow{Visited}_i \leftarrow 1$
11: Update the lengths of i's neighbors.
12: Unshuffle $[\overrightarrow{Dist}]$ using $[\sigma]$ and **return** $[\overrightarrow{Dist}]$

The protocol stays correct due to the multiplication of the lengths by $n + 1$, as the tie-breaker can add at most n to the temporary lengths. Finally, we prove that this fix really fixes the leakage. This requires a brief clarification of the math around shuffled vectors, which tends to be counterintuitive. For example, when $\overrightarrow{B}$ is the shuffled version of $\overrightarrow{A}$ then $\overrightarrow{B}_i = \overrightarrow{A}_{\sigma^{-1}(i)}$, as shown in Eq. (1).

$$\overrightarrow{B} = \sigma(\overrightarrow{A}) \Leftrightarrow \forall i : \overrightarrow{B}_{\sigma(i)} = \overrightarrow{A}_i \Leftrightarrow \forall i : \overrightarrow{B}_i = \overrightarrow{A}_{\sigma^{-1}(i)} \tag{1}$$

It gets more complicated when an ArgMin over a shuffled vector is calculated. Eq. (2) shows that an ArgMin over a shuffled vector or function is equal to the inverse permutation of the ArgMin on the unshuffled vector or function. The equation uses the function notation with the constraint that f is injective, i. e., that each $f(j)$ is distinct. This ensures that the ArgMin is well-defined without the need to specify a tie-breaker.

$$\begin{aligned}
&\sigma^{-1}\left(\text{ArgMin}_{1\le j\le n}\left(f(j)\right)\right)\\
=&\sigma^{-1}\left(j \text{ such that } f(j) = \text{Min}_k\left(f(k)\right)\right)\\
=&\sigma^{-1}\left(j \text{ such that } i = \sigma^{-1}(j) \wedge f(\sigma(i)) = \text{Min}_k\left(f(k)\right)\right)\\
=&i \text{ such that } f(\sigma(i)) = \text{Min}_k\left(f(k)\right)\\
=&\text{ArgMin}_{1\le i\le n}\left(f(\sigma(i))\right)
\end{aligned} \tag{2}$$

Using these tools, Theorem A.1 shows that the revealed traces of Prot. 9 indeed are simulatable by choosing a random permutation of $1, \ldots, n$.

Theorem A.1. *Prot. 9 is secure in the ABB model, assuming Lines 5, 11 and 12 of Prot. 9 and ArgMin are implemented securely in the ABB model.*

Proof. As stated in [5], it only needs to be shown that the revealed order of visited nodes can be simulated.

To do so, let $G = (V = \{1, \ldots, n\}, \mathbf{L})$ be a graph with edge lengths and $v \in V$ be a starting node. Further, let $u^1, \ldots, u^n$ be the trace of revealed node indices obtained from executing the Prot. 9 on G, v without shuffling the nodes, and $\overrightarrow{Dist'}^{\text{iter}}$ the state of the $\overrightarrow{Dist}$ vector before the ArgMin is calculated in iteration iter.

We claim that this unshuffled trace is the same underlying trace for all executions of Prot. 9. In other words, if the protocol shuffles the nodes using the permutation σ and reveals the trace of shuffled node indices $t^1, \ldots, t^n$, then $\sigma^{-1}\left(t^{\text{iter}}\right) = u^{\text{iter}}$.

This is shown by induction over the iterations. Let $\overrightarrow{Dist^{\text{iter}}}$ be the lengths before the ArgMin is calculated in the iteration iter of Prot. 9 while using σ to shuffle the nodes. $\overrightarrow{Dist^1} = \sigma\left(\overrightarrow{Dist'^1}\right)$ holds by definition. Eq. (3) then shows that $\sigma^{-1}\left(t^1\right) = u^1$, i. e., that the same node is selected in both the shuffled and the unshuffled protocol execution. Hence, the length updates will behave analogously and $\overrightarrow{Dist^2} = \sigma\left(\overrightarrow{Dist'^2}\right)$. It follows from induction that $\overrightarrow{Dist^{\text{iter}}} = \sigma\left(\overrightarrow{Dist'^{\text{iter}}}\right)$ and $\sigma^{-1}\left(t^{\text{iter}}\right) = u^{\text{iter}}$ for all iterations $1 \leq \text{iter} \leq n$.

$$
\begin{aligned}
&\sigma^{-1}\left(t^{\text{iter}}\right) = \sigma^{-1}\left(\textsc{ArgMin}\left(\overrightarrow{Tmp^{\text{iter}}}\right)\right)\\
=&\sigma^{-1}\left(\textsc{ArgMin}_{1\leq j\leq n}\left((n+1)\cdot\overrightarrow{Dist^{\text{iter}}}_j + \overrightarrow{InvPerm}_j\right)\right)\\
=&\sigma^{-1}\left(\textsc{ArgMin}_{1\leq j\leq n}\left((n+1)\cdot\overrightarrow{Dist^{\text{iter}}}_j + \sigma^{-1}(j)\right)\right) \qquad (3)\\
=&\textsc{ArgMin}_{1\leq i\leq n}\left((n+1)\cdot\overrightarrow{Dist^{\text{iter}}}_{\sigma(i)} + i\right)\\
=&\textsc{ArgMin}_{1\leq i\leq n}\left((n+1)\cdot\overrightarrow{Dist'^{\text{iter}}}_i + i\right) = u^{\text{iter}}
\end{aligned}
$$

In other words, all protocol executions on G, v have the same underlying unshuffled trace $u^1, \ldots, u^n$. It remains to show that all shuffled traces are equally likely. As the unshuffled trace is independent of the used permutation, this is rather simple: The probability that any given shuffled trace $t'^1, \ldots, t'^n$ is revealed equals the probability that the permutation σ maps $u^1, \ldots, u^n$ to $t'^1, \ldots, t'^n$, which is $\frac{1}{n!}$. Note that this argument could not be made if the unshuffled trace depended on the used permutation.

It follows that the revealed trace can be simulated by choosing a uniformly random permutation of $1, \ldots, n$. Hence, Prot. 9 is secure in the ABB model. □

B Additional proofs

In this section, statements from the main part are proven formally. Concretely, App. B.2 first shows that the trick employed in Sec. 5.3 to calculate successors instead of predecessors indeed results in successors. App. B.3 then shows that the concrete protocols presented in Sec. 5.1 produce cycle-free predecessors. Finally, App. B.4 proves that the shuffle-reveal path extraction sub-protocol is secure, i. e., that revealing shuffled node indices does not leak any information.

B.1 $\Pi^{\textsc{PreprocessReveal}}_{\textsc{Demux}}$ is secure

In Sec. 4.2, the novel demultiplex protocol $\Pi^{\textsc{PreprocessReveal}}_{\textsc{Demux}}$ was introduced. It reveals a masked version of the target unit vector index, which is argued to be secure as the masking "effectively forms a one-time-pad". Lemma 1 formally proves this claim.

Lemma 1. *$\Pi^{\textsc{PreprocessReveal}}_{\textsc{Demux}}$ is at least as secure as $\mathcal{F}_{\textsc{Demux}}$ in the ABB-hybrid-model.*

Proof. Let k be the index of the pre-generated unit vector $\overrightarrow{Pre}$, which is uniformly distributed over $\{1, \ldots, l\}$. We claim that the revealed $offset$, which equals $x - k \pmod l$, is uniformly distributed over $\{0, \ldots, l-1\}$:

$$
\begin{aligned}
\Pr(offset = i) &= \sum_{j=1}^{m} \Pr(x = j) \cdot \Pr(k = j - i \pmod l)\\
&= \sum_{j=1}^{m} \Pr(x = j) \cdot \frac{1}{l}\\
&= \frac{1}{l}\left(\sum_{j=1}^{m} \Pr(x = j)\right)\\
&= \frac{1}{l} \cdot 1 = \frac{1}{l}
\end{aligned}
$$

Thus, the simulator uniformly at random samples the $offset$ value from $\{0, \ldots, l-1\}$, which is not distinguishable from the revealed $offset$ of a real protocol execution. □

B.2 Correctness of successor computation

To show that the trick used to calculate successors instead of predecessors in Sec. 5.3 indeed results in the successors of a shortest path, Lemma 2 first notes that every shortest path in the transformed graph corresponds to a shortest path in the original graph.

Lemma 2. *Let $G = (V, E, l)$ be a graph with edge lengths, and $G' = (V, \{(w, v) : (v, w) \in E\}, l')$ with $l'(v, w) = l(w, v)$ be the corresponding transformed graph. Further, let $p' = (v_1, \ldots, v_k)$ be a shortest path from $t \in V$ to $s \in V$ in G'. Then $p = (v_k, \ldots, v_1)$ is a shortest path from s to t in G.*

Proof. First, we note that $l(p') = l(p)$. Next, we assume p is not a shortest path from s to t in G, i. e., that there is an alternative path $q = (w_1, \ldots, w_k)$ in G with $l(q) < l(p)$. Then we note that there is a corresponding path $q' = (w_k, \ldots, w_1)$ from t to s in G' such that $l'(q') < l'(p')$. This contradicts the premise that p' is a shortest path. Hence, p is a shortest path too. □

Note that the corresponding shortest path p in G to every shortest path p' in G' is just p' in reversed order. Hence, the predecessor of a node in p' is equal to the successor of the same node in p.

B.3 The predecessors are cycle-free

In the following, we prove that the predecessors produced by Prots. 2 and 3 do not contain cycles.

Lemma 3. *Let $G = (V, E, l)$ be a graph with edge lengths and without edges of negative length, $s \in V$, and $\overrightarrow{Preds}$ the predecessors produced by running the Dijkstra protocol on G, s. Then, for all reachable $t \in V \setminus \{s\}$, the path from s to t induced by the predecessors does not contain any cycles.*

Proof. Assume that the predecessors induce a cycle $(c_1, \ldots, c_k, c_1)$. It follows that $\overrightarrow{Dist}_{c_k} = \overrightarrow{Dist}_{c_1} + \sum_{i=1}^{k-1} l(c_i, c_{i+1}) \geq \overrightarrow{Dist}_{c_1}$ since all edges have a non-negative length. However, c_k is only set to be

the predecessor of c_1 if $\overrightarrow{Dist}_{c_k} + l(c_k, c_1) < \overrightarrow{Dist}_{c_1}$. As $l(c_k, c_1)$ is non-negative, this contradicts $\overrightarrow{Dist}_{c_k} \geq \overrightarrow{Dist}_{c_1}$. Hence, the predecessors cannot induce a cycle. □

Lemma 4. *Let $G = (V, E, l)$ be a graph with edge lengths and without cycles of negative length, and* **Preds** *the predecessors produced by running the Floyd-Warshall protocol on G. Then, for all $i, j \in V$, $i \neq j$ such that j is reachable from i, the path from i to j induced by the predecessors does not contain any cycles.*

Proof. This is shown by induction over k, i. e., over the protocol's iterations. To start, we note that after the first iteration, i. e., for $k = 1$, the lemma holds since each path produced by **Preds** has at most one step and therefore cannot contain a cycle.

Next, assume that the lemma holds after k iterations of the protocol. By contradiction, we show that the lemma also must hold after $k + 1$ iterations: Assume that there are i, j such that $\mathbf{Dist}_{i,k+1} + \mathbf{Dist}_{k+1,j} < \mathbf{Dist}_{i,j}$, but such that the corresponding path $p = p_{i,k+1} \parallel p_{k+1,j}$ does contain a cycle. This path then can be decomposed into the part prior to the cycle p'_1, the cycle c, and the part after the cycle p'_2, and the cycle-free version of the path can be constructed as $p' = p'_1 \parallel p'_2$.

Since the paths using nodes $\{1, \ldots, k\}$ are cycle-free by the induction assumption, the cycle must contain the node $k + 1$. Further, the node $k+1$ can appear only once in the complete path. Therefore p' only visits nodes in $\{1, \ldots, k\}$. It follows that $l(p') \geq \mathbf{Dist}_{i,j}$.

Finally, $l(c) \geq 0$ since G does not contain cycles of negative length. Therefore, $l(p) = l(p'_1) + l(c) + l(p'_2) \geq l(p') \geq \mathbf{Dist}_{i,j}$, violating the premise $\mathbf{Dist}_{i,k+1} + \mathbf{Dist}_{k+1,j} < \mathbf{Dist}_{i,j}$. It follows that the lemma also must hold after $k + 1$ iterations, and therefore also holds in general. □

B.4 $\Pi^{\text{ShuffleReveal}}_{\text{PathExtract}}$ is secure

In Sec. 5.2, we present $\Pi^{\text{ShuffleReveal}}_{\text{PathExtract}}$ (Prot. 7), which extracts a path from the successor matrix by first adding fake nodes, shuffling all nodes, and then revealing the shuffled indices of the visited nodes. We argue that due to the shuffling of the nodes, the revealed shuffled node indices do not leak information, which we prove formally in Lemma 5.

Lemma 5. *$\Pi^{\text{ShuffleReveal}}_{\text{PathExtract}}$ is secure in the ABB model, assuming the inputs are well-formed, i. e., each row of the successor matrix is either a unit-vector or only contains zeroes,* $\mathbf{Succs}_{i,i} = 0$ *for all* $1 \leq i \leq n$*, and that $\overrightarrow{Start}$ is a unit-vector.*

Proof. First, we note that the control flow of $\Pi^{\text{ShuffleReveal}}_{\text{PathExtract}}$ only depends on the publicly known number of nodes. Hence, its simulation is trivial, and we only need to show that the revealed (shuffled) node indices can be simulated.

We claim that this can be done by uniformly sampling n distinct values from $\{1, \ldots, 2n-1\}$. Note that the probability of the simulator choosing any trace uniformly is $\frac{(n-1)!}{(2n-1)!}$.

Next, let the input to the protocol be well-formed and arbitrary but fixed, and σ be the permutation used during a protocol execution. Further, let $t^1, \ldots, t^n$ be the trace of shuffled node indices revealed by the protocol execution, and $u^{\text{iter}} = \sigma^{-1}(t^{\text{iter}})$ be the corresponding trace of unshuffled node indices. Finally, let $p = (v_1, \ldots, v_k)$ be the path produced by the protocol on the inputs.

We note that for every possible σ the unshuffled trace $u^1, \ldots, u^n$ always is $v_1, \ldots, v_k, n+1, \ldots, 2n-k$, i. e., it always starts with the resulting path followed by the chain of the fake nodes, independently of the used permutation.

This means that for this arbitrary but fixed well-formed input, the probability of any trace $t'^1, \ldots, t'^n$ being revealed equals the probability of the permutation mapping $v_1, \ldots, v_k, n+1, \ldots, 2n-k$ to $t'^1, \ldots, t'^n$. As the protocol chooses the permutation uniformly at random, this probability is $\frac{(n-1)!}{(2n-1)!}$.

Hence, the distribution of simulated traces equals the distribution of the real traces created by the protocol, and the simulated traces are not distinguishable from real protocol traces. □

C Investigation into runtime discrepancies

Sec. 6 reports a runtime discrepancy: In the WAN setting with semi-honest security, $\Pi^{\text{Blanton + A.D.}}_{\text{PredFollow}}$ achieves a speedup of up to a factor of 1.59 over $\Pi^{\text{Blanton}}_{\text{PredFollow}}$ by using $\Pi^{\text{PreprocessReveal}}_{\text{Demux}}$ instead of $\Pi^{\text{ParallelComp}}_{\text{Demux}}$. However, $\Pi^{\text{PreprocessReveal}}_{\text{Demux}}$ only achieves a speedup of up to 1.47 over $\Pi^{\text{ParallelComp}}_{\text{Demux}}$ in the microbenchmarks. This section presents our investigation into this discrepancy.

First, we extract the time $\Pi^{\text{Blanton}}_{\text{PredFollow}}$ and $\Pi^{\text{Blanton + A.D.}}_{\text{PredFollow}}$ spend on demultiplexing using an MP-SPDZ feature called "timers". Such timers can be imagined as stopwatches that can be started and stopped by MP-SPDZ programs. When MP-SPDZ terminates, it reports the total runtime of each timer, as well as the rounds and communication that occurred while the timer was running.

The timer placements in our implementations of $\Pi^{\text{Blanton}}_{\text{PredFollow}}$ and $\Pi^{\text{Blanton + A.D.}}_{\text{PredFollow}}$ are presented by Listing 1. It can be seen that the "demux" timer only captures the demultiplex. We use an analogous timer in the microbenchmarks.

Listing 1 Placement of the MP-SPDZ timers within the main loop of $\Pi^{\text{Blanton}}_{\text{PredFollow}}$ and $\Pi^{\text{Blanton + A.D.}}_{\text{PredFollow}}$. The `"demux"` timer isolates the time spent demultiplexing.

```
@for_range(n_nodes - 1)
def step(i: cint) -> None:
    with Timer("dotprot"):
        next_node = sint.dot_product(current_node,
        ↪ predecessors_with_fake)
    with Timer("demux"):
        demux_result = self.demux.demux(next_node,
        ↪ n_nodes_with_fakes,
        ↪ computation_setting=computation_setting)
    with Timer("copy-unitvector"):
        current_node[:] = demux_result
        almost_steps[n_nodes - i -
        ↪ 2].assign_part_vector(current_node[0:n_nodes])
```

These timers also act as "merge-barriers" during the MP-SPDZ compilation process, i. e., the compiler will not move operations across timer boundaries even if such moves could reduce the number of rounds. This ensures that, for example, the prior dot-product will not be counted towards the demux time. We have confirmed this manually by disassembling compiled $\Pi^{\text{Blanton + A.D.}}_{\text{PredFollow}}$-evaluation

and microbenchmark programs. Except for register indices, the bytecodes in the "demux" timers are equivalent.

Figs. 11 and 12 present the resulting average rounds, communication and runtimes per demultiplex extracted from $\Pi^{\text{Blanton}}_{\text{PredFollow}}$ and $\Pi^{\text{Blanton + A.D.}}_{\text{PredFollow}}$, which we will refer to as macrobenchmarks from now on, and the corresponding data from the microbenchmarks. The predecessor-following protocols add fake nodes, leading to the unit vector size being twice the input graph's number of nodes. This results in some data points that were evaluated only in the microbenchmarks, e. g., the unit vector size 300, because the predecessor-following protocols were evaluated on numbers of nodes that are multiples of 100.

It can be seen that $\Pi^{\text{ParallelComp}}_{\text{Demux}}$ consistently is slower in the macrobenchmarks than in the microbenchmarks, while $\Pi^{\text{PreprocessReveal}}_{\text{Demux}}$ is faster in the macrobenchmarks than in the microbenchmarks. This confirms that the runtime discrepancy reported in Sec. 6 indeed stems from differences in demultiplex runtimes. Further, the rounds and communication of both protocols are consistent across micro- and macrobenchmarks. This already suggests that the root cause of the discrepancies lies in local factors.

We further ensure that this runtime discrepancy is not caused by other protocol-global states. Particularly, we use the comparison primitive of [54], resulting in all protocols not needing any preprocessed material in the semi-honest setting. MP-SPDZ confirms this empirically by reporting zero communication during the offline phase. The semi-honest security primitive also does not incorporate any MAC-checks, and (potentially delayed) MAC checks thus cannot cause the discrepancy.

Finally, we conducted a review of MP-SPDZ's primitive implementations to ensure that no (accidental) preprocessing, MAC checks, or persistence across executions influences the runtime of the protocols. Thus, we conclude that the runtime discrepancy is not caused by differences in how MP-SPDZ executes the demultiplex protocols, but rather by local computation factors such as CPU caching, speculative execution, and OS process scheduling. We consider controlling for such factors out-of-scope for this paper, and therefore only claim that the microbenchmarks allow for good sub-protocol selections, but not necessarily the best possible.

D Numerical runtimes

This section presents the numerical results of our evaluation, complementing the figures in Sec. 6. All runtimes are given in seconds, and the fastest (sub-)protocol is highlighted using a light-gray background for each computation environment and input size. Tables 3 to 10 present the runtimes, rounds, and communication of the predecessor-calculation, comparing $\Pi^{\text{Dijkstra}}_{\text{PredCalc}}$ and $\Pi^{\text{FloydWarshall}}_{\text{PredCalc}}$. Tables 11 to 26 present the runtimes, rounds, and communication of single and parallel demultiplexes. They are followed by the runtimes, rounds, and communication of the path-extraction protocols in Tables 27 to 34, and by the runtimes of the predecessor-following protocols in Tables 35 to 40. The runtimes, rounds, and communication of the $\Pi_{\text{PaddingReorder}}$ and $\Pi_{\text{GraphTransform}}$ follow in Tables 41 to 48. Finally, Tables 49 to 54 compare the start-to-end runtimes of $\Pi^{\text{Baseline}}_{\text{SPSP}}$ and $\Pi^{\text{Optimized}}_{\text{SPSP}}$ in all evaluated settings.

Table 3: Runtimes (in seconds) of the successor calculation with semi-honest security in the unrestricted network setting.

n	Dijkstra	Floyd-Warshall
10	0.0485	0.0309
20	0.1168	0.1101
30	0.2055	0.2791
40	0.2922	0.4548
50	0.4358	0.8564
60	0.4909	1.4671
70	0.6617	1.7994
80	0.7641	2.7715
90	0.8701	3.9207
100	1.0799	4.8469
200	2.3696	38.1775
300	4.0381	110.9950
400	6.4356	276.3990
500	8.3028	562.6737
600	10.1938	979.0567
700	14.9292	1 337.3200
800	17.3745	2 034.7367
900	20.6398	2 968.4467
1000	23.3226	4 040.2333

Table 4: Runtimes (in seconds) of the successor calculation with semi-honest security in the LAN setting.

n	Dijkstra	Floyd-Warshall
10	0.4297	0.1583
20	1.0670	0.3656
30	1.8672	0.7123
40	2.7007	1.2278
50	3.5685	1.7244
60	4.4055	2.2858
70	5.5406	3.1238
80	6.5148	4.3393
90	7.1094	5.3835
100	8.3893	6.9898
200	19.7094	45.4031
300	30.7837	126.9567
400	44.4699	302.1893
500	58.9112	629.6777
600	73.2229	934.0387
700	88.0245	1 532.0333
800	102.7660	2 212.3067
900	119.7690	3 212.4267
1000	134.1920	4 694.0333

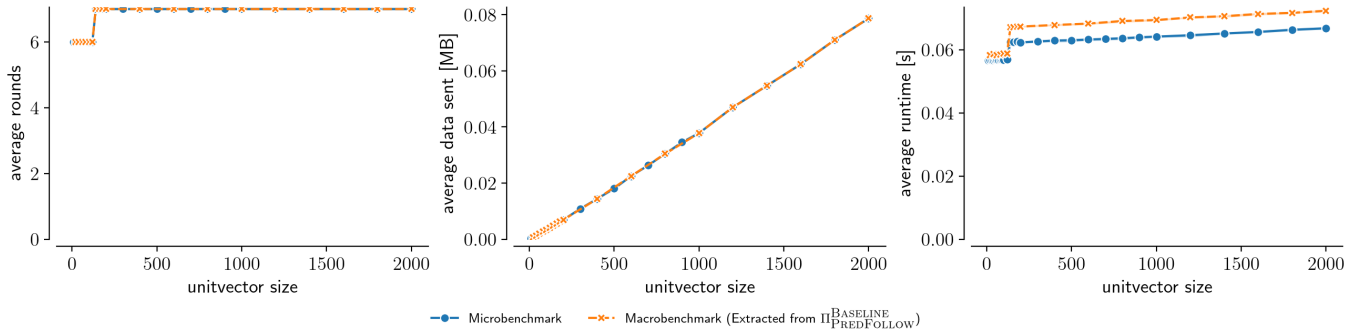


Figure 11: Rounds, communication, and runtime of $\Pi^{\text{ParallelComp}}_{\text{Demux}}$ in micro- and macrobenchmarks in the WAN setting with semi-honest security.

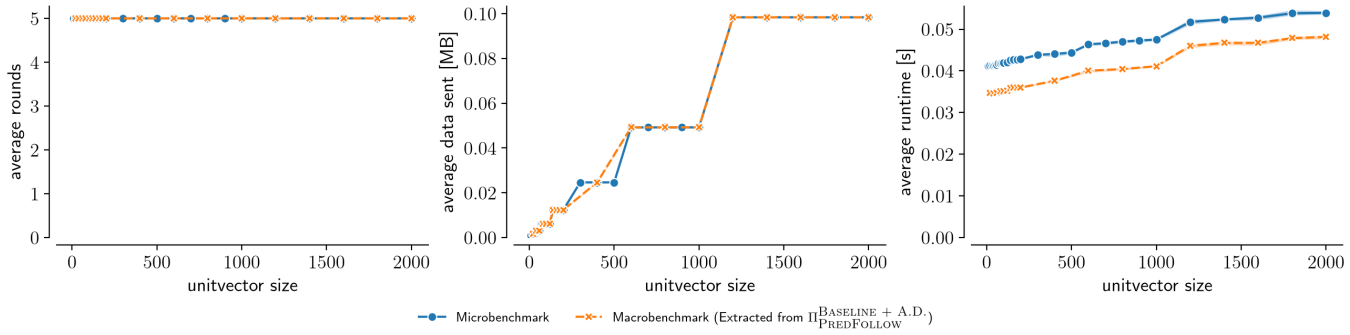


Figure 12: Rounds, communication, and runtime of $\Pi^{\text{PreprocessReveal}}_{\text{Demux}}$ in micro- and macrobenchmarks in the WAN setting with semi-honest security.

Table 5: Runtimes (in seconds) of the successor calculation with semi-honest security in the WAN setting.

n	Dijkstra	Floyd-Warshall
10	3.4378	1.1395
20	8.9292	2.3272
30	15.1907	3.7192
40	22.0999	5.4306
50	29.4680	7.4586
60	36.7172	9.7182
70	44.5238	12.3163
80	52.8197	15.4310
90	61.1662	18.7084
100	69.3379	23.2787
200	160.1597	116.8523
300	258.5653	339.7900
400	363.4237	756.4330
500	469.0657	1 443.3200
600	583.6500	2 396.2333
700	694.8563	3 864.4067
800	811.6690	5 761.7000
900	934.8913	8 680.5717
1000	1 051.5317	11 519.8000

Table 6: Runtimes (in seconds) of the successor calculation with malicious security in the unrestricted network setting.

n	Dijkstra	Floyd-Warshall
10	0.0855	0.0294
20	0.2330	0.1118
30	0.3709	0.2760
40	0.5420	0.4217
50	0.7264	0.9242
60	0.8766	1.5986
70	1.0619	2.2352
80	1.3362	3.6877
90	1.3587	4.5401
100	1.6385	6.5675
200	4.4099	89.9437
300	6.6336	350.9093
400	10.3992	723.5740
500	12.6474	1 511.4133
600	16.2080	2 608.0533
700	20.6969	4 048.9467
800	22.8520	6 412.3600
900	28.8222	8 669.6733
1000	31.7020	11 615.2000

Table 7: Runtimes (in seconds) of the successor calculation with malicious security in the LAN setting.

n	Dijkstra	Floyd-Warshall
10	0.7266	0.1607
20	1.7285	0.4087
30	2.8315	0.8044
40	4.0847	1.3813
50	5.3252	2.0626
60	6.5110	2.8974
70	7.7513	4.2309
80	9.0660	5.5229
90	10.3956	6.7849
100	11.9237	11.6066
200	25.9629	124.1040
300	41.9965	456.9003
400	59.6963	1 007.9200
500	78.9210	1 943.1600
600	96.4658	3 416.1900
700	118.3703	5 332.2217
800	137.4027	7 932.3133
900	158.1280	11 500.5000
1000	179.3390	14 960.3000

Table 8: Runtimes (in seconds) of the successor calculation with malicious security in the WAN setting.

n	Dijkstra	Floyd-Warshall
10	5.8059	1.2055
20	13.9935	2.4872
30	22.9270	4.1025
40	32.5891	6.0241
50	42.5515	8.5359
60	52.6591	11.3667
70	63.1464	14.4768
80	74.0307	19.1314
90	85.1225	23.8635
100	96.0846	34.0149
200	214.2937	388.1250
300	338.1157	1 600.4600
400	473.1400	3 746.0700
500	607.0847	7 188.5733
600	752.4683	12 459.5333
700	900.2523	19 652.4333
800	1 048.1333	29 255.2333
900	1 200.4533	41 987.4500
1000	1 352.4233	57 117.6333

Table 9: Rounds and total communication (in megabytes) of the successor calculation with semi-honest security.

n	Dijkstra		Floyd-Warshall	
	Rounds	Comm	Rounds	Comm
10	345	0.04	111	0.17
20	885	0.15	211	1.30
30	1 495	0.32	311	4.56
40	2 175	0.55	411	10.42
50	2 885	0.89	511	20.62
60	3 595	1.26	611	35.42
70	4 355	1.69	711	56.00
80	5 165	2.20	811	83.33
90	5 975	2.83	911	118.88
100	6 785	3.47	1 011	163.19
200	15 595	13.60	2 011	1 302.01
300	25 125	30.58	3 011	4 395.57
400	35 225	54.06	4 011	10 416.03
500	45 325	84.48	5 011	20 345.94
600	56 295	121.27	6 011	35 154.00
700	67 395	165.21	7 011	55 826.40
800	78 495	217.61	8 011	83 328.00
900	89 595	277.21	9 011	118 648.80
1000	100 695	338.15	10 011	162 750.00

Table 10: Rounds and total communication (in megabytes) of the successor calculation with malicious security.

n	Dijkstra		Floyd-Warshall	
	Rounds	Comm	Rounds	Comm
10	562	0.73	113	0.40
20	1 342	1.86	213	2.90
30	2 192	3.17	313	9.47
40	3 112	4.66	413	21.65
50	4 062	6.30	513	42.81
60	5 012	8.05	613	73.54
70	6 012	10.77	713	116.28
80	7 062	12.96	836	173.06
90	8 112	15.25	936	246.85
100	9 162	17.65	1 113	429.93
200	20 372	53.75	2 735	8 838.23
300	32 302	103.71	6 358	38 523.00
400	44 802	173.38	11 486	91 359.80
500	57 302	253.42	18 677	178 226.80
600	70 672	357.29	25 594	307 317.00
700	84 172	469.56	34 847	488 468.00
800	97 695	605.21	67 636	730 879.00
900	111 249	868.19	72 903	1 039 515.00
1000	124 772	1 051.29	90 459	1 425 746.00

Table 11: Runtimes of a single demux with semi-honest security in the unrestricted network setting.

n	$\Pi^{\text{Keller}}_{\text{Demux}}$	$\Pi^{\text{Keller-NB}}_{\text{Demux}}$	$\Pi^{\text{Launchbury}}_{\text{Demux}}$	$\Pi^{\text{ParallelComp}}_{\text{Demux}}$	$\Pi^{\text{PreprocessReveal}}_{\text{Demux}}$
10	0.0054	0.0017	0.0015	0.0005	0.0007
20	0.0089	0.0022	0.0019	0.0008	0.0008
30	0.0090	0.0022	0.0019	0.0008	0.0008
40	0.0153	0.0025	0.0022	0.0008	0.0008
50	0.0151	0.0024	0.0022	0.0009	0.0009
60	0.0153	0.0025	0.0022	0.0009	0.0009
70	0.0156	0.0029	0.0025	0.0008	0.0011
80	0.0158	0.0028	0.0025	0.0009	0.0011
90	0.0160	0.0029	0.0025	0.0008	0.0012
100	0.0157	0.0029	0.0024	0.0009	0.0012
120	0.0155	0.0029	0.0024	0.0009	0.0015
140	0.0168	0.0033	0.0027	0.0010	0.0014
160	0.0160	0.0032	0.0027	0.0011	0.0016
180	0.0164	0.0033	0.0028	0.0011	0.0016
200	0.0162	0.0034	0.0027	0.0011	0.0015
300	0.0168	0.0038	0.0031	0.0012	0.0019
400	0.0169	0.0039	0.0033	0.0012	0.0020
500	0.0169	0.0039	0.0034	0.0013	0.0025
600	0.0174	0.0042	0.0036	0.0014	0.0029
700	0.0176	0.0045	0.0036	0.0013	0.0030
800	0.0174	0.0044	0.0038	0.0015	0.0037
900	0.0173	0.0046	0.0039	0.0015	0.0033
1000	0.0175	0.0045	0.0038	0.0015	0.0036
1200	0.0182	0.0052	0.0044	0.0015	0.0046
1400	0.0181	0.0049	0.0045	0.0018	0.0046
1600	0.0180	0.0050	0.0045	0.0018	0.0043
1800	0.0187	0.0053	0.0045	0.0018	0.0050
2000	0.0195	0.0054	0.0048	0.0018	0.0059

Table 12: Runtimes of a single demux with semi-honest security in the LAN setting.

n	$\Pi^{\text{Keller}}_{\text{Demux}}$	$\Pi^{\text{Keller-NB}}_{\text{Demux}}$	$\Pi^{\text{Launchbury}}_{\text{Demux}}$	$\Pi^{\text{ParallelComp}}_{\text{Demux}}$	$\Pi^{\text{PreprocessReveal}}_{\text{Demux}}$
10	0.0544	0.0167	0.0133	0.0068	0.0049
20	0.0933	0.0213	0.0182	0.0067	0.0051
30	0.0905	0.0216	0.0183	0.0068	0.0050
40	0.1598	0.0239	0.0203	0.0066	0.0050
50	0.1621	0.0238	0.0197	0.0067	0.0051
60	0.1575	0.0241	0.0200	0.0069	0.0049
70	0.1613	0.0292	0.0227	0.0068	0.0053
80	0.1621	0.0284	0.0226	0.0067	0.0053
90	0.1641	0.0291	0.0234	0.0066	0.0054
100	0.1604	0.0293	0.0229	0.0067	0.0054
120	0.1655	0.0278	0.0228	0.0068	0.0055
140	0.1616	0.0302	0.0245	0.0075	0.0056
160	0.1626	0.0317	0.0239	0.0080	0.0059
180	0.1634	0.0317	0.0250	0.0076	0.0059
200	0.1643	0.0319	0.0241	0.0075	0.0058
300	0.1741	0.0360	0.0295	0.0079	0.0064
400	0.1730	0.0361	0.0294	0.0076	0.0068
500	0.1700	0.0364	0.0297	0.0077	0.0071
600	0.1749	0.0387	0.0316	0.0077	0.0080
700	0.1737	0.0368	0.0313	0.0080	0.0081
800	0.1747	0.0392	0.0313	0.0079	0.0085
900	0.1739	0.0389	0.0316	0.0078	0.0087
1000	0.1752	0.0388	0.0317	0.0083	0.0088
1200	0.1781	0.0420	0.0344	0.0082	0.0109
1400	0.1752	0.0417	0.0340	0.0084	0.0116
1600	0.1774	0.0415	0.0338	0.0086	0.0119
1800	0.1821	0.0419	0.0338	0.0088	0.0125
2000	0.1760	0.0420	0.0339	0.0089	0.0128

Table 13: Runtimes of a single demux with semi-honest security in the WAN setting.

n	$\Pi^{\text{Keller}}_{\text{Demux}}$	$\Pi^{\text{Keller-NB}}_{\text{Demux}}$	$\Pi^{\text{Launchbury}}_{\text{Demux}}$	$\Pi^{\text{ParallelComp}}_{\text{Demux}}$	$\Pi^{\text{PreprocessReveal}}_{\text{Demux}}$
10	0.4430	0.1338	0.1083	0.0567	0.0412
20	0.7639	0.1754	0.1498	0.0568	0.0413
30	0.7634	0.1754	0.1496	0.0566	0.0413
40	1.3105	0.1960	0.1653	0.0566	0.0413
50	1.3103	0.1961	0.1653	0.0568	0.0415
60	1.3106	0.1960	0.1651	0.0568	0.0415
70	1.3356	0.2377	0.1858	0.0567	0.0418
80	1.3358	0.2376	0.1858	0.0568	0.0419
90	1.3354	0.2374	0.1862	0.0568	0.0420
100	1.3357	0.2372	0.1858	0.0568	0.0420
120	1.3382	0.2369	0.1860	0.0570	0.0421
140	1.3761	0.2576	0.2020	0.0624	0.0426
160	1.3736	0.2582	0.2017	0.0624	0.0427
180	1.3754	0.2584	0.2020	0.0627	0.0427
200	1.3733	0.2580	0.2013	0.0623	0.0428
300	1.4128	0.2999	0.2431	0.0626	0.0439
400	1.4131	0.2993	0.2428	0.0630	0.0441
500	1.4135	0.2995	0.2431	0.0631	0.0444
600	1.4338	0.3204	0.2592	0.0632	0.0464
700	1.4346	0.3206	0.2593	0.0634	0.0466
800	1.4345	0.3204	0.2586	0.0636	0.0470
900	1.4335	0.3206	0.2587	0.0639	0.0473
1000	1.4301	0.3210	0.2588	0.0642	0.0476
1200	1.4499	0.3428	0.2755	0.0647	0.0517
1400	1.4519	0.3426	0.2752	0.0652	0.0524
1600	1.4621	0.3428	0.2751	0.0657	0.0527
1800	1.4581	0.3430	0.2756	0.0663	0.0537
2000	1.4591	0.3426	0.2755	0.0668	0.0539

Table 14: Rounds and total communication (in megabytes) of a single demux with semi-honest security.

n	$\Pi^{\text{Keller}}_{\text{Demux}}$		$\Pi^{\text{Keller-NB}}_{\text{Demux}}$		$\Pi^{\text{Launchbury}}_{\text{Demux}}$		$\Pi^{\text{ParallelComp}}_{\text{Demux}}$		$\Pi^{\text{PreprocessReveal}}_{\text{Demux}}$	
	Rounds	Comm	Rounds	Comm	Rounds	Comm	Rounds	Comm	Rounds	Comm
10	53	0.001	15	0.000	13	0.001	6	0.000	5	0.001
20	92	0.001	19	0.001	17	0.001	6	0.001	5	0.002
30	92	0.001	19	0.001	17	0.001	6	0.001	5	0.002
40	162	0.002	22	0.002	19	0.002	6	0.001	5	0.003
50	162	0.002	22	0.002	19	0.002	6	0.002	5	0.003
60	162	0.002	22	0.002	19	0.002	6	0.002	5	0.003
70	166	0.004	26	0.003	22	0.004	6	0.002	5	0.006
80	166	0.004	26	0.003	22	0.004	6	0.003	5	0.006
90	166	0.004	26	0.003	22	0.004	6	0.003	5	0.006
100	166	0.004	26	0.003	22	0.004	6	0.003	5	0.006
120	166	0.004	26	0.003	22	0.004	6	0.004	5	0.006
140	169	0.007	29	0.006	24	0.008	7	0.005	5	0.012
160	169	0.007	29	0.006	24	0.008	7	0.006	5	0.012
180	169	0.007	29	0.006	24	0.008	7	0.006	5	0.012
200	169	0.007	29	0.006	24	0.008	7	0.007	5	0.012
300	173	0.010	33	0.013	28	0.014	7	0.011	5	0.025
400	173	0.014	33	0.013	28	0.014	7	0.014	5	0.025
500	173	0.014	33	0.013	28	0.014	7	0.018	5	0.025
600	176	0.019	36	0.025	30	0.027	7	0.022	5	0.049
700	176	0.021	36	0.025	30	0.027	7	0.026	5	0.049
800	176	0.024	36	0.025	30	0.027	7	0.030	5	0.049
900	176	0.027	36	0.025	30	0.027	7	0.035	5	0.049
1000	176	0.027	36	0.025	30	0.027	7	0.038	5	0.049
1200	179	0.034	39	0.049	32	0.053	7	0.047	5	0.098
1400	179	0.039	39	0.049	32	0.053	7	0.055	5	0.098
1600	179	0.044	39	0.049	32	0.053	7	0.062	5	0.098
1800	179	0.050	39	0.049	32	0.053	7	0.071	5	0.098
2000	179	0.053	39	0.049	32	0.053	7	0.079	5	0.098

Table 15: Runtimes of a single demux with malicious security in the unrestricted network setting.

n	$\Pi^{\text{Keller}}_{\text{Demux}}$	$\Pi^{\text{Keller-NB}}_{\text{Demux}}$	$\Pi^{\text{Launchbury}}_{\text{Demux}}$	$\Pi^{\text{ParallelComp}}_{\text{Demux}}$	$\Pi^{\text{PreprocessReveal}}_{\text{Demux}}$
10	0.0031	0.0020	0.0016	0.0012	0.0057
20	0.0041	0.0023	0.0021	0.0012	0.0060
30	0.0039	0.0024	0.0021	0.0012	0.0059
40	0.0047	0.0021	0.0018	0.0012	0.0059
50	0.0047	0.0022	0.0017	0.0013	0.0059
60	0.0047	0.0021	0.0017	0.0014	0.0059
70	0.0052	0.0026	0.0020	0.0015	0.0059
80	0.0057	0.0026	0.0020	0.0015	0.0060
90	0.0054	0.0026	0.0020	0.0015	0.0059
100	0.0052	0.0026	0.0020	0.0016	0.0062
120	0.0053	0.0026	0.0020	0.0017	0.0061
140	0.0059	0.0027	0.0021	0.0018	0.0064
160	0.0059	0.0026	0.0020	0.0020	0.0066
180	0.0058	0.0026	0.0020	0.0022	0.0069
200	0.0058	0.0026	0.0020	0.0023	0.0065
300	0.0064	0.0031	0.0025	0.0026	0.0086
400	0.0066	0.0031	0.0025	0.0028	0.0084
500	0.0066	0.0031	0.0025	0.0034	0.0089
600	0.0075	0.0035	0.0027	0.0038	0.0114
700	0.0073	0.0035	0.0028	0.0043	0.0120
800	0.0076	0.0034	0.0027	0.0046	0.0129
900	0.0080	0.0035	0.0028	0.0050	0.0123
1000	0.0078	0.0035	0.0028	0.0051	0.0127
1200	0.0085	0.0042	0.0033	0.0062	0.0162
1400	0.0093	0.0042	0.0033	0.0071	0.0153
1600	0.0093	0.0042	0.0033	0.0076	0.0167
1800	0.0098	0.0044	0.0034	0.0079	0.0161
2000	0.0101	0.0043	0.0033	0.0095	0.0160

Table 16: Runtimes of a single demux with malicious security in the LAN setting.

n	$\Pi^{\text{Keller}}_{\text{Demux}}$	$\Pi^{\text{Keller-NB}}_{\text{Demux}}$	$\Pi^{\text{Launchbury}}_{\text{Demux}}$	$\Pi^{\text{ParallelComp}}_{\text{Demux}}$	$\Pi^{\text{PreprocessReveal}}_{\text{Demux}}$
10	0.0212	0.0114	0.0091	0.0076	0.0389
20	0.0280	0.0141	0.0116	0.0076	0.0389
30	0.0279	0.0140	0.0115	0.0078	0.0392
40	0.0336	0.0149	0.0114	0.0078	0.0388
50	0.0335	0.0149	0.0115	0.0080	0.0390
60	0.0337	0.0150	0.0116	0.0079	0.0389
70	0.0363	0.0178	0.0129	0.0080	0.0389
80	0.0360	0.0174	0.0130	0.0083	0.0391
90	0.0358	0.0177	0.0125	0.0081	0.0399
100	0.0365	0.0177	0.0125	0.0082	0.0394
120	0.0367	0.0170	0.0128	0.0086	0.0387
140	0.0388	0.0186	0.0125	0.0097	0.0393
160	0.0373	0.0185	0.0131	0.0102	0.0413
180	0.0384	0.0184	0.0127	0.0100	0.0394
200	0.0378	0.0191	0.0130	0.0104	0.0400
300	0.0404	0.0220	0.0156	0.0107	0.0423
400	0.0417	0.0214	0.0157	0.0114	0.0424
500	0.0417	0.0212	0.0153	0.0119	0.0425
600	0.0441	0.0235	0.0159	0.0123	0.0467
700	0.0432	0.0237	0.0160	0.0132	0.0462
800	0.0428	0.0236	0.0162	0.0136	0.0474
900	0.0437	0.0233	0.0162	0.0142	0.0478
1000	0.0445	0.0235	0.0160	0.0149	0.0475
1200	0.0452	0.0255	0.0169	0.0158	0.0531
1400	0.0473	0.0257	0.0169	0.0170	0.0532
1600	0.0478	0.0258	0.0169	0.0183	0.0536
1800	0.0482	0.0254	0.0169	0.0187	0.0549
2000	0.0485	0.0261	0.0171	0.0197	0.0559

Table 17: Runtimes of a single demux with malicious security in the WAN setting.

n	$\Pi^{\text{Keller}}_{\text{Demux}}$	$\Pi^{\text{Keller-NB}}_{\text{Demux}}$	$\Pi^{\text{Launchbury}}_{\text{Demux}}$	$\Pi^{\text{ParallelComp}}_{\text{Demux}}$	$\Pi^{\text{PreprocessReveal}}_{\text{Demux}}$
10	0.1785	0.0955	0.0749	0.0642	0.3213
20	0.2306	0.1161	0.0955	0.0644	0.3219
30	0.2305	0.1161	0.0956	0.0647	0.3223
40	0.2728	0.1263	0.0958	0.0650	0.3219
50	0.2729	0.1265	0.0957	0.0652	0.3217
60	0.2730	0.1265	0.0956	0.0652	0.3222
70	0.2943	0.1474	0.1065	0.0658	0.3230
80	0.2939	0.1476	0.1064	0.0659	0.3234
90	0.2947	0.1475	0.1064	0.0661	0.3235
100	0.2945	0.1474	0.1066	0.0663	0.3236
120	0.2944	0.1473	0.1065	0.0670	0.3240
140	0.3064	0.1584	0.1069	0.0777	0.3253
160	0.3067	0.1584	0.1072	0.0778	0.3251
180	0.3062	0.1583	0.1070	0.0785	0.3252
200	0.3059	0.1584	0.1071	0.0771	0.3245
300	0.3282	0.1804	0.1286	0.0805	0.3290
400	0.3292	0.1802	0.1285	0.0823	0.3296
500	0.3294	0.1801	0.1284	0.0842	0.3289
600	0.3427	0.1926	0.1301	0.0863	0.3360
700	0.3427	0.1926	0.1303	0.0882	0.3365
800	0.3437	0.1925	0.1303	0.0898	0.3365
900	0.3448	0.1925	0.1302	0.0920	0.3372
1000	0.3451	0.1927	0.1302	0.0934	0.3376
1200	0.3571	0.2067	0.1333	0.0967	0.3492
1400	0.3589	0.2069	0.1333	0.1002	0.3495
1600	0.3610	0.2067	0.1333	0.1049	0.3503
1800	0.3628	0.2068	0.1333	0.1071	0.3505
2000	0.3649	0.2069	0.1333	0.1107	0.3511

Table 18: Rounds and total communication (in megabytes) of a single demux with malicious security.

n	$\Pi^{\text{Keller}}_{\text{Demux}}$		$\Pi^{\text{Keller-NB}}_{\text{Demux}}$		$\Pi^{\text{Launchbury}}_{\text{Demux}}$		$\Pi^{\text{ParallelComp}}_{\text{Demux}}$		$\Pi^{\text{PreprocessReveal}}_{\text{Demux}}$	
	Rounds	Comm	Rounds	Comm	Rounds	Comm	Rounds	Comm	Rounds	Comm
10	17	0.011	9	0.006	7	0.007	6	0.006	32	0.036
20	22	0.018	11	0.007	9	0.009	6	0.008	32	0.038
30	22	0.018	11	0.007	9	0.009	6	0.010	32	0.038
40	26	0.031	12	0.010	9	0.012	6	0.013	32	0.043
50	26	0.031	12	0.010	9	0.012	6	0.015	32	0.043
60	26	0.031	12	0.010	9	0.012	6	0.018	32	0.043
70	28	0.043	14	0.016	10	0.019	6	0.024	32	0.053
80	28	0.043	14	0.016	10	0.019	6	0.027	32	0.053
90	28	0.043	14	0.016	10	0.019	6	0.030	32	0.053
100	28	0.043	14	0.016	10	0.019	6	0.033	32	0.053
120	28	0.043	14	0.016	10	0.019	6	0.039	32	0.053
140	29	0.069	15	0.026	10	0.030	7	0.049	32	0.073
160	29	0.069	15	0.026	10	0.030	7	0.055	32	0.073
180	29	0.069	15	0.026	10	0.030	7	0.062	32	0.073
200	29	0.069	15	0.026	10	0.030	7	0.069	32	0.073
300	31	0.096	17	0.047	12	0.052	7	0.099	32	0.113
400	31	0.121	17	0.047	12	0.052	7	0.132	32	0.113
500	31	0.121	17	0.047	12	0.052	7	0.164	32	0.113
600	32	0.161	18	0.087	12	0.094	7	0.208	32	0.193
700	32	0.174	18	0.087	12	0.094	7	0.241	32	0.193
800	32	0.200	18	0.087	12	0.094	7	0.277	32	0.193
900	32	0.225	18	0.087	12	0.094	7	0.312	32	0.193
1000	32	0.225	18	0.087	12	0.094	7	0.345	32	0.193
1200	33	0.279	19	0.167	12	0.178	7	0.436	32	0.353
1400	33	0.317	19	0.167	12	0.178	7	0.508	32	0.353
1600	33	0.349	19	0.167	12	0.178	7	0.580	32	0.353
1800	33	0.399	19	0.167	12	0.178	7	0.654	32	0.353
2000	33	0.424	19	0.167	12	0.178	7	0.726	32	0.353

Table 19: Runtimes of n parallel demuxes with semi-honest security in the unrestricted network setting.

n	$\Pi^{\text{Keller}}_{\text{Demux}}$	$\Pi^{\text{Keller-NB}}_{\text{Demux}}$	$\Pi^{\text{Launchbury}}_{\text{Demux}}$	$\Pi^{\text{ParallelComp}}_{\text{Demux}}$	$\Pi^{\text{PreprocessReveal}}_{\text{Demux}}$
10	0.0055	0.0018	0.0017	0.0010	0.0021
20	0.0096	0.0025	0.0025	0.0013	0.0036
30	0.0101	0.0028	0.0028	0.0016	0.0051
40	0.0168	0.0036	0.0035	0.0017	0.0085
50	0.0173	0.0037	0.0036	0.0023	0.0126
60	0.0173	0.0038	0.0040	0.0023	0.0153
70	0.0191	0.0050	0.0052	0.0027	0.0190
80	0.0192	0.0048	0.0054	0.0035	0.0195
90	0.0192	0.0055	0.0054	0.0032	0.0258
100	0.0193	0.0057	0.0055	0.0046	0.0343
200	0.0266	0.0124	0.0121	0.0135	0.1417
300	0.0381	0.0186	0.0263	0.0268	0.2493
400	0.0554	0.0345	0.0334	0.0404	0.6077
500	0.0711	0.0421	0.0415	0.0618	0.6199
600	0.0956	0.0771	0.0749	0.1091	1.1670
700	0.1217	0.0868	0.0925	0.1174	1.4292
800	0.1446	0.1079	0.0979	0.1790	1.6393
900	0.1844	0.1198	0.1149	0.2133	2.1172
1000	0.2062	0.1410	0.1261	0.2507	2.3050

Table 20: Runtimes of n parallel demuxes with semi-honest security in the LAN setting.

n	$\Pi^{\text{Keller}}_{\text{Demux}}$	$\Pi^{\text{Keller-NB}}_{\text{Demux}}$	$\Pi^{\text{Launchbury}}_{\text{Demux}}$	$\Pi^{\text{ParallelComp}}_{\text{Demux}}$	$\Pi^{\text{PreprocessReveal}}_{\text{Demux}}$
10	0.0526	0.0160	0.0140	0.0066	0.0088
20	0.0920	0.0208	0.0187	0.0072	0.0112
30	0.0898	0.0212	0.0190	0.0074	0.0148
40	0.1546	0.0241	0.0215	0.0081	0.0206
50	0.1544	0.0244	0.0219	0.0086	0.0239
60	0.1585	0.0246	0.0227	0.0091	0.0256
70	0.1613	0.0307	0.0273	0.0100	0.0307
80	0.1644	0.0307	0.0281	0.0104	0.0353
90	0.1631	0.0314	0.0285	0.0116	0.0387
100	0.1663	0.0317	0.0290	0.0121	0.0479
200	0.1813	0.0412	0.0373	0.0223	0.1555
300	0.2081	0.0626	0.0604	0.0437	0.2901
400	0.2368	0.0758	0.0739	0.0661	0.6155
500	0.2553	0.0873	0.0864	0.1013	0.7704
600	0.2980	0.1457	0.1454	0.1463	1.2413
700	0.3286	0.1601	0.1586	0.2014	1.2414
800	0.3716	0.1981	0.1777	0.2308	1.9447
900	0.4209	0.2037	0.2011	0.3075	2.2294
1000	0.4537	0.2367	0.2179	0.4049	2.3224

Table 21: Runtimes of n parallel demuxes with semi-honest security in the WAN setting.

n	$\Pi^{\text{Keller}}_{\text{Demux}}$	$\Pi^{\text{Keller-NB}}_{\text{Demux}}$	$\Pi^{\text{Launchbury}}_{\text{Demux}}$	$\Pi^{\text{ParallelComp}}_{\text{Demux}}$	$\Pi^{\text{PreprocessReveal}}_{\text{Demux}}$
10	0.4422	0.1336	0.1189	0.0570	0.0634
20	0.7669	0.1752	0.1558	0.0577	0.0672
30	0.7679	0.1764	0.1568	0.0585	0.0713
40	1.3118	0.1982	0.1749	0.0598	0.0824
50	1.3146	0.1989	0.1763	0.0620	0.0899
60	1.3165	0.1999	0.1774	0.0640	0.0973
70	1.3717	0.2470	0.2195	0.0668	0.1183
80	1.3741	0.2484	0.2211	0.0695	0.1266
90	1.3802	0.2502	0.2234	0.0723	0.1349
100	1.3787	0.2513	0.2249	0.0751	0.1440
200	1.4922	0.3060	0.2702	0.1337	0.3138
300	1.6765	0.4356	0.4062	0.2343	0.6754
400	1.8094	0.4879	0.4900	0.3640	0.9950
500	1.8895	0.5426	0.5284	0.5289	1.3680
600	2.1173	0.8337	0.8505	0.7514	2.3573
700	2.2618	0.9282	1.0135	0.9520	2.8667
800	2.5079	1.0168	1.0691	1.2520	3.6666
900	2.7502	1.1054	1.1989	1.5528	4.3170
1000	2.8729	1.1950	1.3071	1.9359	4.9950

Table 22: Rounds and total communication (in megabytes) of n parallel demuxes with semi-honest security.

n	$\Pi^{\text{Keller}}_{\text{Demux}}$		$\Pi^{\text{Keller-NB}}_{\text{Demux}}$		$\Pi^{\text{Launchbury}}_{\text{Demux}}$		$\Pi^{\text{ParallelComp}}_{\text{Demux}}$		$\Pi^{\text{PreprocessReveal}}_{\text{Demux}}$	
	Rounds	Comm	Rounds	Comm	Rounds	Comm	Rounds	Comm	Rounds	Comm
10	54	0.006	15	0.005	14	0.009	6	0.003	6	0.008
20	93	0.021	19	0.017	18	0.030	6	0.012	6	0.032
30	93	0.032	19	0.026	18	0.046	6	0.028	6	0.048
40	163	0.077	22	0.066	20	0.102	6	0.051	6	0.125
50	163	0.097	22	0.083	20	0.128	6	0.080	6	0.156
60	163	0.116	22	0.100	20	0.153	6	0.115	6	0.187
70	167	0.253	26	0.226	23	0.311	6	0.164	6	0.433
80	167	0.288	26	0.258	23	0.356	6	0.214	6	0.495
90	167	0.325	26	0.290	23	0.400	6	0.271	6	0.557
100	167	0.361	26	0.322	23	0.445	6	0.334	6	0.619
200	170	1.391	29	1.264	25	1.576	7	1.380	6	2.467
300	174	3.102	33	3.747	29	4.403	7	3.241	6	7.387
400	174	5.433	33	4.997	29	5.870	7	5.760	6	9.850
500	174	6.795	33	6.246	29	7.338	7	9.001	6	12.312
600	177	11.220	36	14.882	31	16.567	7	13.500	6	29.520
700	177	14.230	36	17.363	31	19.328	7	18.376	6	34.440
800	177	18.867	36	19.844	31	22.090	7	24.000	6	39.360
900	177	24.160	36	22.324	31	24.852	7	30.376	6	44.280
1000	177	26.844	36	24.804	31	27.612	7	37.500	6	49.200

Table 23: Runtimes of n parallel demuxes with malicious security in the unrestricted network setting.

n	$\Pi^{\text{Keller}}_{\text{Demux}}$	$\Pi^{\text{Keller-NB}}_{\text{Demux}}$	$\Pi^{\text{Launchbury}}_{\text{Demux}}$	$\Pi^{\text{ParallelComp}}_{\text{Demux}}$	$\Pi^{\text{PreprocessReveal}}_{\text{Demux}}$
10	0.0038	0.0019	0.0018	0.0016	0.0072
20	0.0059	0.0027	0.0028	0.0026	0.0103
30	0.0068	0.0031	0.0032	0.0044	0.0135
40	0.0116	0.0042	0.0045	0.0064	0.0191
50	0.0135	0.0048	0.0052	0.0078	0.0229
60	0.0145	0.0054	0.0063	0.0120	0.0242
70	0.0272	0.0085	0.0091	0.0160	0.0350
80	0.0342	0.0087	0.0098	0.0227	0.0394
90	0.0367	0.0098	0.0110	0.0253	0.0485
100	0.0401	0.0109	0.0120	0.0298	0.0677
200	0.1019	0.0274	0.0282	0.0976	0.1861
300	0.1961	0.0618	0.0690	0.2127	0.4692
400	0.3216	0.0947	0.0864	0.3943	0.6100
500	0.3623	0.1210	0.1197	0.5591	1.0163
600	0.6667	0.2264	0.2098	0.9507	1.9525
700	0.7784	0.2502	0.2498	1.2850	2.0317
800	0.9897	0.3015	0.2839	1.8495	2.8117
900	1.3334	0.3422	0.3227	1.9662	2.9407
1000	1.4891	0.3707	0.3553	2.7042	3.5473

Table 24: Runtimes of n parallel demuxes with malicious security in the LAN setting.

n	$\Pi^{\text{Keller}}_{\text{Demux}}$	$\Pi^{\text{Keller-NB}}_{\text{Demux}}$	$\Pi^{\text{Launchbury}}_{\text{Demux}}$	$\Pi^{\text{ParallelComp}}_{\text{Demux}}$	$\Pi^{\text{PreprocessReveal}}_{\text{Demux}}$
10	0.0233	0.0115	0.0103	0.0086	0.0391
20	0.0314	0.0146	0.0142	0.0100	0.0430
30	0.0323	0.0153	0.0148	0.0122	0.0492
40	0.0431	0.0183	0.0170	0.0151	0.0586
50	0.0455	0.0192	0.0180	0.0189	0.0646
60	0.0469	0.0203	0.0192	0.0238	0.0689
70	0.0668	0.0264	0.0248	0.0293	0.0875
80	0.0749	0.0275	0.0260	0.0378	0.0938
90	0.0770	0.0286	0.0273	0.0435	0.1019
100	0.0831	0.0303	0.0291	0.0506	0.1082
200	0.1709	0.0542	0.0542	0.1539	0.2750
300	0.3156	0.1192	0.1203	0.3225	0.6196
400	0.4848	0.1561	0.1540	0.5675	0.8591
500	0.6209	0.1884	0.1892	0.8492	1.0569
600	0.9595	0.3507	0.3525	1.2867	2.1481
700	1.1627	0.4047	0.3918	1.6384	2.5167
800	1.4726	0.4572	0.4705	2.3504	2.8443
900	1.9268	0.5317	0.5252	2.7774	3.3009
1000	2.1370	0.5755	0.5779	3.4841	3.9626

Table 25: Runtimes of n parallel demuxes with malicious security in the WAN setting.

n	$\Pi^{\text{Keller}}_{\text{Demux}}$	$\Pi^{\text{Keller-NB}}_{\text{Demux}}$	$\Pi^{\text{Launchbury}}_{\text{Demux}}$	$\Pi^{\text{ParallelComp}}_{\text{Demux}}$	$\Pi^{\text{PreprocessReveal}}_{\text{Demux}}$
10	0.1913	0.0962	0.0868	0.0659	0.3141
20	0.2488	0.1200	0.1122	0.0706	0.3235
30	0.2530	0.1223	0.1153	0.0772	0.3305
40	0.3097	0.1396	0.1236	0.0871	0.3520
50	0.3168	0.1422	0.1272	0.0990	0.3627
60	0.3238	0.1454	0.1308	0.1136	0.3739
70	0.3957	0.1828	0.1596	0.1329	0.4276
80	0.4187	0.1864	0.1652	0.1576	0.4437
90	0.4259	0.1910	0.1710	0.1823	0.4583
100	0.4411	0.1964	0.1772	0.2035	0.4752
200	0.8042	0.3162	0.3058	0.6161	0.8741
300	1.4175	0.6086	0.6405	1.3228	1.7860
400	2.2035	0.7563	0.8020	2.2340	2.4103
500	2.7628	0.9019	0.9989	3.5000	3.0308
600	4.2215	1.8248	1.9486	5.2367	6.4601
700	5.4805	2.0866	2.2601	7.0154	7.6976
800	6.7557	2.3531	2.5643	9.0651	8.5152
900	8.6687	2.6272	2.8997	11.7498	10.3587
1000	9.4999	2.9122	3.2187	14.3487	11.6128

Table 26: Rounds and total communication (in megabytes) of n parallel demuxes with malicious security.

n	$\Pi^{\text{Keller}}_{\text{Demux}}$		$\Pi^{\text{Keller-NB}}_{\text{Demux}}$		$\Pi^{\text{Launchbury}}_{\text{Demux}}$		$\Pi^{\text{ParallelComp}}_{\text{Demux}}$		$\Pi^{\text{PreprocessReveal}}_{\text{Demux}}$	
	Rounds	Comm	Rounds	Comm	Rounds	Comm	Rounds	Comm	Rounds	Comm
10	18	0.047	9	0.023	8	0.036	6	0.026	30	0.059
20	23	0.164	11	0.071	10	0.113	6	0.104	30	0.135
30	23	0.242	11	0.104	10	0.167	6	0.234	30	0.186
40	27	0.583	12	0.245	10	0.360	6	0.443	30	0.437
50	27	0.725	12	0.305	10	0.449	6	0.693	30	0.537
60	27	0.867	12	0.366	10	0.539	6	0.998	30	0.638
70	29	1.940	14	0.793	11	1.071	6	1.446	30	1.438
80	29	2.212	14	0.906	11	1.224	6	1.888	30	1.639
90	29	2.483	14	1.019	11	1.377	6	2.390	30	1.839
100	29	2.767	14	1.131	11	1.529	6	2.951	30	2.040
200	32	10.707	15	4.295	11	5.309	9	12.338	30	8.041
300	38	24.291	17	12.491	13	14.620	11	29.381	30	24.025
400	43	42.402	17	16.655	13	19.494	15	52.231	30	32.023
500	46	54.374	17	20.818	13	24.367	20	81.612	30	40.020
600	56	87.016	18	49.059	13	54.534	26	123.999	30	95.941
700	62	111.124	18	57.234	13	63.622	33	168.778	30	111.925
800	72	147.308	18	65.411	13	72.712	44	220.443	30	127.910
900	83	189.427	18	73.588	13	81.801	53	280.049	30	143.894
1000	89	209.664	18	81.763	13	90.889	64	346.241	30	159.879

Table 27: Runtimes of path extraction protocols with semi-honest security in the unrestricted network setting.

n	$\Pi^{\text{SquareAndMult}}_{\text{PathExtract}}$	$\Pi^{\text{Linear}}_{\text{PathExtract}}$	$\Pi^{\text{ShuffleReveal}}_{\text{PathExtract}}$
10	0.0012	0.0015	0.0049
20	0.0019	0.0026	0.0102
30	0.0031	0.0055	0.0132
40	0.0044	0.0078	0.0197
50	0.0062	0.0070	0.0282
60	0.0082	0.0137	0.0335
70	0.0137	0.0112	0.0446
80	0.0159	0.0192	0.0493
90	0.0198	0.0204	0.0604
100	0.0199	0.0237	0.0720
200	0.1497	0.0706	0.2486
300	0.5726	0.2031	0.5218
400	1.4080	0.4588	0.8880
500	2.7023	0.9600	1.1499
600	5.3801	1.4074	1.7332
700	10.1741	2.2515	2.0821
800	17.7856	4.0640	2.8731
900	26.7371	5.8617	3.5448
1000	45.9398	8.0541	3.9509

Table 28: Runtimes of path extraction protocols with semi-honest security in the LAN setting.

n	$\Pi^{\text{SquareAndMult}}_{\text{PathExtract}}$	$\Pi^{\text{Linear}}_{\text{PathExtract}}$	$\Pi^{\text{ShuffleReveal}}_{\text{PathExtract}}$
10	0.0106	0.0128	0.0307
20	0.0133	0.0258	0.0485
30	0.0145	0.0390	0.0666
40	0.0194	0.0535	0.0891
50	0.0232	0.0668	0.1115
60	0.0269	0.0818	0.1357
70	0.0360	0.0961	0.1650
80	0.0429	0.1126	0.1891
90	0.0489	0.1304	0.2133
100	0.0580	0.1451	0.2480
200	0.1937	0.4066	0.5946
300	0.6695	0.8360	1.0212
400	1.5538	1.4360	1.5504
500	3.0073	1.9376	2.1175
600	5.4963	2.5753	2.9683
700	9.0990	3.3489	3.7354
800	18.8130	4.9441	4.9106
900	23.7452	6.2448	5.8240
1000	44.6526	9.5122	7.0739

Table 29: Runtimes of path extraction protocols with semi-honest security in the WAN setting.

n	$\Pi^{\text{SquareAndMult}}_{\text{PathExtract}}$	$\Pi^{\text{Linear}}_{\text{PathExtract}}$	$\Pi^{\text{ShuffleReveal}}_{\text{PathExtract}}$
10	0.0838	0.1034	0.2442
20	0.1065	0.2078	0.3669
30	0.1092	0.3132	0.4941
40	0.1377	0.4174	0.6139
50	0.1465	0.5222	0.7550
60	0.1552	0.6265	0.8935
70	0.1953	0.7396	1.0230
80	0.2139	0.8407	1.1730
90	0.2237	0.9517	1.3310
100	0.2478	1.0589	1.4792
200	0.6865	2.2699	3.3241
300	1.4981	3.7380	5.5541
400	2.8037	5.5557	8.3578
500	4.8590	7.6619	11.9723
600	8.5206	10.5489	16.1881
700	13.6174	13.8019	20.5022
800	19.8454	16.0544	25.8101
900	28.7639	20.3881	31.2433
1000	54.2066	24.4953	38.5208

Table 30: Runtimes of path extraction protocols with malicious security in the unrestricted network setting.

n	$\Pi^{\text{SquareAndMult}}_{\text{PathExtract}}$	$\Pi^{\text{Linear}}_{\text{PathExtract}}$	$\Pi^{\text{ShuffleReveal}}_{\text{PathExtract}}$
10	0.0027	0.0028	0.0145
20	0.0050	0.0056	0.0289
30	0.0074	0.0087	0.0453
40	0.0153	0.0128	0.0637
50	0.0217	0.0175	0.0815
60	0.0269	0.0291	0.1008
70	0.0502	0.0365	0.1428
80	0.0656	0.0543	0.1814
90	0.0901	0.0395	0.2013
100	0.1379	0.0593	0.2716
200	1.2907	0.2653	0.9493
300	5.3074	0.7780	1.8802
400	15.1784	1.9783	3.1638
500	47.6102	5.4697	4.5612
600	90.8231	11.6807	7.0059
700	150.3660	17.4129	8.7878
800	234.9040	25.6054	12.4471
900	369.8428	42.1353	14.8068
1000	445.2730	50.6572	17.6265

Table 31: Runtimes of path extraction protocols with malicious security in the LAN setting.

n	$\Pi^{\text{SquareAndMult}}_{\text{PathExtract}}$	$\Pi^{\text{Linear}}_{\text{PathExtract}}$	$\Pi^{\text{ShuffleReveal}}_{\text{PathExtract}}$
10	0.0192	0.0238	0.0798
20	0.0269	0.0507	0.1229
30	0.0328	0.0762	0.1686
40	0.0522	0.1063	0.2242
50	0.0698	0.1314	0.2888
60	0.0877	0.1668	0.3405
70	0.1315	0.2016	0.4079
80	0.1499	0.2435	0.4740
90	0.1875	0.2833	0.5350
100	0.1995	0.3173	0.6263
200	1.3626	1.0616	1.5916
300	5.3962	2.2297	3.0058
400	15.2777	3.3743	5.0229
500	42.2657	7.7670	6.9971
600	98.4047	12.2983	10.4225
700	151.0490	18.5130	13.1386
800	234.0983	26.9352	18.0736
900	368.6367	41.9274	21.4213
1000	448.5613	51.7315	26.1207

Table 32: Runtimes of path extraction protocols with malicious security in the WAN setting.

n	$\Pi^{\text{SquareAndMult}}_{\text{PathExtract}}$	$\Pi^{\text{Linear}}_{\text{PathExtract}}$	$\Pi^{\text{ShuffleReveal}}_{\text{PathExtract}}$
10	0.1569	0.1965	0.6029
20	0.2063	0.4041	0.8480
30	0.2197	0.6128	1.1152
40	0.2906	0.8238	1.3918
50	0.3231	1.0420	1.7037
60	0.3670	1.2524	1.9966
70	0.4935	1.4774	2.3228
80	0.5553	1.7046	2.6650
90	0.6193	1.9358	3.0181
100	0.6951	2.1626	3.3831
200	2.4283	4.9786	8.0527
300	8.2006	8.6124	14.5242
400	19.8450	13.3358	23.6339
500	44.3491	20.1176	33.7432
600	106.0913	27.2581	46.8397
700	160.9583	35.6690	61.5330
800	241.4790	48.9085	79.4817
900	405.2007	64.4476	99.0512
1000	476.2217	76.8311	119.0983

Table 33: Rounds and total communication (in megabytes) of path-extraction protocols with semi-honest security.

n	$\Pi^{\text{SquareAndMult}}_{\text{PathExtract}}$		$\Pi^{\text{Linear}}_{\text{PathExtract}}$		$\Pi^{\text{ShuffleReveal}}_{\text{PathExtract}}$	
	Rounds	Comm	Rounds	Comm	Rounds	Comm
10	8	0.010	10	0.002	26	0.045
20	10	0.048	20	0.010	36	0.186
30	10	0.109	30	0.022	46	0.423
40	12	0.231	40	0.038	56	0.756
50	12	0.361	50	0.060	66	1.185
60	12	0.520	60	0.086	76	1.710
70	14	0.825	70	0.118	86	2.330
80	14	1.077	80	0.154	96	3.047
90	14	1.363	90	0.194	106	3.860
100	14	1.682	100	0.240	116	4.769
200	16	7.685	200	0.960	216	19.138
300	18	19.447	300	2.160	316	43.107
400	18	34.570	400	3.840	416	76.675
500	18	54.012	500	6.000	516	119.844
600	20	86.414	600	8.640	616	172.613
700	20	117.617	700	11.760	716	234.982
800	20	153.619	800	15.360	816	306.951
900	20	194.422	900	19.440	916	388.518
1000	20	240.024	1 000	24.000	1 016	479.688

Table 34: Rounds and total communication (in megabytes) of path-extraction protocols with malicious security.

n	$\Pi^{\text{SquareAndMult}}_{\text{PathExtract}}$		$\Pi^{\text{Linear}}_{\text{PathExtract}}$		$\Pi^{\text{ShuffleReveal}}_{\text{PathExtract}}$	
	Rounds	Comm	Rounds	Comm	Rounds	Comm
10	15	0.032	19	0.008	59	0.180
20	19	0.158	39	0.031	79	0.640
30	19	0.353	59	0.070	99	1.411
40	23	0.752	79	0.125	119	2.494
50	23	1.174	99	0.195	139	3.889
60	23	1.689	119	0.281	159	5.597
70	27	2.681	139	0.382	179	7.616
80	27	3.501	159	0.499	199	9.947
90	27	4.430	179	0.632	219	12.590
100	27	5.468	199	0.780	239	15.546
200	31	24.976	399	3.120	439	62.258
300	35	63.203	599	7.020	639	140.170
400	58	112.384	799	12.480	862	249.316
500	81	175.605	999	19.500	1 062	389.629
600	108	280.945	1 199	28.080	1 285	561.174
700	131	382.386	1 399	38.220	1 485	763.887
800	131	499.394	1 599	49.920	1 708	997.830
900	154	632.035	1 799	63.180	1 908	1 262.943
1000	246	780.372	1 999	78.000	2 108	1 559.256

Table 35: Runtimes (in seconds) of predecessor-following protocols with various optimizations with semi-honest security in the LAN setting.

n	$\Pi^{\text{Blanton}}_{\text{PredFollow}}$	$\Pi^{\text{Blanton + A.D.}}_{\text{PredFollow}}$	$\Pi^{\text{Split}}_{\text{PredFollow}}$
10	0.10	0.08	0.02
20	0.19	0.14	0.02
30	0.28	0.20	0.02
40	0.37	0.27	0.03
50	0.45	0.33	0.03
60	0.54	0.40	0.04
70	0.65	0.47	0.05
80	0.76	0.56	0.06
90	0.84	0.63	0.07
100	0.95	0.70	0.08
200	1.90	1.65	0.28
300	2.88	2.88	0.85
400	3.88	3.82	1.68
500	4.86	4.89	2.78
600	5.92	5.92	3.33
700	7.07	6.97	4.81
800	8.11	8.09	5.32
900	9.39	9.20	6.67
1000	10.57	9.76	7.50

Table 36: Runtimes (in seconds) of predecessor-following protocols with various optimizations with semi-honest security in the unrestricted network setting.

n	$\Pi^{\text{Blanton}}_{\text{PredFollow}}$	$\Pi^{\text{Blanton + A.D.}}_{\text{PredFollow}}$	$\Pi^{\text{Split}}_{\text{PredFollow}}$
10	0.01	0.01	0.00
20	0.02	0.02	0.00
30	0.03	0.04	0.01
40	0.05	0.05	0.01
50	0.06	0.06	0.01
60	0.07	0.07	0.01
70	0.09	0.09	0.02
80	0.10	0.11	0.02
90	0.12	0.12	0.03
100	0.14	0.13	0.04
200	0.28	0.27	0.15
300	0.44	0.43	0.69
400	0.63	0.64	1.17
500	0.88	0.85	1.68
600	1.02	1.10	2.32
700	1.31	1.34	2.64
800	1.68	1.68	3.60
900	1.90	1.90	4.12
1000	2.23	2.23	4.70

Table 37: Runtimes (in seconds) of predecessor-following protocols with various optimizations with semi-honest security in the WAN setting.

n	$\Pi^{\text{Blanton}}_{\text{PredFollow}}$	$\Pi^{\text{Blanton + A.D.}}_{\text{PredFollow}}$	$\Pi^{\text{Split}}_{\text{PredFollow}}$
10	0.86	0.64	0.14
20	1.59	1.11	0.16
30	2.31	1.58	0.16
40	3.04	2.06	0.20
50	3.78	2.54	0.21
60	4.51	3.02	0.22
70	5.57	3.54	0.28
80	6.35	4.03	0.30
90	7.15	4.51	0.32
100	7.93	4.99	0.34
200	15.82	10.14	0.88
300	23.80	15.87	1.92
400	32.08	21.29	3.44
500	40.19	26.92	6.99
600	48.74	35.29	13.87
700	57.11	41.62	20.92
800	65.92	47.57	20.40
900	74.49	54.55	24.96
1000	83.45	60.88	30.89

Table 38: Runtimes (in seconds) of predecessor-following protocols with various optimizations with malicious security in the unrestricted network setting.

n	$\Pi^{\text{Blanton}}_{\text{PredFollow}}$	$\Pi^{\text{Blanton + A.D.}}_{\text{PredFollow}}$	$\Pi^{\text{Split}}_{\text{PredFollow}}$
10	0.02	0.02	0.01
20	0.03	0.04	0.01
30	0.05	0.06	0.01
40	0.08	0.08	0.02
50	0.10	0.10	0.03
60	0.12	0.12	0.05
70	0.15	0.16	0.05
80	0.18	0.19	0.06
90	0.21	0.21	0.08
100	0.23	0.23	0.10
200	0.62	0.54	0.45
300	1.07	1.05	2.35
400	1.79	1.21	3.42
500	2.70	1.13	5.03
600	3.44	2.31	7.85
700	4.34	2.76	10.10
800	5.31	3.39	13.66
900	7.43	3.76	16.86
1000	8.03	4.17	19.92

Table 39: Runtimes (in seconds) of predecessor-following protocols with various optimizations with malicious security in the LAN setting.

n	$\Pi^{\text{Blanton}}_{\text{PredFollow}}$	$\Pi^{\text{Blanton + A.D.}}_{\text{PredFollow}}$	$\Pi^{\text{Split}}_{\text{PredFollow}}$
10	0.12	0.12	0.03
20	0.22	0.22	0.04
30	0.31	0.34	0.05
40	0.43	0.43	0.07
50	0.54	0.53	0.09
60	0.65	0.66	0.11
70	0.82	0.86	0.16
80	0.98	0.92	0.19
90	1.08	1.09	0.22
100	1.20	1.22	0.28
200	2.67	2.69	1.25
300	4.30	4.33	2.99
400	6.09	6.13	4.04
500	8.02	8.01	7.65
600	10.37	10.41	11.35
700	12.61	13.48	14.66
800	15.33	15.61	19.09
900	18.29	17.66	23.13
1000	21.07	19.63	27.86

Table 40: Runtimes (in seconds) of predecessor-following protocols with various optimizations with malicious security in the WAN setting.

n	$\Pi^{\text{Blanton}}_{\text{PredFollow}}$	$\Pi^{\text{Blanton + A.D.}}_{\text{PredFollow}}$	$\Pi^{\text{Split}}_{\text{PredFollow}}$
10	1.00	1.00	0.24
20	1.84	1.84	0.30
30	2.69	2.69	0.32
40	3.54	3.53	0.41
50	4.39	4.39	0.46
60	5.25	5.26	0.51
70	6.82	6.84	0.66
80	7.82	7.82	0.76
90	8.81	8.78	0.81
100	9.77	9.75	0.91
200	19.97	20.01	2.95
300	31.08	31.09	13.26
400	42.59	42.50	14.73
500	54.54	54.77	22.84
600	67.74	67.68	32.38
700	80.84	81.15	43.15
800	95.87	95.80	56.32
900	110.85	110.97	76.13
1000	125.38	125.74	98.62

Table 41: Runtimes of padding-handling protocols with semi-honest security in the unrestricted network setting.

n	$\Pi_{\text{PaddingReorder}}$	$\Pi_{\text{GraphTransform}}$
10	0.0029	0.0000
20	0.0037	0.0001
30	0.0045	0.0001
40	0.0053	0.0002
50	0.0070	0.0002
60	0.0077	0.0002
70	0.0094	0.0003
80	0.0108	0.0003
90	0.0119	0.0003
100	0.0121	0.0004
200	0.0322	0.0011
300	0.0658	0.0020
400	0.1085	0.0033
500	0.1509	0.0051
600	0.2261	0.0074
700	0.3002	0.0111
800	0.3887	0.0131
900	0.5055	0.0159
1000	0.5741	0.0200

Table 42: Runtimes of padding-handling protocols with semi-honest security in the LAN setting.

n	$\Pi_{\text{PaddingReorder}}$	$\Pi_{\text{GraphTransform}}$
10	0.0178	0.0000
20	0.0184	0.0001
30	0.0198	0.0001
40	0.0208	0.0002
50	0.0233	0.0002
60	0.0243	0.0003
70	0.0272	0.0003
80	0.0292	0.0004
90	0.0318	0.0004
100	0.0338	0.0004
200	0.0650	0.0010
300	0.1087	0.0021
400	0.1674	0.0033
500	0.2286	0.0051
600	0.3069	0.0073
700	0.4077	0.0109
800	0.5646	0.0125
900	0.6255	0.0168
1000	0.7808	0.0206

Table 43: Runtimes of padding-handling protocols with semi-honest security in the WAN setting.

n	$\Pi_{\text{PaddingReorder}}$	$\Pi_{\text{GraphTransform}}$
10	0.1451	0.0000
20	0.1470	0.0001
30	0.1505	0.0002
40	0.1552	0.0002
50	0.1582	0.0003
60	0.1627	0.0003
70	0.1696	0.0004
80	0.1758	0.0005
90	0.1791	0.0006
100	0.1860	0.0006
200	0.2706	0.0011
300	0.4249	0.0020
400	0.5970	0.0032
500	0.8376	0.0053
600	1.1096	0.0077
700	1.4468	0.0107
800	1.9664	0.0123
900	2.5709	0.0161
1000	3.1528	0.0213

Table 44: Runtimes of padding-handling protocols with malicious security in the unrestricted network setting.

n	$\Pi_{\text{PaddingReorder}}$	$\Pi_{\text{GraphTransform}}$
10	0.0128	0.0000
20	0.0183	0.0001
30	0.0198	0.0002
40	0.0251	0.0002
50	0.0300	0.0002
60	0.0289	0.0003
70	0.0380	0.0003
80	0.0377	0.0004
90	0.0407	0.0004
100	0.0484	0.0005
200	0.1204	0.0016
300	0.2123	0.0036
400	0.4031	0.0061
500	0.5301	0.0097
600	0.8062	0.0132
700	1.1222	0.0173
800	1.4381	0.0223
900	1.8237	0.0318
1000	2.1326	0.0475

Table 45: Runtimes of padding-handling protocols with malicious security in the LAN setting.

n	$\Pi_{\text{PaddingReorder}}$	$\Pi_{\text{GraphTransform}}$
10	0.0828	0.0000
20	0.0844	0.0001
30	0.0886	0.0002
40	0.0944	0.0002
50	0.0963	0.0003
60	0.1067	0.0004
70	0.1152	0.0004
80	0.1190	0.0005
90	0.1201	0.0006
100	0.1326	0.0006
200	0.2180	0.0017
300	0.3438	0.0036
400	0.5278	0.0060
500	0.7867	0.0094
600	1.0931	0.0123
700	1.3137	0.0150
800	1.9081	0.0273
900	2.4231	0.0349
1000	2.9203	0.0389

Table 46: Runtimes of padding-handling protocols with malicious security in the WAN setting.

n	$\Pi_{\text{PaddingReorder}}$	$\Pi_{\text{GraphTransform}}$
10	0.6722	0.0001
20	0.6802	0.0001
30	0.6859	0.0002
40	0.6986	0.0002
50	0.7097	0.0003
60	0.7201	0.0004
70	0.7303	0.0005
80	0.7430	0.0006
90	0.7638	0.0007
100	0.7840	0.0008
200	1.0523	0.0017
300	1.4726	0.0036
400	2.1490	0.0054
500	3.0246	0.0088
600	4.2919	0.0123
700	5.6004	0.0179
800	7.2261	0.0245
900	8.9857	0.0316
1000	11.2390	0.0444

Table 47: Rounds and total communication (in megabytes) of padding-handling protocols with semi-honest security.

n	$\Pi_{\text{PaddingReorder}}$		$\Pi_{\text{GraphTransform}}$	
	Rounds	Comm	Rounds	Comm
10	17	0.008	0	0.000
20	17	0.026	0	0.000
30	17	0.053	0	0.000
40	17	0.089	0	0.000
50	17	0.136	0	0.000
60	17	0.192	0	0.000
70	17	0.257	0	0.000
80	17	0.332	0	0.000
90	17	0.417	0	0.000
100	17	0.511	0	0.000
200	17	1.983	0	0.000
300	17	4.414	0	0.000
400	17	7.805	0	0.000
500	17	12.156	0	0.000
600	17	17.467	0	0.000
700	17	23.739	0	0.000
800	17	30.970	0	0.000
900	17	39.161	0	0.000
1000	17	48.312	0	0.000

Table 48: Rounds and total communication (in megabytes) of padding-handling protocols with malicious security.

n	$\Pi_{\text{PaddingReorder}}$		$\Pi_{\text{GraphTransform}}$	
	Rounds	Comm	Rounds	Comm
10	65	0.092	0	0.000
20	65	0.149	0	0.000
30	65	0.237	0	0.000
40	65	0.356	0	0.000
50	65	0.507	0	0.000
60	65	0.689	0	0.000
70	65	0.902	0	0.000
80	65	1.146	0	0.000
90	65	1.421	0	0.000
100	65	1.727	0	0.000
200	65	6.509	0	0.000
300	65	14.410	0	0.000
400	65	25.431	0	0.000
500	65	39.572	0	0.000
600	65	56.833	0	0.000
700	65	77.214	0	0.000
800	65	100.715	0	0.000
900	65	127.336	0	0.000
1000	88	157.110	0	0.000

Table 49: Comparison of the start-to-end runtimes with semi-honest security in the unrestricted network setting.

n	$\Pi_{\text{SPSP}}^{\text{Baseline}}$ [s]	$\Pi_{\text{SPSP}}^{\text{Optimized}}$ [s]	Speedup	
			Absolute [s]	Relative
10	0.05	0.04	0.01	1.23
20	0.15	0.13	0.02	1.15
30	0.26	0.22	0.03	1.15
40	0.37	0.32	0.05	1.16
50	0.48	0.45	0.03	1.06
60	0.59	0.57	0.01	1.02
70	0.74	0.69	0.05	1.07
80	0.87	0.82	0.05	1.05
90	1.06	0.94	0.13	1.14
100	1.18	1.10	0.08	1.08
200	2.78	2.87	-0.09	0.97
300	4.83	5.28	-0.45	0.91
400	7.19	7.13	0.05	1.01
500	9.55	9.99	-0.44	0.96
600	14.27	13.06	1.21	1.09
700	16.75	15.84	0.91	1.06
800	24.81	18.88	5.94	1.31
900	26.61	26.08	0.53	1.02
1000	28.39	27.69	0.70	1.03

Table 50: Comparison of the start-to-end runtimes with semi-honest security in the LAN setting.

n	$\Pi_{\text{SPSP}}^{\text{Baseline}}$ [s]	$\Pi_{\text{SPSP}}^{\text{Optimized}}$ [s]	Speedup	
			Absolute [s]	Relative
10	0.28	0.17	0.11	1.66
20	0.58	0.39	0.19	1.50
30	1.04	0.75	0.29	1.39
40	1.62	1.25	0.37	1.29
50	2.29	1.85	0.44	1.24
60	2.99	2.54	0.46	1.18
70	3.88	3.48	0.40	1.11
80	5.20	4.18	1.01	1.24
90	6.31	6.15	0.16	1.03
100	8.96	7.24	1.72	1.24
200	20.72	19.96	0.76	1.04
300	34.57	34.18	0.39	1.01
400	47.79	46.90	0.89	1.02
500	61.90	61.76	0.14	1.00
600	78.65	77.32	1.33	1.02
700	92.63	92.88	-0.25	1.00
800	109.73	109.77	-0.04	1.00
900	127.52	125.35	2.16	1.02
1000	148.36	141.96	6.40	1.05

Table 51: Comparison of the start-to-end runtimes with semi-honest security in the WAN setting.

n	$\Pi_{\text{SPSP}}^{\text{Baseline}}$ [s]	$\Pi_{\text{SPSP}}^{\text{Optimized}}$ [s]	Speedup	
			Absolute [s]	Relative
10	2.15	1.29	0.86	1.66
20	4.05	2.48	1.57	1.63
30	6.18	3.91	2.27	1.58
40	8.59	5.56	3.02	1.54
50	11.35	7.60	3.75	1.49
60	14.35	9.99	4.36	1.44
70	18.33	12.61	5.72	1.45
80	22.09	15.64	6.45	1.41
90	26.35	19.32	7.04	1.36
100	31.46	23.62	7.84	1.33
200	132.31	115.15	17.15	1.15
300	282.98	260.25	22.73	1.09
400	396.42	368.08	28.34	1.08
500	510.55	475.30	35.24	1.07
600	633.56	594.65	38.92	1.07
700	759.62	715.44	44.18	1.06
800	887.98	836.91	51.07	1.06
900	1 014.99	960.05	54.95	1.06
1000	1 139.65	1 085.60	54.05	1.05

Table 52: Comparison of the start-to-end runtimes with malicious security in the unrestricted network setting.

n	$\Pi_{\text{SPSP}}^{\text{Baseline}}$ [s]	$\Pi_{\text{SPSP}}^{\text{Optimized}}$ [s]	Speedup	
			Absolute [s]	Relative
10	0.09	0.06	0.03	1.44
20	0.26	0.20	0.06	1.30
30	0.65	0.53	0.12	1.22
40	1.14	1.09	0.05	1.05
50	0.94	0.82	0.12	1.14
60	1.08	1.01	0.08	1.08
70	1.37	1.21	0.16	1.14
80	1.66	1.53	0.13	1.09
90	1.96	1.66	0.30	1.18
100	2.15	1.93	0.22	1.11
200	5.47	5.68	-0.21	0.96
300	10.26	8.93	1.33	1.15
400	14.76	13.41	1.35	1.10
500	18.74	18.57	0.17	1.01
600	25.00	23.49	1.50	1.06
700	32.29	29.59	2.70	1.09
800	39.42	38.14	1.28	1.03
900	46.47	44.87	1.60	1.04
1000	56.03	50.27	5.76	1.11

Table 53: Comparison of the start-to-end runtimes with malicious security in the LAN setting.

n	$\Pi_{\text{SPSP}}^{\text{Baseline}}$ [s]	$\Pi_{\text{SPSP}}^{\text{Optimized}}$ [s]	Speedup	
			Absolute [s]	Relative
10	0.42	0.26	0.17	1.66
20	0.90	0.64	0.26	1.42
30	1.66	1.32	0.34	1.26
40	2.78	2.35	0.44	1.19
50	4.18	3.74	0.43	1.12
60	6.20	5.84	0.36	1.06
70	8.82	7.92	0.90	1.11
80	12.77	12.00	0.78	1.06
90	17.33	16.03	1.29	1.08
100	22.47	21.64	0.83	1.04
200	30.19	28.16	2.02	1.07
300	48.53	47.00	1.53	1.03
400	69.03	67.41	1.62	1.02
500	91.64	90.86	0.78	1.01
600	114.86	114.59	0.27	1.00
700	140.47	138.58	1.89	1.01
800	167.02	163.39	3.63	1.02
900	193.35	190.89	2.46	1.01
1000	220.26	217.09	3.17	1.01

Table 54: Comparison of the start-to-end runtimes with malicious security in the WAN setting.

n	$\Pi_{\text{SPSP}}^{\text{Baseline}}$ [s]	$\Pi_{\text{SPSP}}^{\text{Optimized}}$ [s]	Speedup	
			Absolute [s]	Relative
10	3.17	1.76	1.42	1.81
20	5.72	3.55	2.17	1.61
30	9.09	6.09	3.00	1.49
40	13.72	10.06	3.66	1.36
50	20.18	15.07	5.11	1.34
60	27.28	21.86	5.42	1.25
70	37.82	30.69	7.12	1.23
80	50.72	43.72	7.00	1.16
90	64.84	57.08	7.76	1.14
100	83.30	74.30	9.00	1.12
200	238.77	220.27	18.50	1.08
300	378.84	357.60	21.24	1.06
400	529.27	499.99	29.27	1.06
500	681.38	646.34	35.04	1.05
600	848.15	806.60	41.55	1.05
700	1 017.97	972.92	45.05	1.05
800	1 191.86	1 143.13	48.74	1.04
900	1 368.76	1 322.45	46.30	1.04
1000	1 546.71	1 495.01	51.69	1.03