Securely Solving Simple Combinatorial Graph Problems

Abdelrahaman Aly, Edouard Cuvelier, Sophie Mawet,
Olivier Pereira, and Mathieu Van Vyve

Université catholique de Louvain
ICTEAM and IMMAQ institutes
1348 Louvain-la-Neuve – Belgium

Abstract. We investigate the problem of solving traditional combinatorial graph problems using secure multi-party computation techniques, focusing on the shortest path and the maximum flow problems. To the best of our knowledge, this is the first time these problems have been addressed in a general multi-party computation setting. Our study highlights several complexity gaps and suggests the exploration of various trade-offs, while also offering protocols that are efficient enough to solve real-world problems.

1 Introduction

Secure multi-party computation – the problem of jointly evaluating a function on a set of secret inputs without leaking anything but the output of the function – has been at the center of cryptography research for almost 30 years. A first series of foundational works [1,2,3,4] demonstrated the possibility to evaluate any function in various models, the function being described as a circuit. The attention then largely focused on building solutions for the evaluation of functions of specific interest, leading to secure and efficient protocols for auctions [5], voting [6], benchmarking [7], face recognition [8] or AES evaluation [9] to only mention a few.

One common point of all these applications is that the function evaluation process is naturally oblivious of the inputs on which the function has to be evaluated. Computing the highest of n bids or summing n votes is carried out by performing n comparisons or sums independently of the values that are considered.

There are large classes of problems however for which the natural evaluation process depends on the input data. In that case, even if all the manipulated data are appropriately shared or encrypted, the execution flow might just be sufficient to leak undesirable information.

This is typically the case in combinatorial problems, of which graph problems are one of the most common examples. Consider, for instance, a consortium of delivery companies covering different territories through regular distribution circuits. These companies might be interested in computing the fastest way to bring

a package from one place to another, but be reluctant to share with each other the precise connections they use and the performance of their trucks. Their problem could be solved by securely evaluating traditional shortest path algorithms such as those of Bellman-Ford or Dijkstra. The immediate way of securely computing the shortest path would be to blind (encrypt or share) the weight of all the edges of the corresponding graph. However this approach could completely miss its purpose depending on the graph encoding and shortest path algorithm that are used: if the algorithm conditionally visits the graph by branching as a function of the secret weights, then the branching patterns could leak a substantial amount of secret information. In a similar way, the resolution of combinatorial problems, even on obfuscated inputs, can leak substantial information through the structure of the combinatorial object that is manipulated, as well as through its running time. We stress that this is not just a theoretical concern: numerous techniques have been developed, notably in the line of work on side-channel attacks [10], that can successfully exploit branching patterns and running times in order to recover the secrets on which computation is performed.

1.1 Our Contributions

This paper investigates the problem of securely solving combinatorial problems in a multi-party setting through a series of examples taken from graph theory. To the best of our knowledge, this is the first time that these most classical algorithmic problems have been addressed in a general secure multi-party setting. Our solutions have applications in the numerous contexts where a graph is shared between competing entities. Natural examples include: privacy-preserving GPS guidance in which one party knows the map while the other knows his origin and destination, privacy-preserving determination of topological features in social network (the number of different ways to connect two people can be seen as a special case of the maximum flow problem, for instance, in which case each party would know his own friends but no more), or privacy-preserving determination of the performance of the cooperation between competing network operators (gas, electricity, logistics, ...), in which each party would know the capacity of his own infrastructure but no more. Furthermore, our study raises several intriguing complexity gaps and suggests the exploration of various trade-offs.

Algorithm Design. We focus our research on computing the shortest path and the maximum flow based on the secure arithmetic black-box functionality of Damgård and Nielsen [11] augmented with comparison [12]. That is, our protocols assume access to a functionality that offers secure addition, multiplication and comparison. This allows us to abstract from the specific security model in which we want our protocol to be secure: depending on the implementation of the secure arithmetic black-box that is used, our protocols will be secure only in the presence of an honest majority or with up to all but one corrupted player, in the information theoretic or computational model, in front of passive or active adversaries, ... Various such implementations, in various models, are available in tools designed for multi-party computation such as FairplayMP [13], Sepia [14], Sharemind [15] or VIFF [16].

We focus on two of the most standard graph problems, chosen for their wide diversity of applications: computing shortest paths and maximum flows. For each of these problems, we discuss secure evaluation techniques inspired from classical algorithms of various complexities: Bellman-Ford and Dijkstra for the shortest path, and Edmonds-Karp and Push-Relabel for the maximum flow.

Our resulting algorithms offer quite different overheads, depending on the algorithm and the graph structure, as illustrated in Table 1. For those algorithms, the table shows first the traditional (non secure) complexity, then the complexity of our secure versions expressed in number of calls to the arithmetic functionality. There, we consider the case of a graph with public structure and then with private structure, meaning that not only the weight of each edge is kept secret, but that the adjacency relation between vertices is kept private as well.

Several observations can already be made.

- The best implementations, using advanced data structures as dynamic trees [17] or Fibonacci heaps [18], are definitely non-trivial to replicate in the secure setting (see also discussion in Section 1.2 below). Their relevance is also unclear for the relatively small size of the problems that we are addressing, as they usually come with large constants.
- The overheads resulting from moving from the original algorithms to their secure versions largely differ between algorithms: in the case of a public structure for instance, we see either no difference, or an $|E|$ factor or a $|V|$ factor depending on the algorithm.
- The overhead resulting from hiding the graph structure largely differs depending on the algorithm and type of graph. For Bellman-Ford and Push-Relabel, the difference essentially corresponds to always handling a complete graph when the structure needs to be hidden. For Dijkstra however, the secrecy of the graph structure has no impact.
- While Bellman-Ford is traditionally less efficient than Dijkstra, this is not true anymore (asymptotically at least) for our secure variants: Bellman-Ford becomes substantially more efficient for sparse graphs (e.g., if $|E| = \mathcal{O}(|V|)$) and the asymptotic complexities are similar for dense graphs.

The overheads in terms of number of protocol participants, round complexity, ... largely depend on the implementation of the secure arithmetic functionality, and are in line with traditional works.

	Optimized	Original	Public Structure	Secret Structure
Bellman-Ford	$\|V\|\|E\|$	$\|V\|\|E\|$	$\|V\|\|E\|$	$\|V\|^3$
Dijkstra	$\|E\| + \|V\| \log \|V\|$	$\|V\|^2$	$\|V\|^3$	$\|V\|^3$
Edmonds-Karp	$\|V\|^2\|E\|$	$\|V\|\|E\|^2$	$\|V\|\|E\|^2$	$\|V\|^5$
Push-Relabel	$\|V\|\|E\| \log(\frac{\|V\|^2}{\|E\|})$	$\|V\|^3$	$\|V\|^2\|E\|$	$\|V\|^4$

Table 1. Asymptotic complexities: original algorithms and secure versions with public and private graph adjacency matrix.

Complexity: The Constants Matter. In order to challenge our algorithms in practice, we implemented them all using the Virtual Ideal Functionality Framework (VIFF) of Geisler et al. [16], in the honest-but-curious model.

This allowed us to further investigate the constants hidden by the asymptotic notations discussed above. This made particularly visible the difference of cost between the different black-box primitives that we used: addition based on linear secret sharing [19] comes for free (no communication involved), multiplication is noticeable (it involves one secret sharing), and comparison (based on Toft's protocol [20]) is ≈ 165 times more expensive than a multiplication, something that strongly contrasts with the execution time of traditional algorithms.

These differences have strong practical impact and motivated some trade-offs as well.

- Our version of Dijkstra's algorithm involves only $|V|^2$ comparisons compared to $|V|^3$ (or $|V||E|$) in Bellman-Ford. As a result of this, for dense graphs or when the graph structure is secret, Dijkstra's algorithm remains considerably more efficient than Bellman-Ford's, even when the structure of the graph is public, provided that the graphs have a reasonably small size (a hundred vertices).
- For sparse public graphs that contain a small number of paths from the source to the sink, a variant of the Edmonds-Karp's algorithm that relies on an exhaustive public enumeration of the source to sink paths can be considerably simpler and more efficient than a secure version of the breadth-first search for augmenting paths that is performed in the original algorithm: this allows trading expensive book-keeping and addressing operations for more but much simpler rounds.

So, besides the fact that our work offers the first solutions for the secure evaluation of various graph properties, we think that it raises several intriguing complexity issues. Notably, we wonder whether the complexity gaps that we have are inherent to the added security or if they can be improved.

1.2 Related Works

As mentioned above, the large majority of works on secure multi-party computation focused on functions whose evaluation execution flow is independent of the secret inputs. There are important exceptions to this, however.

Branching Programs. Branching programs are decision procedures that, based on some inputs and decision parameters such as thresholds, perform a specific classification of the input. Secure versions of these programs, where a user does not learn the branching program of the server while the server does not learn the user's inputs, have been considered in various works [21], [22], [23], [7], [24]. While these works share our goals of hiding the data path through which the program is going, they do not aim at hiding the length of that path which, in our case at least, could leak a substantial amount of information.

Shortest Path In The Two-Party Setting. Brickell and Shmatikov [25] addressed the problem of solving some graph problems securely and their work is, as such, the closest to ours. Substantial differences appear, though.

First, their security model is quite different from ours. Their protocols, which are based on a privacy-preserving set union protocol, proceed by making their outputs known to the participants progressively as part of the execution (e.g.,

edge by edge as the protocol runs). Even though this is not revealing more than the eventual outcome, this makes their protocols unusable as sub-components of other higher-level protocols that would rely on using these outputs as part of their secret state. Revealing outputs part-by-part as the protocol runs might also be problematic in applications in which some participants could abort the protocol in the middle of its execution, based on what they have already learned. Our protocols, on the other hand, can be freely used as subroutines, and one of our secure max-flow algorithms will make use of a secure shortest-path algorithm.

Second, the graph problems they consider are different from ours as well. They do not consider the maximum flow problem at all: their work focuses on computing shortest distances, from a known source to all the vertices or for all the vertex pairs, in a setting where all the participants assign a weight to all edges. We further investigate the problem of computing the shortest path from a single source to a single destination, which cannot be done using their set union technique as it would reveal much more information than the specific distance we are interested in.

Eventually, their protocols are not based on generic building-blocks, like the arithmetic black-box functionality on which we rely. Specifically, their protocols are designed for the two-party computation setting in the honest-but-curious model. While these specifics allow them to develop techniques that are quite efficient in this two-party setting, it is unclear how efficient a transposition of their approach to the multi-party setting would be.

Efficient Secure Datastructures. The problem of computing securely on datastructures has recently been investigated by Toft [12], in the case of a secure priority queue, which he implements using a variation of bucket heap. The problem studied there shares similar flavors with those we address here: to compute securely on structured data by keeping the actions independent of the inputs. The computational overhead compared to the efficiency of the original bucket heap is logarithmic, making it occupy an interestingly different spot in the list of overhead examples discussed above.

Similar effects could also be achieved through the use of oblivious memories [26], [27].

Overview. Section 2 describes the building blocks we will use and our main implementation choices. Section 3 describes our approach to the classical single source and single-pair shortest path problems, and Section 4 describes our approaches of the maximum flow problem.

2 Preliminaries

2.1 Black-Box Operations

Arithmetic Black-Box. We build our protocols on top of an ideal functionality : the arithmetic black-box functionality $\mathcal{F}_{ABB}$ of Damgård and Nielsen [11], whose definition captures the properties we need.

This functionality allows n parties to securely store elements of a ring $\mathbb{Z}_m$, to repeatedly perform the ring operations of addition and multiplication on these

elements, and to open the result of the computation when needed. Following Toft [12], we consider a slightly extended and abstracted version of this functionality that offers the possibility to perform secure comparison and consider any possible ring. So, storing, opening, addition, multiplication and comparison will be the only secure operations on which our protocols will rely. Following the tradition, we will write $[x]$ to address the version of x securely stored by $\mathcal{F}_{ABB}$, and denote the secure arithmetic operations on secret values in the natural way, e.g., $[z] \leftarrow [x] + [y]$ for the addition of two secrets. The actual protocol implementing these operations depends on the details of the realization of this functionality. Numerous MPC schemes can be used for that purpose [4,3,11] or, for more recent approaches [28,29,30], depending on the security model that is appropriate.

Graph Representation. Depending on the algorithm we are trying to compute and on the part of the graph description that is part of the secret input, different graph representation approaches will show to be useful.

In all cases, we will assume that the number of vertices in the graph is public (or at least an upper bound on it). Depending on the setting, the adjacency relationship between the vertices might be public or not. For instance, it is natural to have it public if the graph represents the connections between places on a map, but it might be desirable to keep it secret if the presence of edges reveals the existence of transactions between competing companies.

A traditional structure for storing a graph consists in storing, with every vertex, a list of its neighbors (and the weight of the corresponding edges). This structure is quite efficient in terms of memory. However, it might be quite problematic from a security point of view, as it discloses the degree of each vertex. A solution would be to tolerate the leakage of an upper bound on these degrees, but that upper bound would be close to imply the storage of a complete graph as soon as one single vertex is of high degree. Furthermore, even if the leakage of the degree of the vertices is tolerated, algorithms that perform breadth-first search on vertices and branch depending on the weight of edges could reveal a lot of information. As a result, this graph representation can show to be very effective in some cases, but completely inappropriate in others, even when the graph structure is public.

A second traditional way of representing graphs is to store their adjacency matrix, the elements of the matrix representing the weight of the edges between vertices. This approach has the benefit of offering a storage that is independent of the graph structure. While running our algorithms, we will often need to perform some operations on a specific vertex designated by a secret index. This will typically be performed by running that operation on all vertices, including a canceling factor everywhere but on the vertex that needs to be treated. An obvious way of testing whether we are working on the right vertex would be to perform a test at each step. We actually use a more effective approach by representing the index of the vertex i by a vector $[\mathbf{i}] \in \{[0], [1]\}^{1 \times n}$ where each entry is $[0]$ except for the i'th which is $[1]$. We can then access the weight of the edge from vertex i to vertex j by computing the matrix product $[\mathbf{i}].[\mathbf{M}].[\mathbf{j}]^t$.

Protocol 1 provides a way to update an element in a shared list and it can be easily extended to update an element in a shared matrix. This protocol also exemplifies a common way of emulating a branching depending on a secret value: the arithmetic operation in the loop is actually equivalent to computing **if** $[\mathbf{i}]_j = [1]$ **then** $[\mathbf{l}]_j = [x]$.

Protocol 1: Update an element in a shared list and at a shared position

Input: A list $[\mathbf{l}]$ of length n, a shared index $[\mathbf{i}]$, a shared value $[x]$.
Output: The list $[\mathbf{l}]$ with the update $[\mathbf{l}]_i = [x]$.

```
for j ← 1 to n do
    [l]_j ← [l]_j + [i]_j · ([x] − [l]_j);
end
```

For a graph with n vertices, this protocol allows retrieving a secret position in the adjacency matrix in $\mathcal{O}(n^2)$ multiplications instead of $\mathcal{O}(n^2)$ comparisons, which is considerably more efficient, even if it implies a considerable overhead in storage (moving from 1 secret index to n secret bits). We note that, in all cases, this approach implies treating the graph as if it were complete, which can be a considerable waste of resources if the graph is actually sparse.

3 Privacy-Preserving Shortest Path Problem

The single-source shortest path problem is a major problem in graph theory. It has several immediate applications. The typical one is finding the shortest way to connect two cities on a road map where each city is represented by a vertex and each road between two cities by an edge. The edge weights are the road distances between cities. In this context, a user may then want to obtain driving directions without revealing neither his starting point nor his destination. Another application is the one of two entities owning each a secret location in a shared network and willing to compute the distance between them without disclosing their location. We note that such a problem is worth solving even for relatively small graphs. Consider for instance a routing network with a dozen hubs in different European countries and three competing logistic companies having each their own transportation costs for a defined set of roads. As costs typically represent sensitive information that should not be disclosed to competitors, being able to solve the shortest path problem securely for 3 parties and a graph with a dozen of vertices is quite helpful. Similar problems happen for network traffic on routers where a small number of big hubs is involved. Competing companies have to solve the shortest path to define routing scheme without revealing sensitive information about internal network configuration.

Shortest path algorithms are also used as subalgorithms for more advanced problems like the Chinese postman problem or the max-flow problem that we address below: this highlights the importance of keeping our protocols composable.

We investigate two standard algorithms for finding the single-source shortest path in a graph with weighted edges: Dijkstra's algorithm and Bellman-Ford's

algorithm. The first one requires all edge weights to be positive, while the second one only assumes there is no negative-weight cycle in the input graph. As the non-secure version of all the algorithms that we treat is widely available [31], we will only briefly outline them.

All our protocols assume that inputs are already stored in the $\mathcal{F}_{ABB}$ functionality and give access to the stored outputs (that can be opened through opening requests to $\mathcal{F}_{ABB}$). This feature guarantees the composability of the protocols. The way inputs and outputs are shared depends on the application: they might come from a specific problem, or from the needs of a higher-level protocol using this protocol as a sub-routine, for instance.

3.1 Bellman-Ford's Algorithm

The algorithm of Bellman-Ford is particularly simple, making it a natural target for building a secure version. This algorithm proceeds by repeatedly scanning all edges, in search of adding edges that decrease the ongoing distance from the source to the various vertices. If a pass over the edges did not improve the current solution, or if the edges were scanned $|V|$ times, the algorithm halts. An interesting feature of this algorithm is that its flow of operations only depends on the structure of the graph but not on the weight of the edges. Its drawback is its time-complexity: its classical implementation runs in $\mathcal{O}(|V||E|)$ time.

Protocol 2 (the SSP1 protocol) presents our secure shortest path protocol based on Bellman-Ford. Note that $h(e)$ and $t(e)$ represent the head and tail vertex of an edge e respectively. Also, note that $\top$ is a number agreed in advance by the players as a higher bound for some calculations of the protocol. We refer to Appendix A for discussion of the values of $\top$ and m in all our protocols. Finally, note that `updatevector` refers to Protocol 1. The SSP1 protocol differs from the original algorithm only in a limited number of aspects: *a*) the branching corresponding to the discovery of a shorter path is handled on Lines 8–10 through arithmetic operations as in Protocol 1, *b*) the early termination condition of the Bellman-Ford algorithm, which is triggered if the inner loop happens to have no effect during one pass, is removed as it could leak information. This does not invalidate the correctness of the algorithm but only increases the running time.

The structure of this algorithm makes it easy to implement with either of the two graph representations discussed above (list or matrix), making it possible to fully exploit the sparsity of graphs when it is public (we use the matrix representation if it has to be kept secret).

It can be seen that our implementation requires $|V||E|$ secure comparisons, dominating the time required to perform $2|V||E|$ secure multiplications and $5|V||E|$ additions. These complexities grow to $\mathcal{O}(|V|^3)$ when the graph structure is secret, as the graph is then treated as complete (i.e., augmented with edges of infinite weight). Very interestingly, this algorithm is the only one among those we treated in which our solution does not raise any asymptotic overhead (when the structure is public).

Protocol 2: SSP1 protocol based on Bellman-Ford's algorithm

Input: A graph $G = (V, E)$ where V is the list of vertices and E the list of edges, a set of shared weights $[\mathbf{w}]_e$ for each $e \in E$, and a share of the source vertex $[s] \in V$.
Output: The list of immediate predecessors $\mathbf{p}$ and/or total distances $\mathbf{d}$.

for $i \leftarrow 1$ **to** $|V|$ **do**
  $\mathbf{p}_i \leftarrow [0]$; $\mathbf{d}_i \leftarrow [\top]$;
end
`updatevector`$([\mathbf{d}], [\mathbf{s}], [0])$;
for $i \leftarrow 1$ **to** $|V|$ **do**
  for $e \leftarrow 1$ **to** $|E|$ **do**
    $[y] \leftarrow [\mathbf{d}]_{t(e)} - [\mathbf{d}]_{h(e)} + [\mathbf{w}]_e$;
    $[x] \leftarrow [y] < 0$;
    $[\mathbf{d}]_{h(e)} \leftarrow [\mathbf{d}]_{h(e)} + [x] \cdot [y]$;
    $[\mathbf{p}]_{h(e)} \leftarrow ([1] - [x]) \cdot [\mathbf{p}]_{h(e)} + [x] \cdot t(e)$;
  end
end
If there was an update during the very last pass, solution is unbounded ($\exists$ negative cycle). Open required output.

Security. The simulation of an execution of this protocol is immediate from the simulators available for the different calls that can be made by the $\mathcal{F}_{ABB}$ functionality: the simulators corresponding to each of the '$+$', '$\cdot$' and '$<$' operations can be invoked in turn, in an order defined by the protocol execution, and a number of times that only depends on public values ($|V|$ and $|E|$). The same argument will apply to the other protocols we present in this paper, and we will therefore not come back to it.

3.2 Dijkstra's Algorithm

Dijkstra's algorithm computes the shortest path from the source to all vertices in the graph, that is, the shortest path tree rooted at the source. The algorithm is greedy. At each iteration one vertex (the one with the smallest distance label) is permanently updated to the status *scanned*.
Adapting Dijkstra. The fact that Dijkstra's algorithm goes through the graph in an order that depends on the weight of the edges makes it very difficult to efficiently exploit the sparsity of a graph: our best solutions have all a complexity that amounts to the one of a complete graph, and we therefore use the matrix representation in all cases for our protocol.

Protocol 3 (the SSP2 protocol) presents our secure shortest path protocol based on Dijkstra. Note that `updatevector` refers to Protocol 1 and that `updaterow` is the natural extension of `updatevector` for replacing a complete row in a shared matrix. Protocol `binarymin` has been introduced by Toft in [32] to obtain the minimal value out of a vector of shared values. It securely computes a share of the minimal value, [min], along with a share of its index,

Protocol 3: SSP2 protocol based on Dijkstra's algorithm

Input: A graph $G = (V, E)$ where V is the list of vertices and E the list of edges, a matrix of shared weights $[\mathbf{M}]_{i,j}$ for $i, j \in \{1, ..., |V|\}$ and a source vertex $[\mathbf{s}] \in V$.

Output: The vector of distances $\mathbf{d}_i$ and the matrix of predecessor $[\mathbf{P}]_{i,j}$ for $i, j \in \{1, ..., |V|\}$.

```
1  for i, j ← 1 to |V| do
2  |  [P]_{i,j} ← [0]; [d]_i ← [⊤]; [q]_i ← [0];
3  end
4  updatevector([d], [s], [0]);
5  for i ← 1 to |V| do
6  |  [d'] ← [d] + [q];
7  |  [min], [k] ← binarymin([d']);
8  |  updatevector([q], [k], [⊤]);
9  |  for j ← 1 to |V| do
10 |  |  [a] ← ([d] + [M]_{*,j}) · [k];
11 |  |  [c] ← [a] < [d]_j;
12 |  |  [P] ← updaterow([P], j, [P]_j + [c] · ([k] − [P]_j));
13 |  |  [d]_j ← [d]_j + [c] · ([a] − [d]_j);
14 |  end
15 end
16 return [d], [P];
```

[$\mathbf{k}$]. The protocol uses $\mathcal{O}(n)$ comparisons and multiplications. Its overall round complexity is $\mathcal{O}(\log(n))$ rounds. Vector $\mathbf{q}$ records the status of each vertex. An entry is equal to zero if the corresponding vertex has not been scanned yet. It is updated to $\top$ as soon as the vertex has been scanned.

The main differences between the traditional and our secure version of Dijkstra's algorithm happen in the inner loop: *a*) On Line 9, the loop goes through all vertices instead of only considering the neighbors of the current vertex. In particular, this includes an always transparent step where we consider the current vertex and gives a substantial overhead if a public sparse graph is considered. *b*) On Lines 4 – 8 – 12, we need to go through all elements of a row or a vector, even if we know that only one of them is going to be updated. Those two modifications contribute to the same effect: they increase the original complexity of Dijkstra from $\mathcal{O}(|V|^2)$ to $\mathcal{O}(|V|^3)$. More precisely, the exact number of comparisons is $2|V|^2 - 3|V| + 1$ and the exact number of dot products (used for the multiplication of vectors, costing $|V|$ multiplications) is $2|V|^2 - |V|$ for $|V| \geq 4$.

As the comparison protocol we use requires 165 multiplications to compute a comparison, the number of multiplications to compute the shortest path in a complete tree is around $2|V|^3 + 329|V|^2 - 495|V| + 165$.

The switch from quadratic to cubic dominance is at around 165 vertices which is precisely the number of multiplications used by a single comparison.

Our secure version of Dijkstra comes with an overhead of a factor $|V|$ compared to the original one, even when the graph structure can be considered as

public. We note that this was not the case in the work of Brickell [25] who considered running Dijkstra securely as well, but accepted to output the shortest paths step by step. Besides the limitation that this brings when the protocol has to be composed, we also observe that our algorithm can be used to solve problems that could not be solved by Brickell's approach, namely, computing the shortest path between two specific vertices without leaking any other information: their approach indeed leaks the shortest path to *all* vertices.

3.3 Implementation Prototype

We implemented our protocols over the Virtual Ideal Functionality Framework to challenge their performance. We considered a 3-party execution in the information theoretic model with passive security: secret values are shared using Shamir's secret sharing, the BGW protocol is used for multiplication [2], and Toft's protocol is used for comparison [20]. These choices were made for simplicity and ease of prototyping, though much more efficient protocols exist and would have led to considerably shorter running times [28,30]. The computation was performed on a single workstation equipped with an Intel Xeon CPUs X5550 (2.67GHz) and 24GB of memory, running a standard Debian Squeeze.

We ran the two shortest path protocols described above on complete graphs of various sizes. This first showed that Protocol 2 can only be conveniently used for graphs where $|V||E| \approx 10^3$ (a few minutes on a standard laptop): see Table 2.

Number of vertices		4	8	16	32	64	128
Execution times (in seconds)	SSP1	9	63	501	4003	31951	-
	SSP2	9	13	50	217	1018	5622

Table 2. Execution times of Protocols 2 and 3 for a complete shortest path tree.

Our secure versions of Bellman-Ford and Dijkstra have approximately the same complexity for complete graphs. However the quadratic number of comparisons makes it possible to run our secure version of Dijkstra on a 64-vertex complete graph in roughly twice the time as taken by Bellman-Ford on a 16-vertex graph, and we have been able to run it up to a 128-vertex complete graph (i.e., counting 16256 directed edges) in a bit more than an hour.

While these timings might look fairly high, they still make it possible to solve natural problems in a reasonable time. For instance, if the 3-party, 12-vertex problem outlined above could be solved in around 30 seconds.

4 Maximum Flow

In an oriented graph where the edges have a constraint of *capacity*, the maximum flow problem consists in finding the maximum number of units that can be carried from a vertex called *source* to another vertex called *sink*. The *flow* through an edge designates the number of units passing by it. This number cannot exceed the capacity.

This problem has numerous classical applications. In the spirit of our previous examples, one of them could be competing transport companies willing

to determine the capacity they could reach if they decided to realize a joint-venture. It is natural in such a context to expect that these companies will not be willing to disclose their full network structure to each other. As in the case of the shortest path, algorithms solving the maximum flow problem are also very useful as subroutines for solving other problems. The minimum cut problem is one such traditional example, which can be solved using $\mathcal{O}(|V|)$ invocations of the maximum flow algorithm. Solving this problem is then useful to determine where the weak points of the joint network would be.

Although we investigated many different algorithm for a transportation in MPC, this paper only presents two secure protocols based on the Edmonds-Karp's and the Push-Relabel algorithms.

4.1 Edmonds-Karp's Algorithm

The basic idea of the Edmonds-Karp's (EK) algorithm is to find an augmenting path in the residual graph that is the graph in which the edges are weighted by their residual capacity, i.e., the capacity minus the current flow. Each augmenting path increases the total flow so that the algorithm eventually terminates when there are no augmenting paths left. The increase is monotonic and paths are considered once only. Typically, the EK algorithm uses a breadth-first search to find the next augmenting path.

The asymptotic complexity of the traditional EK algorithm is $\mathcal{O}(|V||E|^2)$, which we can match securely (see Appendix B.) As we have seen in the case of the shortest path problem, this complexity will be prohibitive even for very small graphs if they are complete. It therefore makes sense to focus our attention on (oriented) strongly sparse graphs, of which we consider the structure to be public. More precisely, we consider graphs in which the number of paths from the source to the sink is fairly small, e.g., bounded by a small polynomial in the number of vertices.

The algorithm is given on input a list containing all the paths sorted in a growing order of length, $\mathbf{p} = (p_1, ..., p_k)$ where k is the number of paths in the graph. This list is not secret as the structure is not, and can therefore be easily constructed in public. Our protocol based on Edmonds-Karp (the SMF1 protocol) is presented in Protocol 4.

The main differences between this protocol and Edmonds-Karp's approach are: *a*) the public enumeration of all the paths instead of building of a breadth-first search for capacity augmenting paths, and *b*) the treatment of all the paths as if they were augmenting.

The SMF1 protocol is correct as the set of all the augmenting paths is contained in the set of all the paths $\mathbf{p}$. Moreover, it ensures the confidentiality of the edge capacities as no information is leaked about which path of $\mathbf{p}$ is augmenting and which is not.

It is easy to see that the SMF1 Protocol requires $\mathcal{O}(k|V|)$ comparisons, as the length of the longest path in the graph is bounded by $|V| - 1$, and $\mathcal{O}(k)$ multiplications. This protocol makes a crucial use of the existence of a small number of paths in the graph, something that we were not able to use in the

Protocol 4: SMF1 maximum flow protocol based on Edmonds-Karp's algorithm

Input: A graph $G = (V, E)$ where V is the list of vertices and E the list of edges, a source vertex $so \in V$, a sink vertex $si \in V$, and a list $\mathbf{p}$ of length k containing the paths between so and si sorted in a growing order of length. A set of capacities $[\mathbf{c}]_e$ and a set of flows $[\mathbf{f}]_e$ initially set to $[0]$ for $e \in E$. $\bar{e}$ is the edge opposite to e.

Output: The maximum flow value from so to si.

while $|\mathbf{p}| > 0$ **do**
  $p \leftarrow \mathbf{pop}(\mathbf{p})$;
  $[r], [\mathbf{i}] \leftarrow \texttt{binarymin}_{e \in p}([\mathbf{c}]_e - [\mathbf{f}]_e)$;
  $[b] \leftarrow [r] > 0$;
  $[a] \leftarrow [b] \cdot [r]$;
  for $e \in p$ **do**
    $[\mathbf{f}]_e \leftarrow [\mathbf{f}]_e + [a]$;
    $[\mathbf{f}]_{\bar{e}} \leftarrow [\mathbf{f}]_{\bar{e}} - [a]$;
  end
end
return $\sum_{e \in S} [\mathbf{f}]_e$ where $S = \{e \in E | h(e) = so\}$;

SSP2 protocol for instance. It is however highly inefficient for dense graph and would have a factorial complexity for complete graphs.

This protocol applies well to our previous example of the three competing logistic companies trying to determine the max flow in their joint networks. If we consider a case with 10 vertices and 37 different paths, the execution takes less than a minute as shown in Table 3.

Number of paths	2	4	8	14	37	86	135
Number of edges	22	21	25	25	32	30	30
Execution times (in seconds)	3	6	9	18	40	94	148

Table 3. Execution times of Protocol 4 for 10-vertex graphs.

4.2 Push-Relabel Privacy-Preserving Implementation

The Push-Relabel algorithm, also called relabel-to-front when implemented with a FIFO list, introduces two additional attributes for the vertices, the height and the excess. An edge is called admissible if it goes from a higher to a lower vertex. The algorithm alternatively pushes the excess along admissible edges and increases the height of the vertices until all excess has been pushed to the sink or back to the source.

The basic operation of the algorithm is Push/Relabel applied to a given vertex. This operation pushes all the excess through incident admissible edges (updating the excesses of incident vertices accordingly). Finally, in case not all the excess has been pushed, the elevation of the vertex is minimally increased so as to create at least one more admissible edge, and Push/Relabel terminates.

Throughout the algorithm a list L with vertices with positive excess (except the source and the sink) is maintained. At each iteration, one vertex of L is selected and Push/Relabel is applied. The algorithm terminates when the list is empty. In the FIFO implementation, the next vertex of L to be treated is selected in the FIFO order. This FIFO Push/Relabel algorithm terminates in $\mathcal{O}(|V|^3)$ operations.

Our protocol based on Push-Relabel is presented in Protocol 5. The main differences between this protocol and the traditional Push/Relabel algorithm are as follows : *a*) when Push/Relabel is applied to a vertex with zero excess, no update of the elevation is performed at the end, *b*) in each phase, treat *all* vertices except the source and the sink, in a fixed order agreed between the players, and *c*) during each Push/Relabel operation applied to a vertex i, the order in which the edges (i, j) are considered is fixed and agreed in advance between the players. It is clear that these changes do not modify the correctness of the original algorithm.

Protocol 5: A phase of the SMF2 protocol based on Push/Relabel

Input: A complete graph $G = (V, E)$ where V is the list of vertices and E the list of edges. A vertex i to be treated, a vector of elevations $[\mathbf{h}]$, a matrix of residual capacities $[\mathbf{R}]$ and a vector of excesses $[\mathbf{e}]$.

Output: Update of the elevations $[\mathbf{h}]$, the residual capacities $[\mathbf{r}]$ and the excesses $[\mathbf{e}]$.

$[\delta] \leftarrow 2|V|$; **for** $j \leftarrow 1$ **to** $|E|$ **do**
  $[\alpha] \leftarrow [\mathbf{h}]_i > [\mathbf{h}]_j$;
  $[x] \leftarrow \min([\mathbf{e}]_i, [\mathbf{R}]_{i,j})$;
  $[y] \leftarrow [\alpha].[x]$;
  $[\mathbf{R}]_{i,j} \leftarrow [\mathbf{R}]_{i,j} - [y]$; $[\mathbf{R}]_{j,i} \leftarrow [\mathbf{R}]_{j,i} + [y]$;
  $[\mathbf{e}]_i \leftarrow [\mathbf{e}]_i - [y]$; $[\mathbf{e}]_j \leftarrow [\mathbf{e}]_j + [y]$;
  $[\delta] \leftarrow \min([\delta], [\mathbf{h}]_j + 2|V| \cdot [\alpha])$;
end
$[\alpha] \leftarrow [\mathbf{e}]_i > 0$;
$[\mathbf{h}]_i \leftarrow [\mathbf{h}]_i \cdot (1 - [\alpha]) + ([\delta] + 1) \cdot [\alpha]$;

Moreover, it can be verified that the relabel-to-front algorithm terminates in maximum $4|V|^2 - 10|V| + 12$ complete phases. Therefore we obtain an "all-cases" complexity of $\mathcal{O}(|V|^2|E|)$, both in comparisons and multiplications. Note that this does not match the FIFO complexity, because we scan all edges at each pass, even when the excess of the tail vertex is zero.

The complexity of this algorithm remains lower than the one of the original Edmonds-Karp and it is asymptotically better than the optimized version of Edmonds-Karp presented in Section 4.1 for graphs with vertices of high degree. However, the running time of Protocol SMF2 remains very high. Experiments showed that the use of a traditional halting criterion at the end of each SMF2 phase (i.e. nothing has been pushed) results in dramatic running time improvements. However it also demonstrated a huge variability (the algorithm may halt after a single phase), which suggests that a substantial amount of information

could be derived from it. Quantifying this information is left for future work, and its impact is likely to depend on the application.

5 Conclusion

We proposed two protocols for securely computing shortest paths as well as two protocols for securely computing maximum flows in graphs. Besides the interest that these protocols have in the numerous contexts in which their insecure counterparts have found applications in the past (possibly relying on a trusted third party), our investigation raised interesting complexity gaps between centralized algorithms and secure protocols, ranging from a constant to something growing like the number of vertices in the graphs. It is then natural to wonder whether these gaps, when they arise, can be decreased. Various avenues appear for that purpose:

- Design efficient datastructures adapted to the investigated problems. For instance, the recent work of Toft [12] on priority queues could lead to considerably more efficient versions of our secure shortest-path protocols. In particular, whether data structures similar to dynamic trees or Fibonacci heaps are implementable in a secured setting without revealing the execution flow remains an open question.
- Investigate whether secure comparisons, which often are a bottleneck, can be traded for other, cheaper, arithmetic operations. This raises unusual questions from a traditional algorithmic point of view, as comparisons are usually considered as basic operations.

Considering other standard combinatorial problems could also provide new insights. The protocols and results presented in the paper are prototypes that validate the theoretical complexity evaluations. While the running times given for the protocols look unpractical for large graphs, this issue must be put in perspective. Indeed, an implementation for concrete applications should definitively be improved by relying on lower level programming languages and optimized underlying libraries. Various optimization techniques (see, e.g., [28,30]) would lead to performance increases of several orders of magnitude, as was observed in the case of the AES during the last 3 years for instance (see [29] and the references within).

Acknowledgements

This research was supported by the WIST Walloon Region project CAMUS. Edouard Cuvelier and Sophie Mawet are funded by a FRIA grant of the F.R.S.-FNRS. Mathieu Van Vyve is supported by the Belgian IAP Program initiated by the Belgian State, Prime Minister's Office, Science Policy Programming. The scientific responsibility is assumed by the authors. The authors are grateful to Claudio Orlandi and the anonymous reviewers for their constructive feedback. They also sincerely thank Sylvie Baudine for her help in improving the paper.

References

1. Yao, A.C.C.: Protocols for secure computations (extended abstract). In: 23rd Annual Symposium on Foundations of Computer Science, IEEE (1982) 160–164
2. Ben-Or, M., Goldwasser, S., Wigderson, A.: Completeness theorems for non-cryptographic fault-tolerant distributed computation. In: STOC, ACM (1988) 1–10
3. Chaum, D., Crépeau, C., Damgård, I.: Multiparty unconditionally secure protocols. In: STOC, ACM (1988) 11–19
4. Goldreich, O., Micali, S., Wigderson, A.: How to play any mental game or a completeness theorem for protocols with honest majority. In: STOC, ACM (1987) 218–229
5. Bogetoft, P., Damgård, I., Jakobsen, T.P., Nielsen, K., Pagter, J., Toft, T.: A practical implementation of secure auctions based on multiparty integer computation. In: Financial Cryptography. Volume 4107 of LNCS., Springer (2006) 142–147
6. Cramer, R., Franklin, M.K., Schoenmakers, B., Yung, M.: Multi-autority secret-ballot elections with linear work. In: EUROCRYPT. Volume 1070 of LNCS., Springer (1996) 72–83
7. Barni, M., Failla, P., Kolesnikov, V., Lazzeretti, R., Sadeghi, A.R., Schneider, T.: Secure evaluation of private linear branching programs with medical applications. In: ESORICS. Volume 5789 of LNCS., Springer (2009) 424–439
8. Sadeghi, A.R., Schneider, T., Wehrenberg, I.: Efficient privacy-preserving face recognition. In: ICISC. Volume 5984 of LNCS., Springer (2009) 229–244
9. Pinkas, B., Schneider, T., Smart, N.P., Williams, S.C.: Secure two-party computation is practical. In: ASIACRYPT. Volume 5912 of LNCS., Springer (2009) 250–267
10. Kocher, P.C.: Timing attacks on implementations of diffie-hellman, rsa, dss, and other systems. In: CRYPTO, Springer-Verlag (1996) 104–113
11. Damgård, I., Nielsen, J.B.: Universally composable efficient multiparty computation from threshold homomorphic encryption. In: CRYPTO. Volume 2729 of LNCS., Springer (2003) 247–264
12. Toft, T.: Secure data structures based on multi-party computation. In: PODC, ACM (2011) 291–292
13. Ben-David, A., Nisan, N., Pinkas, B.: Fairplaymp: a system for secure multi-party computation. In: CCS, ACM (2008) 257–266
14. Burkhart, M., Strasser, M., Many, D., Dimitropoulos, X.: Sepia: privacy-preserving aggregation of multi-domain network events and statistics. In: Proceedings of the 19th USENIX conference on Security. USENIX Security'10, USENIX (2010)
15. Bogdanov, D., Laur, S., Willemson, J.: Sharemind: A Framework for Fast Privacy-Preserving Computations. In: Proceedings of the 13th ESORICS. Volume 5283 of LNCS., Springer (2008) 192–206
16. Geisler, M.: Cryptographic protocols: theory and implementation. PhD thesis, Aarhus University Denmark, Department of Computer Science (2010)
17. Sleator, D.D., Tarjan, R.E.: A data structure for dynamic trees. J. Comput. Syst. Sci. **26**(3) (June 1983) 362–391
18. Fredman, M.L., Tarjan, R.E.: Fibonacci heaps and their uses in improved network optimization algorithms. J. ACM **34**(3) (1987) 596–615
19. Shamir, A.: How to share a secret. Commun. ACM **22**(11) (1979) 612–613
20. Toft, T.: Primitives and Applications for Multi-party Computation. PhD thesis, Department of Computer Science, Aarhus University (2007)

21. Kruger, L., Jha, S., Goh, E.J., Boneh, D.: Secure function evaluation with ordered binary decision diagrams. In: ACM CCS, ACM (2006) 410–420
22. Ishai, Y., Paskin, A.: Evaluating branching programs on encrypted data. In: Theory of Cryptography, TCC 2007. Volume 4392 of LNCS., Springer (2007) 575–594
23. Brickell, J., Porter, D.E., Shmatikov, V., Witchel, E.: Privacy-preserving remote diagnostics. In: ACM CCS. CCS '07, ACM (2007) 498–507
24. Barni, M., Failla, P., Lazzeretti, R., Sadeghi, A.R., Schneider, T.: Privacy-preserving ECG classification with branching programs and neural networks. IEEE TIFS **6**(2) (June 2011) 452–468
25. Brickell, J., Shmatikov, V.: Privacy-preserving graph algorithms in the semi-honest model. In: ASIACRYPT. Volume 3788 of LNCS. Springer (2005) 236–252
26. Goldreich, O., Ostrovsky, R.: Software protection and simulation on oblivious rams. J. ACM **43**(3) (May 1996) 431–473
27. Damgård, I., Meldgaard, S., Nielsen, J.B.: Perfectly secure oblivious ram without random oracles. In: Proceedings of the 8th TCC, Springer-Verlag (2011) 144–163
28. Bendlin, R., Damgård, I., Orlandi, C., Zakarias, S.: Semi-homomorphic encryption and multiparty computation. In: EUROCRYPT. Volume 6632 of LNCS., Springer (2011) 169–188
29. Damgård, I., Keller, M., Larraia, E., Miles, C., Smart, N.: Implementing aes via an actively/covertly secure dishonest-majority mpc protocol. In: Security and Cryptography for Networks. Volume 7485 of LNCS., Springer (2012) 241–263
30. Damgård, I., Pastro, V., Smart, N.P., Zakarias, S.: Multiparty computation from somewhat homomorphic encryption. In: CRYPTO. Volume 7417 of LNCS., Springer (2012) 643–662
31. Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.: Introduction to Algorithms, Third Edition. 3rd edn. The MIT Press (2009)
32. Toft, T.: Solving linear programs using multiparty computation. In: Financial Cryptography. Volume 5628 of LNCS., Springer (2009) 90–107

A Bounds

The size of the ring $\mathbb{Z}_m$ has to be chosen carefully to prevent overflows. For each protocol presented in this paper, we provide the bounds of m and the value of $\top$ in Figure 1. These bounds depend on numbers such as the maximum weight $\mathbf{w}$ or the maximum capacity $\mathbf{c}$ allowed for the edges. These maxima are agreed in advance by the players. Remark that $\top$ is smaller than m. Most comparison protocols require a much larger m than the values to compare. This dependence is taken into account via a function f.

Protocol	$\top >$	$m >$
SSP1	$\lvert V\rvert \cdot \mathbf{w}$	$f(\top)$
SSP2	$\lvert V\rvert \cdot \mathbf{w}$	$\lvert V\rvert \cdot f(\top)$

Protocol	$\top >$	$m >$
SMF1	-	$\lvert V\rvert \cdot f(\mathbf{c})$
SMF2	-	$\max(2\lvert V\rvert, \lvert V\rvert \cdot f(\mathbf{c}))$
SMF3	$\mathbf{c}$	$\max(f(\top + \mathbf{c}), \lvert V\rvert \cdot f(\mathbf{c}))$

Fig. 1. Minimal bounds on $\top$ and m to avoid overflows.

B Protocols

Protocol 6 presents the complete Edmonds-Karp's implementation of the maximum flow problem in MPC. The `binarymin` function refers to the function

introduced by Toft in [32] to securely compute the minimum between different values. The `Bellman-Ford` function is a natural adaptation of Protocol 2 that outputs the shortest path from the source to the sink in the form of a vector of $\{[0], [1]\}$ where a [1] at the i-th position indicates that edge i belongs to the augmenting path. Note that the first augmenting path $\mathbf{p}$ is public and given in input. We refer to Figure 1 for the value of $\top$.

Protocol 6: SMF3 maximum flow protocol based on complete Edmonds-Karp's algorithm

Input: A graph $G = (V, E)$ where V is the list of vertices and E the list of edges, a source vertex $so \in V$, a sink vertex $si \in V$. A list of positive capacities $[\mathbf{c}]_i$ for each edge. The first augmenting path $\mathbf{p}$.
Output: The maximum flow value from so to si.

for $i \leftarrow 1$ **to** $|E|$ **do**
  $[\mathbf{f}]_i \leftarrow [0]$; $[\mathbf{w}]_i \leftarrow [1]$;
end
for $i \leftarrow 1$ **to** $|E|$ **do**
  for $j \leftarrow 1$ **to** $|E|$ **do**
    $[c]_j \leftarrow (1 - [\mathbf{p}]_j) \cdot \top + [\mathbf{c}]_j$; $[f]_j \leftarrow [\mathbf{p}]_j \cdot [\mathbf{f}]_j$;
  end
  $[r], [\mathbf{k}] \leftarrow \texttt{binarymin}_{j \in \{1,\ldots,|E|\}}([c]_j - [f]_j)$; $[b] \leftarrow [r] > 0$; $[a] \leftarrow [b] \cdot [r]$;
  for $j \leftarrow 1$ **to** $|E|$ **do**
    $[\mathbf{f}]_j \leftarrow [\mathbf{f}]_j + [\mathbf{p}]_j \cdot [a]$; $[\mathbf{f}]_{\bar{j}} \leftarrow [\mathbf{f}]_{\bar{j}} - [\mathbf{p}]_j \cdot [a]$; $[cond] \leftarrow [\mathbf{c}]_j - [\mathbf{f}]_j = 0$;
    $[\mathbf{w}]_j \leftarrow [cond] \cdot |V| + (1 - [cond]) \cdot [\mathbf{w}]_j$;
  end
  $[\mathbf{p}], [\mathbf{d}] \leftarrow \texttt{SSP1}(G, [\mathbf{w}], so, si)$;
end
return $\sum_{e \in S} [\mathbf{f}]_e$ where $S = \{e \in E | h(e) = so\}$;

The main differences between protocol and the traditional Edmonds-Karp algorithm are as follows :

- Each iteration go through all the edges but only those which form the current path are updated.
- The SSP1 protocol is used instead of the Breath-First-Search protocol to find the smallest augmenting path because there is a serious overhead in a straightforward secure implementation of the BFS. To run the SSP1 protocol, SMF3 maintains a list of shared weights $[\mathbf{w}]$ for the edges where the weight of an edge is [1] when it remains in the residual graph and it is $[|V|]$ otherwise.

It is straightforward to see that the asymptotic complexity of the algorithm is $\mathcal{O}(|V||E|^2)$ as the original algorithm. The number of comparisons is $|V||E|^2 + |E|^2 + |E|$ and the number of multiplications is $|V||E|^2 + 5|E|^2 + |E|$.