Server-Aided Secure Computation with Off-line Parties

Foteini Baldimtsi[1](✉), Dimitrios Papadopoulos[2], Stavros Papadopoulos[3], Alessandra Scafuro[4], and Nikos Triandopoulos[5]

[1] George Mason University, Fairfax, USA
foteini@gmu.edu
[2] Hong Kong University of Science and Technology, Sai Kung, Hong Kong
dipapado@cse.ust.hk
[3] Intel Labs, MIT, Cambridge, USA
stavrosp@csail.mit.edu
[4] North Carolina State University, Raleigh, USA
ascafur@ncsu.edu
[5] Stevens Institute of Technology, Hoboken, USA
ntriando@stevens.edu

Abstract. Online social networks (OSNs) allow users to jointly compute on each other's data (e.g., profiles, geo-locations, etc.). Privacy issues naturally arise in this setting due to the sensitive nature of the exchanged information. Ideally, nothing about a user's data should be revealed to the OSN provider or non-friends, and even her friends should only learn the output of a specific computation. A natural approach for achieving these strong privacy guarantees is via secure multi-party computation (MPC). However, existing MPC-based approaches do not capture two key properties of OSN setting: Users does not need to be online while their friends query the OSN server on their data; and, once uploaded, user's data can be repeatedly queried by the server on behalf of user's friends. In this work, we present two concrete MPC constructions that achieve these properties. The first is an adaptation of garbled circuits that converts inputs under different keys to ones under the same key, and the second is based on 2-party mixed protocols and involves a novel 2-party re-encryption module. Using state- of-the-art cryptographic tools, we provide a proof-of-concept implementation of our schemes for two concrete use cases, overall validating their efficiency and efficacy in protecting privacy in OSNs.

1 Introduction

Secure computation is a cryptographic tool that enables n mutually distrustful parties to compute the output of a function on their combined inputs, while keeping the inputs secret. Originally, the problem of secure computation considered *n equally powerful, fully connected* parties that interact for a *one-time* computation and was mostly regarded as an intriguing theoretical question. However, as more data and services are managed by remote untrusted machines, this tool

DOI: 10.1007/978-3-319-66402-6_8

became increasingly relevant for real-world scenarios over time. Thus, the effort of the community has focused on making secure computation amenable to real applications in ways that can be summarized in the following three directions:

(a) Optimization of Existing Classical Protocols. An amazing line of work focused on improving the concrete efficiency of existing results in the model of equally powerful, fully connected parties such as Yao's garbled circuit [49,50], and GMW [25] and BGW [11]. For example, a sequence of work [10,17,36–38,42, 45,51] showed that classic garbled circuits can be implemented very efficiently, reducing the number of encryptions required for each garbled gate.

(b) Introduction of New Interaction/Computation Models to Reduce the Computational Burden of the Parties. These new models consider distinguished nodes (often called "servers") that carry out most of the computation and communication. For example, [39] considers a *single-server* model, where parties encrypt their inputs using homomorphic encryption, send them to an untrusted server that performs the computation and delivers the encrypted output to the parties who engage in a MPC protocol to decrypt the result. In this way, parties have to do work that is independent of the function complexity, but depends only on the input/output size. While asymptotically advantageous, [39] has poor concrete efficiency. Other works [18,19,23,32,33] have looked at leveraging this "server-aided" model with the additional assumption that the server does not collude with the parties. In this setting, they are able to provide efficient multiparty protocols based on garbled circuits, where parties communicate with the servers upon each computation to provide the encoding of their inputs (some parties need to additional engage with the server requiring communication complexity proportional to the circuit size). A *multi-server* model has also been considered [14,20,31,43] where computation is performed by multiple servers and the non-collusion requirement is moved from client-to-server to server-to-server only.

(c) Introduction of Models Tailored to Specific Real World Applications. Works in this direction proposed models that better reflect real world threat models and interaction patterns. An example of such work is the model for computation over the Internet introduced by Halevi et al. in [27]. In their model there is a single server which is always online but the parties involved in the protocol are not expected to be online. Instead, they connect only when they desire to provide inputs to the computation or learn the output of the protocol. However, every time a new computation needs to be performed, parties must connect and provide fresh encodings of their input (even if it has not changed). This makes [27] very relevant to applications that require a *one-time* computation, such as e-voting, where clients connect once to cast their vote and once to get the election result.

Our Contribution. In this work, we make progress in the last direction, by proposing a new model that fits a specific real world scenario –*Online Social Networks (OSN)*– and we provide two new protocols and respective implementations. OSNs enable users to store information they wish to share with other authorized users –their *friends*– and the latter can access friends' data at their own convenience. As opposed to one-time computation applications which are captured by [27], OSNs allow *repeated* computations over a party's data. E.g., in the friend-finder application of Facebook, called "Nearby Friends", Alice's location is sent to Facebook's server once and can be re-used by her friends several times, without Alice performing any further action. This type of interaction mandates two key properties: (i) *data re-usability*, i.e., personal data that a user may upload once to the OSN server can be repeatedly computed upon (possibly in different ways), and (ii) *friend non-participation*, i.e., a user need not be online when one of her friends requests a computation that involves her data. We introduce a model where parties upload their secret input to a single, untrusted, server in a *one-time* step, and then they do not have to be on-line anymore unless they want to update their own input or they want to compute (via the server) a function on the combined inputs of their friends. Crucially, and in contrast with [27], whenever a friend requests a computation from the server, the other parties do not need to provide a new encoding for their inputs.

Our Model. We consider a *single-server* hosting the OSN, and *multiple users* that form the social network. We represent the OSN as a graph; the users constitute the nodes of the graph, and an edge denotes that the two vertices are friends. Users *upload* their data to the server, and update them at any time. The users agree upon arbitrary queries (i.e., specific computations over their uploaded data) with their friends (e.g., *"who is my geographically nearest friend"*), and each user may repeatedly issue queries to the server about her own and her friends' data. Both upload and query executions involve only the server and a single user, while the remaining users do not need to participate or even to be online.

Our *privacy goals* are: (i) the server learns nothing about the user data or the query results, (ii) the querier learns nothing about her friends' data other than what is inferred from the results, and (iii) the querier learns nothing about non-friends' data. Note that we do not consider the social graph structure to be sensitive. Moreover, we assume that every user allows all of her friends to query on her data, i.e., "friendship" implies access control on one's data. Hiding the graph and supporting more sophisticated access policies are interesting problems that are orthogonal to our work. Our *performance requirements* are: (i) the cost to upload/update a user's input should be constant, and (ii) the constructions should involve lightweight cryptographic tools, with reasonable upload and query times.

Definitional Choices. Our security model has two relaxations. First, we assume that friends do not collude with the OSN provider, they can collude with each other however (*non-friend* collusion with the OSN server is also accepted). This type of security relaxation, first formalized as *bounded-collusions* by Kamara et al. in [32], has since been adopted in a sequence of works [18,19,23,32,33]. We believe that this collusion model is meaningful in OSN applications where friendship implies some level of trust. Note that regardless of the collusion model, in the OSN interaction model (where the server can perform a computation without other users), only a weaker input-privacy can be achieved. Indeed, [27] shows that a collusion between server and any user U_j allows them to learn the *residual function* on many inputs of their choice. Second, we consider the semi-honest model, i.e., we assume that the parties execute the protocols correctly. Although weaker, this model provides full protection against security breaches suffered by OSN providers or by friends. In the full version of the paper [8], we elaborate on some of challenges that arise when moving to the malicious setting where adversaries may arbitrarily deviate from the protocol.

Our Technique: Multi-party Computation from a Two-Party Protocol. Our approach consists of implementing a multi-party functionality, using *strictly two-party protocols* run between a single user and the server. Our key technical contribution is developing "translation" mechanisms to translate input encrypted under a friend's secret key, into data that is encrypted with a common key which is secret shared between the OSN server and the user, but is not known by any of them. In developing this tool, we leverage the assumption that a friend does not collude with the OSN server. In this way, parties upload encodings of their inputs to the server and, any time a party wishes to compute a function, the server will use her friends' encodings and interact with the querier to carry out the computation. This might seem relatively easy to achieve, e.g., if the friend input encodings are all produced under the querier's key, or by establishing fresh shared randomness before every single computation (as in [23]). The former approach requires each friend to produce a separate encoding of her value for each of her friends, leading to considerable overhead for upload. The latter prevents re-usability of values, forcing friends to get involved in someone else's computations. Thus, the challenge in realizing the multi-party OSN functionality from two-party protocols boils down to simultaneously achieving re-usability, friend non-participation and efficient uploads, while employing lightweight cryptographic primitives (such as symmetric or additively homomorphic encryption). At the core of our solutions are mechanisms for *re-randomizing* the encoding of the inputs upon each computation, without involving any party except the querier and the server.

Overview of our Protocols. We design two MPC-based constructions based on well-studied techniques for secure two-party computation, *garbled circuits* [49,50] and *mixed protocols* [13,21,28,34]. Each user independently encrypts a value under her own key and uploads the encryption to the server with constant cost. The difficulty lies in implementing a two-party query protocol on encryptions produced by different keys. We achieve this by having two users exchange common secrets *once* upon establishing their friendship. Using these secrets, the querier can emulate a multi-party protocol by solely interacting with the server.

Our first construction, presented in Sect. 4 is based on garbled circuits. The main idea is that the querier prepares a *selection table* utilizing the common secrets during the query, which allows the server to map the (unknown to the querier) encoded friend inputs to the encoding expected by the querier's circuit. A similar idea was used in [40] for a different setting, namely *garbled RAMs*.

A positive side-effect of this is that is eliminates the need for costly *oblivious transfers* (OT) required in traditional two-party garbled circuit schemes.

Our second construction, presented in Sect. 5, adopts the two-party mixed protocols approach, motivated by the fact that the performance of garbled circuits is adversely affected by functions with large circuit representation. The main idea is to substitute the parts of the computation that yield a large number of circuit gates with arithmetic modules. The latter are implemented via two-party protocols, executed between the querier and the server involving homomorphic ciphertexts. A core component of our solution is a novel two-party *re-encryption protocol*, which enables the server to privately convert the homomorphic ciphertexts of the querier's friends, to ciphertexts under the querier's key. Unlike existing proxy re-encryption schemes [5,6,35], our simple technique maintains the homomorphic properties of ciphertexts, and can be retrofitted into any existing scheme that uses (partially) homomorphic encryption (e.g., [46]), allowing computation over ciphertexts produced with different keys of collaborating users.

Implementation. In Sect. 6, we provide a proof-of-concept implementation and experimentally evaluate its performance for applications that measure closeness under the Euclidean and the Manhattan distance metrics, which are useful in OSNs (e.g., location closeness in Foursquare, or profile closeness in Match.com).

2 Preliminaries

Semi-Homomorphic Encryption. We utilize public-key additively homomorphic schemes (e.g., Paillier [47]). Hereafter, $[\![\cdot]\!]_{pk}$ denotes a ciphertext encrypted with additively homomorphic encryption under key pk. When it is clear from the context we omit pk from the subscript. Given ciphertexts $[\![a]\!], [\![b]\!]$ of a and b under the same key, additively homomorphic encryption allows the computation of the ciphertext of $a+b$ as $[\![a]\!] \cdot [\![b]\!] = [\![a+b]\!]$, where $\cdot$ denotes a certain operation on ciphertexts (e.g., modular multiplication in Paillier). Given $[\![a]\!]$ it allows to efficiently compute $[\![au]\!]$, for a plaintext value u, by computing $[\![a]\!]^u$. Note that $[\![a]\!]^{-u} \equiv [\![a]\!]^{u'}$, where u' is the additive inverse of u in the plaintext space. Moreover, given $[\![a]\!]$ one can produce a fresh re-encryption without the secret key, by generating a new encryption $[\![0]\!]$ of 0, and computing $[\![a]\!] \cdot [\![0]\!]$.

Yao's Garbled Circuits [49,50]**.** This is the de-facto method for secure two-party computation, which was originally proposed for the *semi-honest* model. For readers that are not familiar with the concept of garbled circuits, we include a detailed description in the full version of our paper [8]. At a high level the scheme works as follows: consider two parties, U_q and S (this notation will be helpful later). Suppose that U_q wishes to compute a function f on S's and her own data. First U_q expresses f as a Boolean circuit, i.e., as a directed acyclic graph of Boolean gates such as AND and OR, and sends a "garbled" version of the circuit to S to evaluate it using its own input. Note that U_q does not send her inputs to S, instead her inputs are encoded into the garbled circuit such that

S can not determine what they are. U_q is typically referred to as the *garbler* and S as the *evaluator*.

Mixed Protocols. In garbled circuits, even simple functions may result in a circuit with an excessive number of gates. For instance, textbook multiplication of two ℓ-bit values is expressed with $O(\ell^2)$ gates. Motivated by this, many recent works (e.g. [13,21,28,34]) focus on substituting a large portion of the circuit with a small number of boolean or *arithmetic gates* (i.e., ADD and MUL). The secure evaluation of the Boolean gates is done efficiently via garbled circuits, while that of the arithmetic via schemes like homomorphic encryption or arithmetic secret-sharing, yielding efficient protocols for functionalities like comparison of encrypted values [7,15,22]. Such protocols, referred to as *mixed protocols*, also provide ways for converting from one to the other, i.e., from garbled circuit values to homomorphic encryptions and vice versa. Note that all possible functions can be expressed as combinations of additions and multiplications, thus mixed protocols exist for every function. Without loss of generality, in the sequel we assume that both parties' initial inputs to every mixed protocol are encrypted under an additively homomorphic encryption scheme, and with one party's key.

Figure 1 illustrates two examples of mixed protocols evaluating functions f and g, denoted as π_f and π_g. Function f is expressed as the composition $f_2 \circ f_1$, where f_1 is represented with an arithmetic circuit evaluated by a homomorphic encryption protocol π_{f_1}, and f_2 is represented by a Boolean circuit evaluated by a garbled circuit protocol π_{f_2}. Moreover, there exists a secure conversion protocol π_C from homomorphically encrypted values to garbled inputs. Function g is expressed as $g_2 \circ g_1$, where π_{g_1} is based on a garbled circuit, π_{g_2} on homomorphic encryption, and $\pi_{C'}$ is the corresponding secure conversion protocol. Since we assume that the inputs are homomorphic encryptions, π_g first requires their conversion to garbled values via π_C. Given f, the challenge is to find a decomposition to simpler functions $f_1, \ldots, f_n$, where each f_i is expressed either as a Boolean or arithmetic circuit, such that the mixed protocol is more efficient than evaluating f solely with a garbled circuit. [13,21,28,34] addressed this challenge by providing automated tools for decomposing certain functions, as well as appropriate conversions. If there exist protocols for the secure evaluation of all f_i's, and given that the conversion protocols are secure, the composition of these protocols *securely evaluates* f [16]. In the full version, we present two mixed protocols we use for private multiplication and comparison of encrypted values.

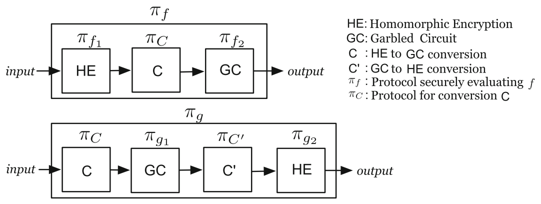

Fig. 1. Examples of mixed protocols

3 Problem Formulation

Our setting involves a server S, and a set of users $\mathcal{U}$. The server maintains an (initially empty) undirected graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$. A vertex $v_i \in \mathcal{V}$ represents the information that the server knows about a user $U_i \in \mathcal{U}$. An edge $e_{ij} \in \mathcal{E}$ between vertices v_i and v_j stores information about the (bidirectional) friendship between U_i and U_j. By $\mathcal{G}_i$ we denote the friend list of U_i. Table 1 summarizes the notation used in the rest of the paper.

Table 1. Summary of symbols

Symbol	Meaning
U_i, U_q, S	User i, querier, server
$\mathcal{G} = (\mathcal{V}, \mathcal{E})$	Graph with vertices $v_i \in \mathcal{V}$ and edges $e_{ij} \in \mathcal{E}$
$\mathcal{G}_i$	Friend list of U_i
E_k	Symmetric encryption under key k
F_K	Pseudorandom function (PRF) under key K
$[\![\cdot]\!]_{pk}$	Additively homomorphic encryption under key pk
x_i	Input of U_i
ℓ	Length of x_i
$x_i[l]$	l^{th} bit of x_i
GC	Garbled circuit
[0.05cm] X^b_{jl}	Encryption of $b = x_j[l]$ in our generic protocol
[0.1cm] w^b_{jl}	Garbled value for $b = x_j[l]$ in our generic protocol
[0.1cm] s^b_{jl}	Key for selecting w^b_{jl} in our generic protocol
[0.1cm] T_q	Selection table of U_q in our generic protocol

3.1 Security Definition

We formalize the privacy requirements for the OSN model in the *semi-honest* setting, using the *ideal/real world* paradigm [25]. Specifically, we first define the *ideal functionality*, $\mathcal{F}_{\mathsf{OSN}}$, that captures the security properties we want to guarantee in the OSN model. In the ideal world, $\mathcal{F}_{\mathsf{OSN}}$ is implemented by a *trusted third party* that privately interacts with all parties, while the latter do not interact with each other. In this setting, parties can only obtain the information allowed by $\mathcal{F}_{\mathsf{OSN}}$. In the real world, the trusted party is replaced by a protocol π executed jointly by the parties. Informally, π securely realizes $\mathcal{F}_{\mathsf{OSN}}$, if whatever can be learned by an adversary $\mathcal{A}$ running the real protocol and interacting with other parties, can be simulated by an algorithm, called the *simulator* Sim, interacting only with the trusted party. We define here our ideal functionality, which meets the privacy goals stated in Sect. 1. Note that $\mathcal{F}_{\mathsf{OSN}}$ is a reactive functionality that responds to messages received by parties.

Ideal Functionality $\mathcal{F}_{\mathsf{OSN}}$. Interact with a set $\mathcal{U}$ of users and a server S. Initialize an empty graph $\mathcal{G}$.

- $\mathsf{Join}(U_i)$. Upon receiving a Join request from user U_i, if vertex v_i already exists in $\mathcal{G}$ do nothing; else, add v_i to $\mathcal{G}$, and send (Join, U_i) to S and $(\mathsf{Join}, \mathsf{ok})$ to U_i.
- $\mathsf{Connect}(U_i, U_j)$. Upon receiving a Connect request from users U_i, U_j, if $\mathcal{G}$ contains edge e_{ij} do nothing; else, add e_{ij} to edge list $\mathcal{E}$ of $\mathcal{G}$, and send $(\mathsf{Connect}, U_j, U_i)$ to S and $(\mathsf{Connect}, U_i, U_j, \mathsf{ok})$ to U_i and U_j.
- $\mathsf{Upload}(U_i, x_i)$. Upon receiving an Upload request from U_i with input x_i, if v_i does not exist, do nothing; otherwise, store x_i in v_i. Finally, send (Upload, U_i) to S and $(\mathsf{Upload}, \mathsf{ok})$ to U_i.
- $\mathsf{Query}(U_q, f)$. Upon receiving a Query request from user U_q for function f, retrieve the adjacent vertices of v_q from $\mathcal{G}$, then compute $y = f(\alpha, x_q, \{x_j \mid \forall j : U_j \in \mathcal{G}_q\})$, where α is a query-dependent parameter. Finally, send (out, y) to U_q and (Query, f, U_q) to S.

Ideal World Execution. Each user $U_i \in \mathcal{U}$ receives as input $\mathsf{in}_i = (\mathcal{G}_i, \mathbf{x}_i, r_i, \mathbf{f}_i)$, where $\mathcal{G}_i$ is U_i's friend list, $\mathbf{x}_i = (x_i^{(1)}, x_i^{(2)}, \ldots)$ is the sequence of inputs that U_i uses in her Upload queries, r_i represents U_i's random tape, and $\mathbf{f}_i = (f_i^{(1)}, f_i^{(2)}, \ldots)$ is the functions used in her Query requests. $\mathcal{G}_i$ dictates the calls to Connect, $\mathbf{x}_i$ the calls to Upload, and $\mathbf{f}_i$ the calls to Query. Note that the functionality keeps only the x_i value of the *latest* Upload. Finally, the server's only input is the random tape r_S. Each U_i hands her $\mathsf{in_i}$ to the trusted party implementing $\mathcal{F}_{\mathsf{OSN}}$, and receives only the outputs of her Query executions and the acknowledgments of the Join, Upload and Connect requests. We denote the output of U_i from the interaction with $\mathcal{F}_{\mathsf{OSN}}$ by out_i. S receives only (ordered) notifications of the requests made by the users. We denote the output of S from the interaction with $\mathcal{F}_{\mathsf{OSN}}$ by out_S.

Real World Execution. In the real world, there exists a protocol specification $\pi = \langle \mathcal{U}, S \rangle$, played between the users in $\mathcal{U}$ and the server S. Each user $U_i \in \mathcal{U}$ has as input $\mathsf{in}_i = (\mathcal{G}_i, \mathbf{x}_i, r_i, \mathbf{f}_i)$, defined as in the ideal world, whereas S has random tape r_S. An adversary $\mathcal{A}$ can corrupt *either* a set CorrUsers of users *or* the server S (but not both). We denote by $\mathsf{view}^{\pi}_{\mathcal{A}_{\mathsf{CorrUsers}}}$ the view of the real adversary $\mathcal{A}$ corrupting users U_i in the set CorrUsers. This consists of the input of every $U_i \in \mathsf{CorrUsers}$, and the entire transcript Trans_i obtained from the execution of protocol π between the server and every $U_i \in \mathsf{CorrUsers}$. Respectively, view^{π}_S denotes the view of the corrupted server, which contains r_S and transcripts Trans_i obtained from the execution of π with every $U_i \in \mathcal{U}$.

Bounded Collusions. Note that, based on the above description, our scheme does not allow *any* user to collude with the server. However, it is straightforward to extend our security definition to permit users that are not connected with the querier in $\mathcal{G}$ to collude with the server. Intuitively, since such users share no data with the querier, the coalition of S with them offers no additional knowledge. We choose not to formulate such collusions to alleviate our notation.

More Elaborate Access Policies. One extension of our model would be to allow users to specify more elaborate access policies, e.g., that certain friends may only ask for certain computations, limit the number of times their data may be queried, or revoke a friendship entirely. In the semi-honest model with bounded collusions all these can be trivially achieved by simply specifying this to the server who notifies the affected parties (which can be implemented by whatever access policy mechanism the OSN provider operates). These become more challenging problems in the malicious setting which we leave as future work.

Definition 1. *A protocol* $\pi = \langle \mathcal{U}, S \rangle$ securely realizes *the functionality* $\mathcal{F}_{\mathsf{OSN}}$ *in the presence of* static, semi-honest adversaries *if, for all* λ*, it holds that:*

Server Corruption: There exists PPT Sim_S *such that* $\mathsf{Sim}_S(1^\lambda, \mathsf{out}_S) \cong \mathsf{view}_{\mathcal{A}_S^\pi}$.
Users Corruption: For all sets $\mathsf{CorrUsers} \subset \mathcal{U}$*, there exists PPT* $\mathsf{Sim}_{\mathsf{CorrUsers}}$ *such that:* $\mathsf{Sim}_{\mathsf{CorrUsers}}(1^\lambda, \mathsf{in}_i, \mathsf{out}_i\}_{U_i \in \mathsf{CorrUsers}}\} \cong \mathsf{view}_{\mathcal{A}^\pi_{\mathsf{CorrUsers}}}$.

3.2 Our General Approach

This subsection presents an approach that is common in both our constructions for realizing the functionality $\mathcal{F}_{\mathsf{OSN}}$. It also provides a more practical interpretation of the party interaction in our protocols, which will facilitate their presentation in the next sections. The key idea in this approach is twofold: (i) every user has her own key, which she uses to encrypt her input in Upload, and (ii) during $\mathsf{Connect}$, the two involved users exchange keys that are used in subsequent Query executions initiated by either user. The protocol interfaces are as follows:

– $\mathsf{Join}\langle U_i(1^\lambda), S(\mathcal{G})\rangle$: On input security parameter λ, U_i generates a key K_i and notifies the server S that she joins the system. The output of the server is graph $\mathcal{G}'$, where vertex v_i is added into $\mathcal{V}$ of $\mathcal{G}$.
– $\mathsf{Connect}\langle U_i(K_i), U_j(K_j), S(\mathcal{G})\rangle$: U_i and U_j establish keys $k_{i \to j}$ and $k_{j \to i}$ via S. S creates an edge e_{ij} that stores the two keys and adds it to $\mathcal{E}$ of $\mathcal{G}$. The private output of S is the updated graph $\mathcal{G}'$.
– $\mathsf{Upload}\langle U_i(K_i, x_i), S(\mathcal{G})\rangle$: User U_i encodes her data x_i (for simplicity we assume x_i is a single value, but it is straightforward to extend our model for vectors of values) into c_i under her secret key K_i and sends it to S who stores the received value into v_i in $\mathcal{G}$. For simplicity, we assume that v_i stores a single c_i, and every Upload execution overwrites the previous value. The private output of S is the updated $\mathcal{G}'$.
– $\mathsf{Query}\langle U_q(K_q, \alpha), S(\mathcal{G})\rangle(f)$: On input function f and auxiliary parameters α, U_q interacts with S and learns the value $y = f(\alpha, x_q, \{x_j \mid \forall j : U_j \in \mathcal{G}_q\})$, using keys $\{k_{j \to q} \mid \forall j : U_j \in \mathcal{G}_q\}$.

We describe the execution of the interfaces in Fig. 2. The left part of the figure illustrates the party interaction and the right part depicts how the server's graph $\mathcal{G}$ changes by the protocol execution. In Join, U_1 generates her key and notifies S, who adds v_1 to the graph. In $\mathsf{Connect}$, U_2 and U_3 establish $k_{2 \to 3}, k_{3 \to 2}$ and send them to S. The latter adds edge e_{23} (storing the two values) to $\mathcal{G}$. In Upload, U_4 encodes her input x_4 under her key K_4 into c_4, and sends it to S who stores it in vertex v_4 (overwriting any previous value). Finally, in Query, U_5 engages in a *two-party* protocol with S and computes the output of a function f on α and (x_5, x_6, x_7, x_8). The latter are the current plain data of U_5 and her friends U_6, U_7 and U_8, respectively. Note that S possesses only the *encryptions* of these values, namely (c_5, c_6, c_7, c_8). Also, (c_6, c_7, c_8) were produced by U_6, U_7, U_8 with

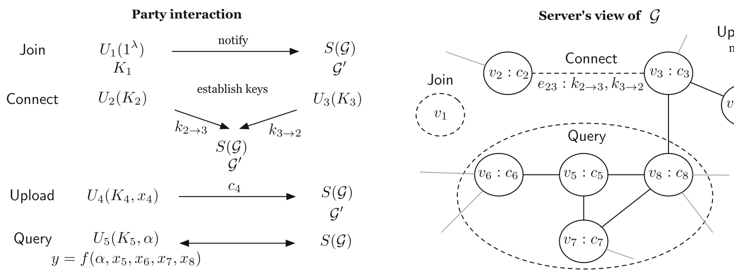

Fig. 2. Example protocol executions of our scheme

keys (K_6, K_7, K_8), which are not known to U_5 and S. Performing the computation without these keys is the main challenge in our model, since U_6, U_7, U_8 should *not participate* in this phase. As we shall see, our solutions overcome this challenge using the keys $k_{6\to5}, k_{7\to5}, k_{8\to5}$ that U_5 received upon connecting with U_6, U_7, U_8, respectively. A final remark concerns our decision to store keys $k_{i\to j}$ at the server. Alternatively, each user U_j could store all keys $k_{i\to j}$ locally. However, this would lead to a linear storage cost in the number of friends at the end of Connect at U_j. In Sects. 4 and 5 we show how to instantiate our general approach using garbled circuits and mixed protocols, respectively.

4 Garbled Circuit Protocol

Suppose querier U_q wishes to compute a function f. She first expresses f as a Boolean circuit, garbles it (see Sect. 2), and sends it to the server S along with the garbled values corresponding to her input x_q. In order to evaluate the circuit, S needs the garbled values corresponding to the input x_j of every U_j in friend list $\mathcal{G}_q$ of U_q. *How can S and U_q figure out which garbled values U_q should send to S for the input x_j of U_j, without knowing x_j?*

There are approaches [23,32,33] that solve this problem by having each friend $U_j \in \mathcal{G}_q$ interact with U_q once to agree on a *common randomness*. Then, whenever U_q wishes to evaluate f, she creates a garbled circuit using the common randomness and sends it to S, whereas, all friends send their garbled values to S. This means that all friends must actively participate in Query. Note also that the garbled values *cannot be reused*, and, thus, the friends must participate in the protocol *every time* U_q executes Query. Other approaches [18,43] instead enable the transferring of the friends' garbled values via an "outsourced" OT, run between the server S, the querier U_q and each friend U_j in $\mathcal{G}_q$. This approach gets rid of the common randomness, and hence, the pre-processing phase, but it still requires all friends to be on-line (to run the outsourced OT) for each Query request.

We take a different approach that capitalizes on the pre-processing phase (Connect), in a way that turns Query into a *strictly two-party* protocol run between U_q and S, and no friends need to be involved. In our solution, each user U_i has a secret key K_i for a pseudorandom function (PRF), that exchanges with a friend upon each Connect phase. This is done via the server, using their respective public keys. To upload her secret input x_i, U_i encodes each bit of x_i as a PRF evaluation under key K_i, and sends them to S. Finally, the Query is performed as follows. Querier U_q first prepares a garbled circuit for the function f and sends it to S, together with the garbled values corresponding to her *own* input. The garbled values of each friend U_i are instead *encrypted* with keys derived from the PRF evaluations under K_i, which S uses to evaluate the circuit. We illustrate this idea using the example of Fig. 3 which focuses on the evaluation of an AND gate A. For a comparison of the modifications required by our scheme compared to standard garbled circuits, see the full version of the paper [8]. The top wire of A corresponds to the first bit of x_q (i.e., $x_q[1]$) belonging to U_q, whereas the bottom wire to the l^{th} bit of x_j (i.e., $x_j[l]$) of U_j for some $l \in [\ell]$. Moreover, $x_q[1] = 1$ and $x_j[l] = 1$. Upon Upload, U_j sends to the server an encryption of $x_j[l]$ as $X^1_{jl} = F_{K_j}(1, l, r_j)$, where F is a PRF and r_j is a random nonce sent to S along with X^1_{jl} (note that, if $x_j[l]$ was 0, U_j would send $X^0_{jl} = F_{K_j}(0, l, r_j)$).

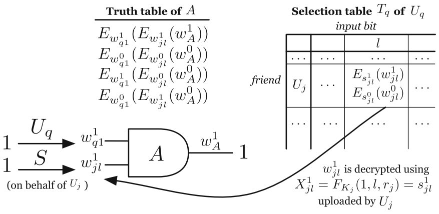
Truth table of A
$E_{w^1_{q1}}(E_{w^1_{jl}}(w^1_A))$
$E_{w^0_{q1}}(E_{w^0_{jl}}(w^0_A))$
$E_{w^1_{q1}}(E_{w^0_{jl}}(w^0_A))$
$E_{w^0_{q1}}(E_{w^1_{jl}}(w^0_A))$
Selection table T_q of U_q
input bit
l
friend
U_j
$E_{s^1_{jl}}(w^1_{jl})$
$E_{s^0_{jl}}(w^0_{jl})$
1
U_q
w^1_{q1}
1
S
w^1_{jl}
(on behalf of U_j)
A
w^1_A
1
w^1_{jl} is decrypted using
$X^1_{jl} = F_{K_j}(1, l, r_j) = s^1_{jl}$
uploaded by U_j

Fig. 3. Use of selection tables in garbled circuits

In Query, U_q garbles gate A, obtaining all garbled values w, and producing the garbled truth table for A. She then sends to S the garbled truth table and her garbled value w^1_{q1} corresponding to $x_q[1]$. When sending the above, U_q does not know the actual value of $x_j[l]$ and, thus, she does not know if she should send w^0_{jl} or w^1_{jl}. Nevertheless, in Connect, U_j provided U_q with the means to help S *select* between w^0_{jl}, w^1_{jl}. Specifically, S stores $k_{j \to q}$ which encrypts U_j's K_j under U_q's public key. U_q retrieves $k_{j \to q}$ and nonce r_j (uploaded by U_j along with X^1_{jl}) from S. Next, she decrypts K_j from $k_{j \to q}$ and computes *selection keys* $s^0_{jl} = F_{K_j}(0, l, r_j)$ and $s^1_{jl} = F_{K_j}(1, l, r_j)$. Then, she encrypts U_j's possible garbled values using these keys, producing $E_{s^1_{jl}}(w^1_{jl})$ and $E_{s^0_{jl}}(w^0_{jl})$. She stores this pair in *random order* into a two-dimensional *selection table* $T_q[j, l]$, where rows represent U_q's friends and columns the input bits. In the general construction U_q fills the $|\mathcal{G}_q| \cdot \ell$ entries of T_q and sends it to S with the garbled circuit.

Upon receiving the garbled circuit and T_q, S attempts to decrypt the values in $T[j, l]$, using X^1_{jl} as the decryption key. Since, by construction, $X^1_{jl} = s^1_{jl}$,

S successfully decrypts *only* w^1_{jl}. Note that this can be seen as an OT played between S and user U_q, where S uses the knowledge of the encrypted input X^1_{jl} to select the garbled value w^1_{jl}. The rest of the circuit evaluation proceeds normally, noting that the final garbled output is decrypted by the querier (i.e., the output mapping to plaintext is not disclosed to the server).

The idea of mapping encoded bits (unknown to the garbler) to the appropriate garbled values expected by a circuit, appeared first in [40] for a different problem, namely to construct *garbled RAMs*. In that setting, a single user wishes to execute a program in a RAM outsourced to some untrusted server, without the latter ever learning the contents of the RAM. In our setting, the unknown garbled inputs of U_q's friends can be perceived as the unknown state of the server's RAM before the evaluation of our garbled circuit.

Construction. We follow the notation of Table 1 and assume that GC is constructed and evaluated as explained at a high level in Sect. 2, without formalizing the algorithms to alleviate notation. Let F be a PRF, (E, D) a CPA-secure symmetric-key encryption scheme, and let (E', D') be a CPA-secure public-key encryption scheme. We assume that encryption algorithms are randomized. Our garbled circuit protocol, π_{GP}, works as follows.[1]

1. $\mathsf{Join}\langle U_i(1^\lambda), S(\mathcal{G})\rangle$: On input 1^λ, U_i randomly chooses a PRF key $K_i \in \{0,1\}^\lambda$, and sends her public-key pk_i to S. S adds v_i initialized with value pk_i into $\mathcal{V}$ of $\mathcal{G}$.
2. $\mathsf{Connect}\langle U_i(K_i), U_j(K_j)\rangle$: U_i receives the public key pk_j of U_j from S. Sets $k_{i\to j}$ to $E'(pk_j, K_i)$ and sends it to S. U_j computes and sends $k_{j\to i}$ to S who then creates edge e_{ij} storing $k_{i\to j}$, $k_{j\to i}$, and adds it to $\mathcal{E}$ of $\mathcal{G}$.
3. $\mathsf{Upload}\langle U_i(K_i, x_i), S(\mathcal{G})\rangle$: U_i chooses nonce r_i, computes value $X^{x_i[l]}_{il}$ as $F_{K_i}(x_i[l], l, r_i)$ $\forall$ $l \in [\ell]$, and sends them to S who stores the value $c_i = ((X^{x_i[1]}_{i1}, \ldots, X^{x_i[\ell]}_{i\ell}), r_i)$ in v_i.
4. $\mathsf{Query}\langle U_q(K_q, \alpha), S(\mathcal{G})\rangle(f)$: U_q does the following:
   (a) **Key and nonce retrieval.** For each $U_j \in \mathcal{G}_q$, retrieve key $k_{j\to q}$ and (latest) nonce r_j from S, and decrypt $k_{j\to q}$ to get K_j.
   (b) **Garbled circuit computation.** U_q transforms f into a circuit, and garbles it as GC.
   (c) **Selection table generation.** For each user U_j in $\mathcal{G}_q$ and index $l \in [\ell]$:
   Compute selection keys: Generate $s^0_{jl} = F_{K_j}(0, l, r_j)$, $s^1_{jl} = F_{K_j}(1, l, r_j)$.
   Compute garbled inputs: Produce encryptions $E_{s^0_{jl}}(w^0_{jl})$ and $E_{s^1_{jl}}(w^1_{jl})$ with the selection keys.
   Set selection table entry: Store $E_{s^0_{jl}}(w^0_{jl})$ and $E_{s^1_{jl}}(w^1_{jl})$ into $T_q[j, l]$ in a random order.
   (d) **Circuit transmission.** Send GC, T_q to S.
   S then decrypts the garbled values of each $U_j \in \mathcal{G}_q$ from T_q, with the encoding $X^{x_j[l]}_{jl}$ for each $l \in [\ell]$. He evaluates GC and sends output to U_q who Obtains the result y by decoding the circuit output.

[1] Due to space limitations, we include all proofs in the full version of the paper [8].

Theorem 1. *If F is a PRF, (E, D) is a symmetric-key CPA-secure encryption scheme with efficiently verifiable range, (E', D') is a public-key CPA-secure encryption scheme, the garbling scheme satisfies privacy and obliviousness, and assuming secure channels between S and the users, protocol π_{GP} securely realizes $\mathcal{F}_{\mathsf{OSN}}$ as per Definition 1.*

5 Mixed Protocol

Sharing the motivation of mixed protocols we explore an alternative construction for evaluating a function f in the OSN model, which combines garbled circuits with additive homomorphic encryption. Recall from Sect. 3.2 that our general approach for designing private constructions for the OSN model entails only two-party interactions. Let $\mathcal{F}_f$ denote the functionality that evaluates f on input homomorphically encrypted values (i.e., the function which the querier wishes to apply to the server stored data). In this work we define the function f to operate over *additively homomorphic* ciphertexts when also given as input the decryption key (formally defined in the full version [8]). Let π_f be a mixed protocol that securely realizes $\mathcal{F}_f$ as discussed in Sect. 2, executed by the server S and the querier U_q. Assume that S possesses the values of U_q and her friends, homomorphically encrypted under the U_q's key. These constitute the input to π_f. In this case, S and U_q can securely evaluate f upon Query by executing π_f. The challenge lies in bringing the inputs of U_q's friends into homomorphic encryptions under U_q's key, without necessitating friend participation in Query. A naive solution would be to have every user send her input to S during Upload, encrypted under all of her friends' keys. This would allow the server to readily have all inputs in the right form upon U_q's Query, but it would also violate our performance requirement for Upload, since the cost would be linear in the number of friends.

In our proposed approach, each user uploads only a single encryption of her input (under her own key), rendering the cost of Upload independent of the number of her friends. In addition, during Connect, each friend U_j of the querier U_q provides her with the means (namely through the $k_{j \to q}$ key shown in Fig. 2) to *re-encrypt* U_j's input into a homomorphic ciphertext under the querier's key.

Construction. Throughout this section, we utilize the symbols summarized in Table 1. π_{RE} represents a protocol implementing the *re-encryption functionality* $\mathcal{F}_{RE}$, fully described in Sect. 5.1. The protocol π_f is executed between a server S holding a sequence of encrypted values $(\llbracket x_1 \rrbracket_{pk_q}, \llbracket x_2 \rrbracket_{pk_q}, \ldots)$, and U_q holding pk_q. At the end of the execution, U_q receives $y = f(\alpha, \ldots, x_q, \ldots)$, whereas S receives nothing. Below, we describe our mixed protocol π_{MP}:

1. $\mathsf{Join}\langle U_i(1^\lambda), S(\mathcal{G})\rangle$: On input the security parameter λ, U_i generates a PRF key K_i, and notifies S that she joins the system by sending pk_i. S adds node v_i (initialized with pk_i) to graph $\mathcal{G}$.
2. $\mathsf{Connect}\langle U_i(K_i), U_j(K_j), S(\mathcal{G})\rangle$: Users U_i and U_j, having each other public keys, compute $k_{j \to i} = \llbracket K_j \rrbracket_{pk_i}$, $k_{i \to j} = \llbracket K_i \rrbracket_{pk_j}$ respectively, and send them to S. Then, S creates an edge e_{ij} in $\mathcal{G}$ storing the two values.

3. $\mathsf{Upload}\langle U_i(K_i, x_i), S(\mathcal{G})\rangle$: User U_i picks random nonce r_i, computes $\rho_i = F_{K_i}(r_i)$, and sends $c_i = (x_i + \rho_i, r_i)$ to S, who stores it into $v_i \in \mathcal{G}$.
4. $\mathsf{Query}\langle U_q(K_q, \alpha), S(\mathcal{G})\rangle(f)$: User U_q and S run π_{RE}, where U_q has as input K_q and S has $\mathcal{G}$. Recall that $\mathcal{G}$ contains c_j and $k_{j\to q}$ for every friend U_j of U_q. The server receives as output $[\![x_j]\!]_{pk_q}$, where x_j is the private input of a friend U_j. Subsequently, S and U_q execute π_f, where S uses as input the ciphertexts $[\![x_j]\!]_{pk_q}$, along with $[\![\alpha]\!]_{pk_q}$ which is provided by the querier. At the end of this protocol, U_q learns $y = f(\alpha, x_q, \{x_j \mid \forall j \,:\, U_j \in \mathcal{G}_q\})$.

Theorem 2. *If F is a PRF and the homomorphic public-key encryption scheme is CPA-secure, assuming secure channels between S and the users, and assuming π_{RE} and π_f securely realize functionalities $\mathcal{F}_{\mathsf{RE}}$ and $\mathcal{F}_f$, respectively, protocol π_{MP} securely realizes $\mathcal{F}_{\mathsf{OSN}}$ as per Definition 1.*

5.1 Re-Encryption Protocol

Our re-encryption protocol π_{RE} implements $\mathcal{F}_{\mathsf{RE}}$ which is a two-party functionality executed between the server S and a querier U_q. Let c_j be the ciphertext of input x_j of user U_j (under U_j's key), stored at S. The goal is to switch c_j into a new ciphertext c'_j under U_q's key, without the participation of U_j. Moreover, it is crucial that c'_j is an encryption under an (additive) homomorphic scheme, because this will subsequently be forwarded to the two-party mixed protocol (π_f) that expects homomorphically encrypted inputs. We provide a formal definition of the re-encryption functionality $\mathcal{F}_{\mathsf{RE}}$ in the *semi-honest* setting using the real/ideal paradigm in the full version [8].

A re-encryption protocol, π_{RE}, can be achieved via the well-known notion of *proxy re-encryption* [12,30]. Specifically, U_j can provide S with a *proxy re-encryption key* $k_{j\to q}$ for U_q during $\mathsf{Connect}$. S can then re-encrypt c_j into c'_j using $k_{j\to q}$ in Query, without interacting with either U_j or U_q. Nevertheless, recall that π_{RE} needs the resulting c'_j to be additive homomorphic. Therefore, this approach needs the proxy re-encryption scheme to also be additive homomorphic. One such candidate is the classic ElGamal-like scheme of [6], which is multiplicative homomorphic, but can be turned into additive homomorphic by a simple "exponential ElGamal" trick. The problem of this modified scheme is that it requires a small message domain, since decryption entails a discrete logarithm computation. Even if the x values are indeed small in a variety of applications, all existing mixed protocols frequently inject some large (e.g., 100-bit) randomness ρ into the homomorphically encrypted value x, necessitating afterwards the decryption of (the large) $x + \rho$ instead of x. This renders the scheme inefficient in our context. To the best of our knowledge, the only other proxy re-encryption schemes with additive homomorphic properties are based on lattices [5,35], whose efficiency is rather limited for practical purposes.

Our Construction. Our alternative approach can be efficiently implemented with *any* additive homomorphic scheme and a PRF. The key idea is to engage the server S and the querier U_q in a single-round interaction that does not reveal

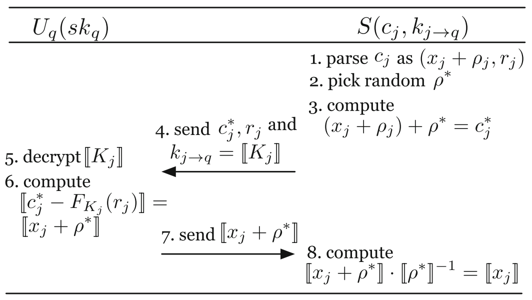
$U_q(sk_q)$
$S(c_j, k_{j\to q})$
1. parse c_j as $(x_j + \rho_j, r_j)$
2. pick random ρ^*
3. compute $(x_j + \rho_j) + \rho^* = c_j^*$
4. send c_j^*, r_j and $k_{j\to q} = [\![K_j]\!]$
5. decrypt $[\![K_j]\!]$
6. compute $[\![c_j^* - F_{K_j}(r_j)]\!] = [\![x_j + \rho^*]\!]$
7. send $[\![x_j + \rho^*]\!]$
8. compute $[\![x_j + \rho^*]\!] \cdot [\![\rho^*]\!]^{-1} = [\![x_j]\!]$

Fig. 4. The re-encryption protocol π_{RE}

anything to U_q. We illustrate our protocol in Fig. 4 for the re-encryption of c_j (produced with U_j's key) to c'_j under U_q's key. S has as input c_j (obtained during U_j's Upload) and $k_{j\to q}$ (obtained during the execution of Connect between U_q and U_j), whereas U_q has key sk_q. In the following, $[\![\cdot]\!]$ denotes a homomorphic ciphertext under U_q's key. S first parses c_j as $(x_j+\rho_j, r_j)$ in Step 1. She then picks a random value ρ^* from an appropriate large domain and computes $c_j^* = x_j+\rho_j+\rho^*$ to statistically hide $x_j+\rho_j$ (Steps 2-3). Subsequently, she sends $c_j^*, r_j, k_{j\to q}$ to U_q (Step 4). The latter decrypts $k_{j\to q}$ using sk_q to retrieve K_j, then computes $c_j^* - F_{K_j}(r_j)$ to remove randomness ρ_j, homomorphically encrypts the result under pk_q and sends it back to S (Steps 5-7). Finally, S computes $[\![\rho^*]\!]^{-1}$ and uses it to remove ρ^* from the received ciphertext. The final output is $c'_j = [\![x_j]\!]$, i.e., U_j's original input encrypted under U_q's key. The above protocol can also be extended to accommodate the simultaneous conversion of *all* ciphertexts c_j such that U_j is a friend of U_q, into homomorphic ciphertexts c'_j under U_q's key.

Lemma 1. *If F is a PRF and the additive homomorphic scheme is CPA-secure, π_{RE} is secure in the presence of static semi-honest adversaries, under the standard secure MPC definition of [24].*

6 Experimental Evaluation

In this section we experimentally evaluate our schemes for two concrete use cases: *(squared) Euclidean* and *Manhattan* distances. These two metrics are used extensively in location-based applications (e.g., where the inputs are geographical coordinates and the query returns the geographically closest friend), and they entail different arithmetic operations (recall that the performance of a garbled circuit or mixed protocol is tightly dependent on the types of operations involved).[2]

[2] For simplicity, we focus on returning the smallest distance, rather than the identity of the closest friend (which can be done easily in garbled circuits and with a standard technique in mixed protocols, e.g., see [7,22]).

Cryptographic Libraries. We used JustGarble [9], a state-of-the-art tool with excellent performance for circuit garbling and evaluation. It supports two important optimizations, *free-XOR* [37] and *row-reduction* [45], which reduce the size of the garbled circuit, and the time to garble and evaluate it. Existing compilers (e.g., [34,41]) for constructing the necessary circuits for our use cases are not directly compatible with JustGarble. Thus, we designed the necessary circuits ourselves, using the basic building blocks that come with JustGarble and employing heuristic optimizations for reducing the number of non-XOR gates.

For our mixed protocols, we used the cryptographic tools described in Sect. 2. We used the Paillier implementation of [1] for the additive homomorphic scheme. For oblivious transfers (OT), we used the code of [52] that implements the OT of [44] with the extension of [29], over an elliptic curve group instantiated with the Miracl C/C++ library [2]. When possible, we used the standard ciphertext-packing method to save communication cost.

Setup. We tested four instantiations: our garbled circuit protocol for the Euclidean and Manhattan case (referred to as GP-Euc and GP-Man, respectively), and their mixed protocol counterparts (referred to as MP-Euc and MP-Man, respectively). All experiments were run on a single 64-bit machine with an Intel®Core$^{\text{TM}}$ i5-2520M CPU at 2.50 GHz and 16 GB RAM, running Linux Ubuntu 14.04. We employed the OpenSSL AES implementation [3] for PRF evaluation and symmetric key encryption at 128-bit level security, leveraging the AES-NI capability [26] of our testbed CPU. For Paillier, we used a 2048-bit group, and for OT a 256-bit elliptic curve group of prime order. Finally, we set the statistically hiding randomness (e.g., ρ in our re-encryption protocol) to 100 bits.

We assess the following costs: size of the garbled circuit in GP-Euc and GP-Man, total communication cost over the channel between two parties, and computational cost at each party. Note that we focus only on Query, since the costs for Join, Upload, and Connect are negligible. We vary the number of friends (10, 100, 1000), the bit-length of each value in the input vector of a user (16, 32, 64), and the number of dimensions (1, 2, 4). Larger numbers of dimensions can capture more general applications entailing Euclidean/Manhattan distance (e.g., user profiles in matchmaking applications). In each experiment, we vary one parameter fixing the other two to their middle values. For computation overhead, we run each experiment 100 times and report average (wall-clock) time.

Circuit Size and Bandwidth Cost. Our first set of experiments evaluates the circuit size (in terms of number of non-XOR gates) in the garbled circuit instantiations, and the communication cost (in MB) in all methods. The results are shown in Fig. 5. First, we vary the number of friends, while fixing the bit size to 32 and the dimensions to 2. The circuit size grows linearly in the number of friends for both distance functions. In the Euclidean case, the circuit is an order of magnitude larger than in Manhattan. This is due to the multiplications Euclidean involves, which require a quadratic number of gates in the number of input element bits. This impacts the communication cost accordingly, since the querier must send a number of garbling values per gate. The overhead of

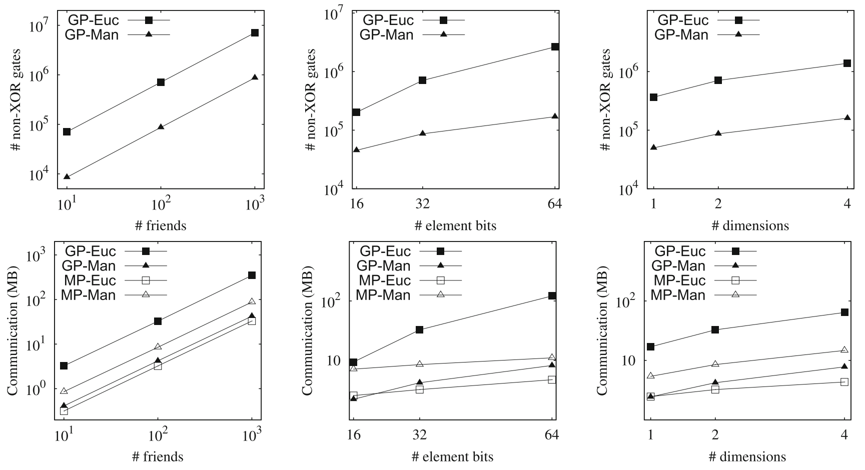

Fig. 5. Circuit size in terms of non-XOR gates (top) and total communication cost in MBs (bottom) vs. number of friends (left), element bit-size (middle), and number of dimensions (right).

MP-Euc is approximately an order of magnitude smaller than that of GP-Euc (e.g., ∼33 MB vs. ∼346 MB for 1000 friends). For the case of Manhattan, the corresponding gap is smaller, due to its substantially smaller circuit size. Note that the communication cost in MP-Man is larger than that of MP-Euc. This is because, recall, MP-Man involves two comparison stages; one during distance computation (due to the absolute values) and one for the final comparison phase.

Then, we show the same two costs for variable bit sizes, setting the number of friends to 100 and dimensions to 2. The circuit size for the Euclidean case grows more steeply with the number of bits; when the bit size doubles, the number of gates almost quadruples. This is expected due to the quadratic (in the bit size) complexity of multiplication. This is not true for the case of Manhattan, where the size roughly doubles when doubling the bit size. The circuit size trend carries over in the communication cost for the garbled circuit approaches. For the mixed protocols, the communication cost grows linearly, but less severely than when varying the number of friends. The reason is that the main cost in these schemes stems mostly from transmitting the necessary garbled circuits the size of which is dominated by the statistical randomness that is fixed to 100 bits (and thus is independent of the variable parameter).

Finally, we plot circuit size and communication overhead as a function of the number of dimensions, for 100 friends and 32-bit inputs. There is a linear dependence between the number of dimensions and the required gates and, thus, both metrics grow linearly for the case of garbled circuits. The same is true for MP-Man, since it entails one absolute value computation per dimension. In the case of MP-Euc there is one multiplication component per dimension and, hence, the communication cost scales linearly as well. However, contrary to MP-Man, MP-Euc involves a comparison protocol only in the final stage: as we explained above, this component receives inputs with a fixed 100-bit length, independently of dimensions. Since this component introduces the dominant communication cost, the total overhead is marginally affected by the number of dimensions.

Computational Cost. The second set of our experiments assesses the computational cost at the querier and the server upon Query, and the results are illustrated in Fig. 6. A first observation is that the computational cost in the garbled circuit approaches is extremely small due to our selection table technique that entirely eliminates the need for oblivious transfers, and the very efficient implementation of JustGarble. Our mixed protocols feature a higher overhead (at both client and server) than their counterparts, because they entail expensive public-key operations (mainly for homomorphic encryptions and decryptions, but also for the base OTs). Still, the computational times for our mixed protocol constructions are not prohibitive even for our largest tested parameters. In most cases the overhead for both querier and server is below 3 s, whereas even for 1000 friends it is below 14 s. A general observation regarding the garbled circuit approaches is that, for all varied parameters, the cost at the server is significantly smaller than that at the client. This is due to the fact that the server performs only symmetric key operations (for extracting the garbled inputs from the selection table and evaluating the garbled circuit), whereas the client also has to decrypt the keys established with her friends during the connection phase, using public-key operations. Finally, regarding the individual curves in the plots, note that they follow similar trends to the corresponding ones in Fig. 5, for the same reasons we explained for the communication cost.

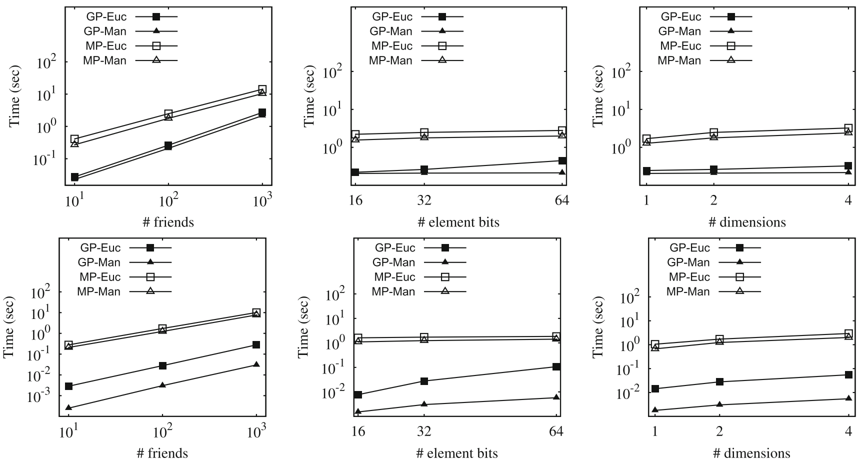

Fig. 6. Total computational cost in seconds at querier (top) and server (bottom) vs. number of friends (left), element bit-size (middle), and number of dimensions (right).

Summary and Discussion. Overall, our GC implementations feature excellent computational times for our tested settings, in the order of a few milliseconds for most scenarios. However, they incur an excessive communication cost for the Euclidean distance (more than 300 MBs for the case of 1000 friends). Our MP implementation is very beneficial for this case, reducing the communication cost by roughly 10x. On the other hand, our MP incur higher computational times than GC, as they entail numerous public key operations to manipulate the Paillier ciphertexts; yet they still offer reasonable performance. Overall, our schemes offer different computation/communication trade-offs in the OSN setting and, interestingly, the overall performance is comparable to existing works that use the same tools in the standard secure two-party computation setting. It is beyond the scope of this paper to advocate one approach over the other. Their performance is highly dependent on the query function and the capabilities of a given system and is a hot research topic in the secure computation literature (e.g., see [34,48]). Moreover, ongoing research can help optimize both alternatives, e.g., the half-gate optimization of [51] reduces the garbled circuit size, whereas [21] shows how faster mixed protocols are achieved using arithmetic shares.

Acknowledgements. We would like to thank Payman Mohassel and Arash Afshar for sharing parts of their code from [4], and the anonymous reviewers for their detailed comments and suggestions. Work partially done while the first and second authors were at Boston University and the fourth author was at Boston University and Northeastern University. Research supported in part by the U.S. National Science Foundation under CNS grants 1012798, 1012910, 1347350, 1413964, and 1414119.

References

1. CPABE (Ciphertext-Policy Attribute-Based Encryption) toolkit. http://acsc.cs.utexas.edu/cpabe/
2. MIRACL cryptographic SDK. https://www.certivox.com/miracl
3. OpenSSL cryptography and SSL/TLS toolkit. https://www.openssl.org/
4. Afshar, A., Mohassel, P., Pinkas, B., Riva, B.: Non-interactive secure computation based on cut-and-choose. In: Nguyen, P.Q., Oswald, E. (eds.) EUROCRYPT 2014. LNCS, vol. 8441, pp. 387–404. Springer, Heidelberg (2014). doi:10.1007/978-3-642-55220-5_22
5. Aono, Y., Boyen, X., Phong, L.T., Wang, L.: Key-private proxy re-encryption under LWE. In: Paul, G., Vaudenay, S. (eds.) INDOCRYPT 2013. LNCS, vol. 8250, pp. 1–18. Springer, Cham (2013). doi:10.1007/978-3-319-03515-4_1
6. Ateniese, G., Fu, K., Green, M., Hohenberger, S.: Improved proxy re-encryption schemes with applications to secure distributed storage. ACM TISSEC **9**(1), 1–30 (2006)
7. Baldimtsi, F., Ohrimenko, O.: Sorting and searching behind the curtain. In: Böhme, R., Okamoto, T. (eds.) FC 2015. LNCS, vol. 8975, pp. 127–146. Springer, Heidelberg (2015). doi:10.1007/978-3-662-47854-7_8
8. Baldimtsi, F., Papadopoulos, D., Papadopoulos, S., Scafuro, A., Triandopoulos, N.: Secure computation in online social networks. Cryptology ePrint Archive, Report 2016/948 (2016)

9. Bellare, M., Hoang, V.T., Keelveedhi, S., Rogaway, P.: Efficient garbling from a fixed-key blockcipher. In: IEEE SP (2013)
10. Ben-Efraim, A., Lindell, Y., Omri, E.: Optimizing semi-honest secure multiparty computation for the Internet. In: ACM CCS (2016)
11. Ben-Or, M., Goldwasser, S., Wigderson, A.: Completeness theorems for non-cryptographic fault-tolerant distributed computation. In: STOC (1988)
12. Blaze, M., Bleumer, G., Strauss, M.: Divertible protocols and atomic proxy cryptography. In: Nyberg, K. (ed.) EUROCRYPT 1998. LNCS, vol. 1403, pp. 127–144. Springer, Heidelberg (1998). doi:10.1007/BFb0054122
13. Bogdanov, D., Laud, P., Randmets, J.: Domain-polymorphic language for privacy-preserving applications. In: CCS-PETShop (2013)
14. Bogetoft, P., Christensen, D.L., Damgård, I., Geisler, M., Jakobsen, T., Krøigaard, M., Nielsen, J.D., Nielsen, J.B., Nielsen, K., Pagter, J., Schwartzbach, M., Toft, T.: Secure multiparty computation goes live. In: FC (2009)
15. Bost, R., Popa, R.A., Tu, S., Goldwasser, S.: Machine learning classification over encrypted data. In: NDSS (2015)
16. Canetti, R.: Security and composition of multiparty cryptographic protocols. J. Cryptol. **13**(1), 143–202 (2000)
17. Carmer, B., Rosulek, M.: Linicrypt: a model for practical cryptography. In: Robshaw, M., Katz, J. (eds.) CRYPTO 2016. LNCS, vol. 9816, pp. 416–445. Springer, Heidelberg (2016). doi:10.1007/978-3-662-53015-3_15
18. Carter, H., Mood, B., Traynor, P., Butler, K.R.B.: Secure outsourced garbled circuit evaluation for mobile devices. In: USENIX Security (2013)
19. Choi, S.G., Katz, J., Kumaresan, R., Cid, C.: Multi-client non-interactive verifiable computation. In: Sahai, A. (ed.) TCC 2013. LNCS, vol. 7785, pp. 499–518. Springer, Heidelberg (2013). doi:10.1007/978-3-642-36594-2_28
20. Damgård, I., Ishai, Y.: Constant-round multiparty computation using a black-box pseudorandom generator. In: Shoup, V. (ed.) CRYPTO 2005. LNCS, vol. 3621, pp. 378–394. Springer, Heidelberg (2005). doi:10.1007/11535218_23
21. Demmler, D., Schneider, T., Zohner, M.: ABY - a framework for efficient mixed-protocol secure two-party computation. In: NDSS (2015)
22. Erkin, Z., Franz, M., Guajardo, J., Katzenbeisser, S., Lagendijk, I., Toft, T.: Privacy-preserving face recognition. In: Goldberg, I., Atallah, M.J. (eds.) PETS 2009. LNCS, vol. 5672, pp. 235–253. Springer, Heidelberg (2009). doi:10.1007/978-3-642-03168-7_14
23. Feige, U., Kilian, J., Naor, M.: A minimal model for secure computation (extended abstract). In: STOC (1994)
24. Goldreich, O.: Foundations of Cryptography: Basic Applications, vol. 2. Cambridge University Press, New York (2004)
25. Goldreich, O., Micali, S., Wigderson, A.: How to play ANY mental game. In: STOC (1987)
26. Gueron, S.: Intel advanced encryption standard AES instruction set white paper. Intel Corporation, August 2008
27. Halevi, S., Lindell, Y., Pinkas, B.: Secure computation on the web: computing without simultaneous interaction. In: Rogaway, P. (ed.) CRYPTO 2011. LNCS, vol. 6841, pp. 132–150. Springer, Heidelberg (2011). doi:10.1007/978-3-642-22792-9_8
28. Henecka, W., Kögl, S., Sadeghi, A.-R., Schneider, T., Wehrenberg, I.: TASTY: tool for automating secure two-party computations. In: CCS (2010)
29. Ishai, Y., Kilian, J., Nissim, K., Petrank, E.: Extending oblivious transfers efficiently. In: Boneh, D. (ed.) CRYPTO 2003. LNCS, vol. 2729, pp. 145–161. Springer, Heidelberg (2003). doi:10.1007/978-3-540-45146-4_9

30. Ivan, A., Dodis, Y.: Proxy cryptography revisited. In: NDSS (2003)
31. Jakobsen, T.P., Nielsen, J.B., Orlandi, C.: A framework for outsourcing of secure computation. In: CCSW (2014)
32. Kamara, S., Mohassel, P., Raykova, M.: Outsourcing multi-party computation. Cryptology ePrint Archive, Report 2011/272 (2011)
33. Kamara, S., Mohassel, P., Riva, B.: Salus: a system for server-aided secure function evaluation. In: CCS (2012)
34. Kerschbaum, F., Schneider, T., Schröpfer, A.: Automatic protocol selection in secure two-party computations. In: Boureanu, I., Owesarski, P., Vaudenay, S. (eds.) ACNS 2014. LNCS, vol. 8479, pp. 566–584. Springer, Cham (2014). doi:10.1007/978-3-319-07536-5_33
35. Kirshanova, E.: Proxy re-encryption from lattices. In: Krawczyk, H. (ed.) PKC 2014. LNCS, vol. 8383, pp. 77–94. Springer, Heidelberg (2014). doi:10.1007/978-3-642-54631-0_5
36. Kolesnikov, V., Mohassel, P., Rosulek, M.: FleXOR: flexible garbling for XOR gates that beats free-XOR. In: Garay, J.A., Gennaro, R. (eds.) CRYPTO 2014. LNCS, vol. 8617, pp. 440–457. Springer, Heidelberg (2014). doi:10.1007/978-3-662-44381-1_25
37. Kolesnikov, V., Schneider, T.: Improved garbled circuit: free XOR gates and applications. In: ICALP (2008)
38. Kreuter, B., Shelat, A., Shen, C.: Billion-gate secure computation with malicious adversaries. In: USENIX Security (2012)
39. López-Alt, A., Tromer, E., Vaikuntanathan, V.: On-the-fly multiparty computation on the cloud via multikey fully homomorphic encryption. In: STOC (2012)
40. Lu, S., Ostrovsky, R.: How to garble RAM programs. In: EUROCRYPT (2013)
41. Malkhi, D., Nisan, N., Pinkas, B., Sella, Y.: Fairplay: a secure two-party computation system. In: USENIX Security (2004)
42. Mohassel, P., Rosulek, M., Zhang, Y.: Fast and secure three-party computation: the garbled circuit approach. In: ACM CCS (2015)
43. Mood, B., Gupta, D., Butler, K.R.B., Feigenbaum, J.: Reuse it or lose it: more efficient secure computation through reuse of encrypted values. In: CCS (2014)
44. Naor, M., Pinkas, B.: Efficient oblivious transfer protocols. In: SODA (2001)
45. Naor, M., Pinkas, B., Sumner, R.: Privacy preserving auctions and mechanism design. In: EC (1999)
46. Nikolaenko, V., Weinsberg, U., Ioannidis, S., Joye, M., Boneh, D., Taft, N.: Privacy-preserving ridge regression on hundreds of millions of records. In: IEEE SP (2013)
47. Paillier, P.: Public-key cryptosystems based on composite degree residuosity classes. In: Stern, J. (ed.) EUROCRYPT 1999. LNCS, vol. 1592, pp. 223–238. Springer, Heidelberg (1999). doi:10.1007/3-540-48910-X_16
48. Schneider, T., Zohner, M.: GMW vs. yao? efficient secure two-party computation with low depth circuits. In: Sadeghi, A.-R. (ed.) FC 2013. LNCS, vol. 7859, pp. 275–292. Springer, Heidelberg (2013). doi:10.1007/978-3-642-39884-1_23
49. Yao, A.C.: How to generate and exchange secrets. In: FOCS (1986)
50. Yao, A.C.: Protocols for secure computations. In: FOCS (1982)
51. Zahur, S., Rosulek, M., Evans, D.: Two halves make a whole. In: Oswald, E., Fischlin, M. (eds.) EUROCRYPT 2015. LNCS, vol. 9057, pp. 220–250. Springer, Heidelberg (2015). doi:10.1007/978-3-662-46803-6_8
52. Zohner, M.: OTExtension library. https://github.com/encryptogroup/OTExtension