Asynchronous provably-secure hidden services

Philippe Camacho and Fernando Krell
{philippe.camacho,fernando.krell}@dreamlab.net

December 20, 2017

Abstract

The client-server architecture is one of the most widely used in the Internet for its simplicity and flexibility. In practice the server is assigned a public address so that its services can be consumed. This makes the server vulnerable to a number of attacks such as Distributed Denial of Service (DDoS), censorship from authoritarian governments or exploitation of software vulnerabilities.

In this work we propose an asynchronous protocol for allowing a client to issue requests to a server without revealing any information about the location of the server. In addition, our solution reveals limited information about the network topology, leaking only the distance from the client to the corrupted participants.

We also provide a simulation-based security definition capturing the requirement described above. Our protocol is secure in the semi-honest model against any number of colluding participants, and has linear communication complexity.

Finally, we extend our solution to handle active adversaries. We show that malicious participants can only trigger a premature termination of the protocol, in which case they are identified. For this solution the communication complexity becomes quadratic.

To the best of our knowledge our solution is the first asynchronous protocol that provides strong security guarantees.

1 Introduction

1.1 Motivation

The client-server architecture is one of the most widely used in the Internet for its simplicity and flexibility. In practice the server is assigned a domain name and one or more IP addresses so that its services can be consumed. This makes the server vulnerable to a number of attacks such as DDoS, censorship from authoritarian governments or exploitation of software vulnerabilities. Thus, it would be desirable to hide the location of the server in the network. By doing so, an attacker will not be able to attack directly the host containing the server's code nor interrupt the execution of its services by non-technical means. While the literature is abundant on the topic of anonymous channels [7, 6, 22, 23], the problem of hiding the location of a server remains of great interest. Tor hidden services [8] is without a doubt the most popular alternative for this purpose. Unfortunately, the security provided by Tor is not guaranteed; in fact, several practical attacks have been discovered [20, 17, 26, 31].

We observe that simple solutions for the problem described above do not work. Standard end-to-end encryption is vulnerable to tracing the ciphertext across the network, and hence, an adversary that is powerful enough to corrupt several nodes is very likely to detect the origin or destination of the message. Other approaches like using multicast are not enough either since clients that are close to the server will notice that the response comes back within short time. The main challenge is to prevent nodes to distinguish whether the server is close or far away.

In this work we focus on solving the following problem. A client wishes to establish a communication with a server, yet we want to hide the location of this server in the network. We also expect the client's queries and server's responses to remain private.

At a high level our protocol implements two phases: (1) a client issues a *request* to the server, and then (2) the server returns a *response*. The first phase of the protocol is straightforward to implement: the client encrypts the request using the public key of the server and then multicasts the message across the network. Note that the server must still forward the request as if it were any other node, otherwise its neighbors may infer its location. The second phase is much more complex because as mentioned above the client or other nodes could detect the presence of the server by a simple timing attack. To circumvent this difficulty we rely on the following idea: we force all the nodes to behave as the server. We achieve this by using a secret sharing scheme where every participant holds a share of the response. To perform this split-and-reconstruct phase, every node (including the server) generates a random share, and then all these are propagated to the server. At this stage the server replaces its share by a value that enables to reconstruct the response. Finally all the participants send their shares to the client.

In order to improve performance, we use an arbitrary spanning tree[1] over the network graph. This allows us to optimize multicast invocations and shares aggregation. We emphasize that our protocol is *asynchronous*, which means that participants *do not rely a on shared clock* to run the protocol, but rather act upon the reception of neighbors messages. Unfortunately, asynchronism comes at price: Since nodes do not know when a participant initiates a request, it is impossible to hide the requesters activity. Hence our protocol leaks proximity information of the requester to other nodes.

1.2 Contributions

Our contributions are the following:

[1] which we borrow from Dolev and Ostrovsky [9].

- To the best of our knowledge we provide the first simulation-based security definition capturing the requirement of hiding a server in a network. This definition considers the full interaction (request and response) between the client and the server.
- We provide a protocol (and implementation alternatives) for the hidden server problem in the semi-honest adversarial mode.
- Our protocol is secure against any number of corrupted participants. In particular, if the adversary controls all nodes but two (one of them being the server), then it will not be able to guess the right location with probability better than $\frac{1}{2}$.
- Our solution has linear communication complexity. Although, this is may not be practical in large environments, it is *asymptotically optimal*: a sublinear protocol would leak the fact that silent nodes cannot be the server.
- Finally, we extend our solution to handle active adversaries. We show that malicious participants can only trigger a premature termination of the protocol, in which case they are identified. For this solution the communication complexity becomes quadratic in the number of participants.
- To the best of our knowledge the proposed protocols are the first to provide strong security guarantees in an asynchronous setting (see Figure 1).

1.3 Related Work

While the problem of hiding the physical location of a server in a network is not exactly an anonymity problem (we do not want to hide the fact that a specific client connects to the server) the techniques and concepts we use are borrowed from the area of anonymity. Since Chaum's two seminal papers on mixes [7, 6], a large body of work has been written in order to enable communications that do not reveal the identity of participants. An alternative to mixers for achieving anonymity has been introduced by Reiter *et al.* with a protocol named Crowds[24] and consists of using random paths among a set of "dummy" nodes a.k.a. *jondo* before reaching a specific destination (the server). In this protocol – contrary to our setting – the location of the server is public and the goal is to hide the clients. This solution is simple, efficient and provide some level of anonymity for the client. Beyond the protocol itself, the authors highlight some fundamental problems that arise with these types of constructions where traffic is routed through possible corrupted nodes: In particular, preserving the initiator's anonymity turns out to be more complex than expected [30, 27]. Indeed in our case, we have to solve a similar problem where we must hide the location of the server during the phase of responding a request. Hordes [18] is an improvement to Crowds where the reply from the server is done using multicast. This change makes passive attacks consisting in tracing back messages harder while adding only a reasonable operational cost. While Crowds and Hordes do not aim to hide the server like we do, these protocols highlight the difficulty of hiding nodes in a network where the adversary controls a subset of the participants and can leverage traffic analysis. Another approach to establish anonymous channels between client and servers is onion routing [13]. An onion is obtained by encrypting the message in a layered fashion using the public keys of the nodes on a path from sender to receiver. By doing so, a node on the circuit will not be able to identify the original source, the final destination, nor the message itself. The most popular onion routing protocol is without a doubt Tor [8]. Tor not only enables to preserve the anonymity of clients but also provides a mechanism to hide the location of the server through a *rendez-vous* node where both client and server meet. Unfortunately, as in Crowds and Hordes, a number of practical attacks based on traffic analysis are possible [17, 26, 31, 21]: In particular if a node manages to be the first relay between the server and the *rendez-vous* node, it will likely detect the server presence [21]. In case managing a Public-Key Infrastructure is too complex, one can use Katti *et al.*'s protocol [16] that relies on the idea of splitting the routing information in such a way that only the right nodes on the circuit are able to reconstruct it correctly. In our protocol we also leverage secret-sharing techniques, but for splitting and reconstructing the message only. Also our solution does not require a sender to control different nodes as in the onion slicing approach.

Early attempts to counter traffic analysis attacks were not practical as they assumed the existence of some broadcast channel or ad-hoc topology and required a synchronous execution [6, 23, 29]. The more general problem of hiding the topology of a network has been solved recently in the secure multi-party computation setting [1, 19, 14]. However, these solutions involves a lot of communication and computational overhead. One of the most promising attempts for hiding the location of a server was due to Dolev and Ostrovsky [9]: Indeed our solution borrows some of the techniques of their work, in particular we also use spanning-trees to make the multicast communications more efficient. Nonetheless our solution has two major advantages: it is asynchronous and it is secure against any number of corrupted nodes.

In Figure 1 we compare our work with other proposals that allow arbitrary topologies.

1.4 Organization of the paper

This paper is organized as follows. Section 2 introduces definitions and notations. The abstract functionality capturing the secure interaction between client and server is introduced in Section 3. We describe a protocol secure against semi-honest adversaries in Section 4, and prove its security in Section 5. Then, in Section 6 we present a protocol secure against malicious players in which deviation of the protocol is either harmless or identifiable. Finally, we conclude in Section 7.

Protocol	Asynchronous	Collusion-resistant	Communication complexity
Tor [8]	YES	NO	$O(D \cdot \lvert M \rvert)$
Dolev and Ostrovsky [9]	NO	Up to $\lfloor (N-1)/2 \rfloor$	$O(N \cdot \lvert M \rvert)$
MPC-Hiding topology [1]	NO	YES	$O(\kappa(\kappa + \log N) \cdot N^5 \cdot \lvert M \rvert)$
Our work	YES	YES	$O(N \cdot \lvert M \rvert)$

Figure 1: **Comparison of protocols for hiding a node location.** In this table N is the number of participants, D is the diameter of the graph representing the network, $|M|$ is the number of bits of the message and κ is the security parameter. Tor is not collusion-resistant because some attacks can succeed with only two corrupted nodes [21]. Regarding communication complexity, we do not take into account the setup phase occurring in Dolev and Ostrovsky's construction and ours.

2 Preliminaries

2.1 Definitions and notations

Let $n \in \mathbb{N}$ be an integer, we denote by $[n]$ the set $\{1, 2, 3, \cdots, n\}$. Let B be a set, we write $b \in_R B$ to denote a value b chosen uniformly at random from B.

For a graph $G = \langle V, E \rangle$ the distance $d(u, v)$ between two vertices u and v is the length of the shortest path between u and v. Let $(\mathsf{M}, \circ)$ be an abelian group and $\kappa \in \mathbb{N}$ the security parameter. A (single-operation) homomorphic encryption scheme over message space M is a tuple of algorithms $\mathcal{H} = \langle \mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{Add} \rangle$ in which $\langle \mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec} \rangle$ is a public-key encryption scheme and algorithm Add satisfy the following property: For every key-pair $(\mathsf{pk}, \mathsf{sk}) \leftarrow \mathsf{Gen}(1^\kappa)$, and for every pair of messages $m_1, m_2 \in \mathsf{M}$: $\mathsf{Dec}_{\mathsf{sk}}(\mathsf{Add}_{\mathsf{pk}}(\mathsf{Enc}_{\mathsf{pk}}(m_1), \mathsf{Enc}_{\mathsf{pk}}(m_2))) = m_1 \circ m_2$.

For some arbitrary ciphertext set $C = \{c_i = \mathsf{Enc}_{\mathsf{pk}}(m_i)\}_{i \in I}$, we abuse notation by using $\sum_{i \in I} c_i$ or $\mathsf{Enc}_{\mathsf{pk}}(\sum_{i \in I} m_i)$ to denote the result of a sequential computation of $\mathsf{Add}_{\mathsf{pk}}$ over C.

2.2 Modeling networks

We can think of a regular communication network as a graph G, composed by a set of nodes V and a set of edges E between them. Participants (nodes) v_i and v_j cannot communicate directly unless there is an edge (v_i, v_j) in E. To allow communication between distant participants, nodes can forward incoming messages to neighbor nodes following some protocol.

We use the approach of [14] in which the participants in the real protocol are restricted to use a *network* functionality to communicate. The network functionality is specified in Figure 2, and allows any participant to send messages to a neighbor at an arbitrary time[2]. It provides two services, **Setup** and **Comm**. On the setup phase, the communication graph is specified. This can be done by

[2]The network functionality of [14] is rather different in the sense that all participant call it at same time, and all have message to all its neighbors.

$\mathcal{F}_{\mathsf{network}}$

Participants: On-line participants $\mathcal{P} = \{P_1, P_2, \ldots, P_N\}$, and off-line operator $\mathtt{Op}$

Setup. Operator $\mathtt{Op}$ inputs an undirected graph $G = \langle V, E \rangle$ and a mapping $M : \mathcal{P} \leftrightarrow V$. Each participant P_i gets $v = M(P_i)$, and set of neighbors $\{u : (v, u) \in E\}$.

Comm. On input $(\mathtt{msg}, v_j)$ from participant P_i, P_j outputs $(\mathtt{msg}, u)$ (where u is $M(P_i)$) if $(M(P_i), v_j) \in E$. Otherwise, P_i outputs error symbol $\perp$.

Figure 2: Physical Network Functionality

an off-line operator, or by the participant itself describing their neighbors (or their pseudonyms as inputs). The **Comm** service allows for neighbor participant to exchange messages. We require that **Setup** is called before any **Comm** service can be processed.

We will use this functionality as the basic mechanism to send message throughout the network. Protocols in this model will be called $\mathcal{F}_{\mathsf{network}}$-restricted, meaning that the only way participants can communicate is via $\mathcal{F}_{\mathsf{network}}$.

2.3 Multicast protocol

In this section we describe a simple multicast protocol (see Figure 3) that uses functionality $\mathcal{F}_{\mathsf{network}}$ as its basic communication mechanism. We assume that a trusted party has already instantiated the network functionality, and hence each participant knows the vertex label associated with its neighbor for functionality $\mathcal{F}_{\mathsf{network}}$. When a participant issues a multicast, it sends the message to its neighbor using functionality $\mathcal{F}_{\mathsf{network}}$. Each participant, upon reception of a multicast message, first checks if the message has not been seen before. In this case, it forwards the message to its neighbors and outputs the message. Jumping ahead, our main protocol will use this functionality on a subgraph of the network graph to efficiently broadcast the client's encrypted requests.

2.4 Security definition

As standard in cryptographic protocols, we define security in terms of a real-versus-ideal world procedures. That is, we first specify a desired functionality for our protocol. Then, we say that a protocol computing the functionality is *secure* if its real-world execution realizes an ideal procedure. In this ideal procedure, the participants get their outputs by sending their inputs to a trusted party computing the functionality on behalf of them. More specifically, we say that our protocol *privately computes* the functionality if whatever can be achieved by adversary interacting in the real execution of the protocol, can also be obtained with only inputs and outputs of the corrupted participants in an ideal execution.

Protocol $\Pi_{\mathsf{MCast}}^{\mathcal{F}_{\mathsf{network}}}$

Participants. On-line participants $\mathcal{P} = \{P_1, P_2, \ldots, P_N\}$, and an off-line trusted party T.

Requirement. Off-line trusted party T has initialized $\mathcal{F}_{\mathsf{network}}$ functionality for $\mathcal{P}$ on a graph $G = \langle V, E \rangle$.

Setup. On input a graph $G' = \langle V, E' \subseteq E \rangle$, T specify to each participant the label of its neighbors in G'. Each participant initiates an empty set L.

MCast. Any participant P: On input a message m, invoke $\mathcal{F}_{\mathsf{network}}.\mathbf{Comm}(\langle \mathtt{mcast}, m \rangle, u)$ for each neighbor u in G'.

Upon receiving $\mathcal{F}_{\mathsf{network}}.\mathbf{Comm}$'s output $\langle \mathtt{mcast}, m \rangle$ from neighbor v, check if $m \notin L$. If so, add m to set L, invoke $\mathcal{F}_{\mathsf{network}}.\mathbf{Comm}(\langle \mathtt{mcast}, m \rangle, u)$ for each neighbor $u \neq v$ in G', and output m. Otherwise, do nothing.

Figure 3: $\Pi_{\mathsf{MCast}}^{\mathcal{F}_{\mathsf{network}}}$

In this section we provide a security definition for *semi-honest static* adversaries. In what follows we let algorithms Sim, Adv, and $\mathcal{Z}$ be stateful.

$\mathsf{Ideal}_{\mathcal{Z},\mathsf{Sim}}^{\mathcal{F}}(\kappa)$: 1) Run $\mathcal{Z}(1^\kappa)$ to produce participant inputs $\{\mathsf{in}_j\}_{j\in[N]}$ and adversary input $\mathsf{in}_{\mathsf{Sim}}$. 2) Run $\mathsf{Sim}(1^\kappa, \mathsf{in}_{\mathsf{Sim}})$ to get the index set of corrupted parties $I_{\mathcal{C}} \subseteq [N]$. 3) Run $\mathsf{Sim}(\{\mathsf{in}_k\}_{k\in I_{\mathcal{C}}})$ to obtain modified input $\{\mathsf{in}'_k\}_{k\in I_{\mathcal{C}}}$ for the corrupted parties. 4) Call functionality $\mathcal{F}$ on previous inputs to obtain output $\{\mathsf{out}_j\}_{j\in[N]}$. 5) Run $\mathsf{Sim}(\{\mathsf{out}_k\}_{k\in I_{\mathcal{C}}})$ to get adversary's output $\mathsf{out}_{\mathsf{Sim}}$. 6) Run $\mathcal{Z}(\{\mathsf{out}_j\}_{j\in[N]\setminus I_{\mathcal{C}}}, \mathsf{out}_{\mathsf{Sim}})$ to obtain output bit b. 7) Return b as the output of the ideal-world execution.

$\mathsf{Real}_{\mathcal{Z},\mathsf{Adv}}^{\Pi}(\kappa)$: 1) Run $\mathcal{Z}(1^\kappa)$ to produce participant inputs $\{\mathsf{in}_j\}_{j\in[N]}$ and adversary input $\mathsf{in}_{\mathsf{Adv}}$. 2) Run $\mathsf{Adv}(1^\kappa, \mathsf{in}_{\mathsf{Adv}})$ to get set of corrupted parties $I_{\mathcal{C}} \subseteq [N]$. 3) Run $\mathsf{Adv}(\{\mathsf{in}_k\}_{k\in I_{\mathcal{C}}})$ to obtain modified input $\{\mathsf{in}'_k\}_{k\in I_{\mathcal{C}}}$ for the corrupted parties. 4) Execute protocol Π with previously computed inputs, saving the view of every corrupted participant, $\{\mathsf{view}_k\}_{k\in I_{\mathcal{C}}}$. When every participant finishes the protocol execution, recollect output of every uncorrupted participants, $\{\mathsf{out}_j\}_{j\in[N]\setminus I_{\mathcal{C}}}$. 5) Run $\mathsf{Adv}(\{\mathsf{view}_k\}_{k\in I_{\mathcal{C}}})$ to get adversary's output $\mathsf{out}_{\mathsf{Adv}}$. 6) Run $\mathcal{Z}(\{\mathsf{out}_j\}_{j\in[N]\setminus I_{\mathcal{C}}}, \mathsf{out}_{\mathsf{Adv}})$ to obtain output bit b. 7) Return b as the output of the real-world execution.

Definition 1. *A protocol* Π privately computes *functionality* $\mathcal{F}$ *if for every PPT algorithm* Adv*, there exists a PPT algorithm* Sim *such that for every PPT algorithm* $\mathcal{Z}$ *the random variables* $\mathsf{Ideal}_{\mathcal{Z},\mathsf{Sim}}^{\mathcal{F}}(1^\kappa)$ *and* $\mathsf{Real}_{\mathcal{Z},\mathsf{Adv}}^{\Pi}(1^\kappa)$ *are computationally indistinguishable, for all sufficiently long* κ*.*

In our work it is sufficient to show a PPT simulator Sim that can produce a

$\mathcal{F}^{\mathcal{L}(\cdot)}_{\mathsf{ReqResp}}$

Participants. On-line participants $\mathcal{P} = \{P_1, P_2, \dots, P_N\}$, and an off-line operator `Op` that instantiate the parameters on a setup phase.

Parameters. A graph $G = \langle V, E \rangle$.

Setup. On input $j^* \in [|V|]$, a mapping $M : \mathcal{P} \leftrightarrow V$, and a polynomial-time computable function `ProcessReq`, store `ProcessReq` and j^*.
Output v_j to participant P_j for every $j \neq j^*$, and, $\langle v_{j^*}, \texttt{ProcessReq} \rangle$ to participant P_{j^*}.

Req. On input `req` from participant P_i do:

- $\texttt{res} \leftarrow \texttt{ProcessReq}(\texttt{req})$.
- Output `res` to P_i and `req` to P_{j^*}.
- Leak $\mathcal{L}(\mathbf{Req}, |\texttt{req}|, |\texttt{res}|, G, M, P_i)$ to the adversary.

Figure 4: Hidden-Server Request-Response functionality $\mathcal{F}_{\mathsf{ReqResp}}$ over an incomplete network with leakage profile $\mathcal{L}(\cdot)$.

view that is computationally indistinguishable from the corrupted participants view. Then, the simulator can run $\mathcal{A}$ to produce a simulated output to $\mathcal{Z}$.

We slightly modify the ideal world to include a leakage function, $\mathcal{L}$, whose output is leaked to the simulator Sim. This leakage function models the fact the protocol may reveal some partial private information to the adversary (for example, the length of the messages to encrypt). It also allows for the specification of trade-offs between protocol features or efficiency and security. This leakage information is added to the simulator's input on step 5.

3 Request Response Functionality

The functionality is executed between a set of participant $\mathcal{P} = \{P_1, P_2, P_3, \dots\}$. A server node, which we denote as $\mathcal{S}$, provides an arbitrary polynomial-time request-response service for all participants. A protocol realizing this functionality needs to hide which of the participant is the server node. A secondary goal is to hide the requests and the responses.

In Figure 4, the functionality is parametrized by a public graph G. During a setup phase, the operator participant `Op` specifies the server node, its service Turing machine `ProcessReq`, and a mapping M between graph nodes and participants. As a result of this setup phase, every node gets its graph label, and the server node gets the Turing machine `ProcessReq`.

4 A protocol secure against semi-honest adversaries

4.1 Overview

For a set of participants $\mathcal{P} = \{P_1, P_2, \ldots, P_N\}$ communicating over an arbitrary network graph G, the goal of our protocol is to hide the location of a server $\mathcal{S} = P_{j^*}$ in G while enabling other participants to consume its services. The main difficulty is to make it impossible for an adversary to leverage timing information to obtain (or estimate) the distance between $\mathcal{S}$ and some other corrupted nodes in G.

The protocol proceeds in two high level steps. The first step corresponds to enabling a client P_i to send a request `req` to the server $\mathcal{S}$. This step can be easily implemented using a multicast protocol (see Section 2.3): The client encrypts `req` using $\mathcal{S}$'s public key and multicasts the ciphertext $c = \mathsf{Enc}_{\mathsf{pk}_{\mathcal{S}}}(\mathtt{req})$. Indeed, $\mathcal{S}$'s location is not leaked[3].

The second step consists of letting the server $\mathcal{S}$ to send the response `res` back to P_i. This turns out to be more challenging. Indeed, proceeding as in the first step is not secure since nodes that are close to $\mathcal{S}$ would detect $\mathcal{S}$'s activity and be able to deduce its location or some information about it (as for example the subnet that contains $\mathcal{S}$). In order to circumvent this difficulty we introduce the following high level idea: each node P_j sends a random share s_j to the server $\mathcal{S}$ (including the $\mathcal{S}$ itself). The server will obtain all the shares $\{\mathsf{share}_j\}_{j \neq i}$ and recompute its share share_{j^*} so that combination of all shares reconstruct to `res`. Then, all the participants send their shares to the requester P_i, and finally, P_i reconstructs and outputs the response.

Since shares on the last step reconstruct the response, it is clear that they need to be encrypted under P_i's public-key. As the initial shares sent to the server reconstruct to a random value, it is tempting to send these in plaintext. However, an adversary that controls the requester can see the shares both times, and therefore notice when a share was updated, inferring information on $\mathcal{S}$'s location.

We take the approach of [9] and restrict the communication to an (arbitrary) spanning-tree on the network graph. This allows us to efficiently communicate the messages on all phases. In particular, we use the following mechanism to send the shares to $\mathcal{S}$ and P_i: First, the shares are sent up to the root node of the spanning tree, and then the root node multicasts the shares down the tree. By using n-out-of-n information-theoretic secret sharing, we note that nor the server or the requester need to know every individual share. In fact, they only need to learn the final secret. Our idea, hence, is to use homomorphic encryption on the shares, and have each internal node to "add-up" its share to the shares computed by its children, and then send a single result up the tree (rather than the individual shares of every node in its subtree). The root

[3] Note that messages needs to be forwarded once – and only once– to neighbors, even when the message has arrived to its destination.

Protocol $\Pi_{\mathsf{ESR}}^{\mathcal{F}_{\mathsf{network}}}$

Participants. On-line participants $\mathcal{P} = \{P_1, P_2, \ldots, P_N\}$, and an off-line trusted party T.

Parameters. An homomorphic encryption scheme $\mathcal{H} = \langle \mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{Add} \rangle$.

Requirement. Off-line trusted party T has initialized $\mathcal{F}_{\mathsf{network}}$ functionality for $\mathcal{P}$ on a graph $G = \langle V, E \rangle$.

Setup. On input a spanning tree $\mathsf{ST} = \langle \mathsf{root} \in V, E_{\mathsf{ST}} \subset E \rangle$ over G, T specifies to every participant its parent p and children set $\mathsf{children}$ on ST.

SendUp. Any participant $P \in \mathcal{P}$: On input a message m, public key pk, and session id sid, compute $c = \mathsf{Enc}_{\mathsf{pk}}(m)$ and store $\langle \mathsf{sid}, c \rangle$.

If P_i has no children jump to $\star$.

Upon receiving $\mathcal{F}_{\mathsf{network}}.\mathbf{Comm}$'s output $\langle \mathsf{sid}, \mathtt{up}, c' \rangle$ from children u, use sid to get c and update it to $\mathsf{Add}_{\mathsf{pk}}(c, c')$. If all children have submitted their $\mathtt{up}$ message, then jump to $\star$.

$\star$: If P is the root of the tree, output (c, sid). Otherwise, invoke $\mathcal{F}_{\mathsf{network}}.\mathbf{Comm}(\langle \mathsf{sid}, \mathtt{up}, c \rangle, p)$.

SendDown. Participant $P \in \mathcal{P}$ (root of the tree): On input a message c and session id sid, invoke $\Pi_{\mathsf{MCast}}^{\mathcal{F}_{\mathsf{network}}}.\mathbf{MCast}(\langle \mathsf{sid}, \mathtt{down}, c \rangle)$, and output (c, sid).

(Any participant). Upon receiving $\Pi_{\mathsf{MCast}}^{\mathcal{F}_{\mathsf{network}}}.\mathbf{MCast}'$s output $\langle \mathsf{sid}, \mathtt{down}, c \rangle$, output (c, sid)

Figure 5: $\Pi_{\mathsf{ESR}}^{\mathcal{F}_{\mathsf{network}}}$

node then obtains an encrypted secret, which is sent down the tree to reach the server or the requester. This efficient procedure allows our protocol to have linear communication complexity, and is formally described in Section 4.2.
Our full protocol implementing functionality $\mathcal{F}_{\mathsf{ReqResp}}$ is specified in Section 4.3.

4.2 Encrypted Share Reconstruction Protocol

In this section we describe an important sub-protocol of our solution. This protocol, denoted Π_{ESR}, allows to efficiently and privately reconstruct a secret out of each participant share. In a nutshell, each party encrypts its share under the public-key of the recipient, and sends the ciphertext up into a spanning tree of the network graph. The participant at the root node of this tree can homomorphically compute the encrypted secret, and then send the result down the tree to reach the recipient. We do this efficiently in the following way: Each internal node privately reconstructs part of the secret by homomorphically combining its encrypted share with the ciphertext obtained from its children. Hence, each internal node needs to send a single ciphertext up the tree. Furthermore, we use n-out-of-n information-theoretic secret sharing so that we only need a single homomorphic operation for the encryption scheme. Protocol Π_{ESR} is specified in Figure 5.

4.3 Request-Response Server Protocol

In this section we introduce an $\mathcal{F}_{\mathsf{network}}$-hybrid protocol achieving functionality $\mathcal{F}_{\mathsf{ReqResp}}$. Our protocol is divided in an off-line setup phase and three on-line phases. In the setup phase, a trusted party T chooses a server participant $\mathcal{S}$

and generates for it a key-pair $(\mathsf{pk}_{\mathcal{S}}, \mathsf{sk}_{\mathcal{S}})$. T also chooses an arbitrary rooted spanning-tree in order to instantiate the protocol Π_{ESR}. On the first on-line phase, the requester P_i encrypts its query $\mathtt{req}$ under the server's public key, and uses protocol Π_{MCast} to propagate the ciphertext across the network. Then, on the second on-line phase every participant (including the server) generates a random string of length outlen (used as a share for the response) and sends it to the server using protocol Π_{ESR}. Upon receiving the combined shares $cs = \sum_{j \neq i} \mathsf{share}_j$, $\mathcal{S}$ recomputes its share share_{j^*} as $\mathtt{res} - (cs - \mathsf{share}_{j^*})$ so that the reconstruction procedure outputs the response $\mathtt{res}$. On the third on-line phase, every participant P_j use Π_{ESR} to send its share_j (encrypted under P_i's public key), so that the response can be homomorphically reconstructed and sent to P_i. P_i decrypts and output the response.

Notice that these three phases can be executed in a pipeline. In fact, each encrypted share sent on the second on-line phase can be sent as soon as the participant sees the request multicast message issued by P_i on the first phase. Similarly, each participant can send its share in the third phase as soon as the participant sees the multicast-down message issued by the root node in the second phase. Therefore, our protocol is *asynchronous*.

We also note that the initial multicast of the encrypted request leaks the direction towards the requester node to each participant. Therefore, the encrypted response on the third phase, can be sent efficiently from the root to the requester. In fact, when a participant receives the request message from neighbor u, this is saved so that at the final phase, each participant knows where to send the encrypted response.

Since all participants act according to the same communication pattern, and all messages are encrypted, our protocol does not reveal the location of the server, nor the request or response. We can observe that every participant send a constant number of messages during the execution of the protocol and thus the communication complexity is equal to $O(N \cdot \max(|\mathtt{req}|, |\mathtt{res}|))$. Our protocol is formally described in Figure 6.

4.4 Variants of the protocol

Avoiding an off-line trusted party. Protocol 6 relies on a trusted party to set up the initial parameters of each participant. By using state-of-the-art topology-hiding secure computation protocols [19, 14, 2, 1] we can achieve a secure distributed setup without any trusted party.

Precomputing shares using PRG. It is possible to simplify the protocol described in Figure 6 by having the server computing the other participant shares locally. In practice, all the participants would receive a secret seed R_j to generate its seed, and the server receives the secret seeds of every participant. This means that the second on-line phase of the protocol can be removed, and hence save $2N$ in communication complexity and N homomorphic operations. The other steps remain unchanged.

Response recipient. Our protocol can be modified so that the recipient of the response can be any arbitrary participant (or set of participants). This is

Protocol $\Pi^{\mathcal{F}_{\mathsf{network}}}_{\mathsf{ReqResp}}$

Participants. On-line $\mathcal{P} = \{P_1, P_2, \ldots, P_N\}$, and an off-line trusted party T.

Parameters. A security parameter κ and an homomorphic encryption scheme $\mathcal{H} = \langle \mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{Add} \rangle$.

Requirement. Off-line trusted party T has initialized $\mathcal{F}_{\mathsf{network}}$ functionality for $\mathcal{P}$ on a graph G.

Setup. a) T chooses a server participant $\mathcal{S} \in \mathcal{P}$ and an arbitrary spanning tree ST on graph G. b) T instantiate protocols $\Pi^{\mathcal{F}_{\mathsf{network}}}_{\mathsf{MCast}}$ and $\Pi^{\mathcal{F}_{\mathsf{network}}}_{\mathsf{ESR}}$ on input ST, c) T generates server's key pair $(\mathsf{pk}_{\mathcal{S}}, \mathsf{sk}_{\mathcal{S}}) \leftarrow \mathsf{Gen}(1^{\kappa})$ and securely distributes $\mathcal{S}$'s public key $\mathsf{pk}_{\mathcal{S}}$ to every participant. d) Finally, T securely sends $(\mathsf{sk}_{\mathcal{S}}, \mathsf{pk}_{\mathcal{S}})$ and a Turing Machine $\texttt{ProcessReq}$ to $\mathcal{S}$.

(In what follows, let j^* denote the index of $\mathcal{S}$ in $\mathcal{P}$.)

Req. On input $\texttt{req} \in \{0,1\}^{\mathsf{inlen}}$, participant P_i chooses a session key-pair $(\mathsf{pk}_{\mathsf{sid}}, \mathsf{sk}_{\mathsf{sid}})$ and a fresh session id sid, and invokes multicast protocol $\Pi^{\mathcal{F}_{\mathsf{network}}}_{\mathsf{MCast}}.\mathbf{MCast}(\langle \texttt{request\_to\_server}, \mathsf{sid}, \mathsf{Enc}_{\mathsf{pk}_{\mathcal{S}}}(\texttt{req}), \mathsf{pk}_{\mathsf{sid}} \rangle)$ over the spanning tree.

Response phase. Every participant P_j (including $\mathcal{S}$):

1. Upon receiving $\Pi^{\mathcal{F}_{\mathsf{network}}}_{\mathsf{MCast}}.\mathbf{MCast}$'s output $\langle \texttt{request\_to\_server}, \mathsf{sid}, C, \mathsf{pk}_{\mathsf{sid}} \rangle$ from neighbor u, pick a random share $\mathsf{share}_j \in \{0,1\}^{\mathsf{outlen}}$ and invoke $\Pi_{\mathsf{ESR}}.\mathbf{SendUp}(\mathsf{share}_j, \mathsf{pk}_{\mathcal{S}}, \mathsf{sid})$, and store $\langle \mathsf{sid}, \mathsf{pk}_{\mathsf{sid}}, \mathsf{share}_j, u \rangle$. In addition, if P_j is $\mathcal{S}$, compute $\texttt{req} = \mathsf{Dec}_{\mathsf{sk}_{\mathcal{S}}}(C)$ and $\texttt{res} \leftarrow \texttt{ProcessReq}(\texttt{req})$, and store $\langle \mathsf{sid}, \texttt{req}, \texttt{res} \rangle$.
2. (Root node) Upon receiving $\Pi_{\mathsf{ESR}}.\mathbf{SendUp}$'s output $(C = \mathsf{Enc}_{\mathsf{pk}_{\mathcal{S}}}(\sum_{j \neq i} \mathsf{share}_j), \mathsf{sid})$, invoke $\Pi_{\mathsf{ESR}}.\mathbf{SendDown}(C, \mathsf{sid})$.
3. Upon receiving $\Pi_{\mathsf{ESR}}.\mathbf{SendDown}$'s output (C, sid), use sid to get $\mathsf{pk}_{\mathsf{sid}}$ and share_j from local storage, and:
   - If P_j is $\mathcal{S} = P_{j*}$, decrypt C to get $\mathsf{share}_{\mathsf{sum}} = \sum_{j \neq i} \mathsf{share}_j$, and update share_{j*} to $\texttt{res} - (\mathsf{share}_{\mathsf{sum}} - \mathsf{share}_{j*})$.
   - Invoke $\Pi_{\mathsf{ESR}}.\mathbf{SendUp}(\mathsf{share}_j, \mathsf{pk}_{\mathsf{sid}}, \mathsf{sid})$
4. (Root node) Upon receiving $\Pi_{\mathsf{ESR}}.\mathbf{SendUp}$'s output $(C = \mathsf{Enc}_{\mathsf{pk}_{\mathsf{sid}}}(\sum_{j \neq i} \mathsf{share}_j), \mathsf{sid})$, use sid to get neighbor label u, and invoke $\mathcal{F}_{\mathsf{network}}.\mathbf{Comm}(\langle C, \mathsf{sid} \rangle, u)$.
5. Upon receiving $\mathcal{F}_{\mathsf{network}}.\mathbf{Comm}$'s output (C, sid) do:
   - If P_j is P_i, use sid to get $\mathsf{sk}_{\mathsf{sid}}$ from local storage, and output $\texttt{res} \leftarrow \mathsf{Dec}_{\mathsf{sk}_{\mathsf{sid}}}(C)$.
   - Otherwise, use sid to get u, and invoke $\mathcal{F}_{\mathsf{network}}.\mathbf{Comm}(C, u)$.

Figure 6: $\Pi^{\mathcal{F}_{\mathsf{network}}}_{\mathsf{ReqResp}}$

achieved as follows: (a) the client chooses the public key of another participant as the session public key, and (b) because the location of the recipient is not necessarily known, the root node multicasts the encrypted response down the tree instead of sending it directly to the originator of the request.

Avoiding the use of the spanning tree. In a practical environment, the spanning tree could affect the resilience of the protocol and can be hard to maintain or configure. In such a scenario, the steps (**SendUp**,**SendDown**) can be replaced by multicast operations of the shares for each participant.

5 Proof of Security

In this Section we prove the security of the protocol against semi-honest adversaries. We begin by defining the leakage of our protocol.

Leakage 1. $\mathcal{L}(G, \mathtt{ST}, M, P_i, \mathcal{C})$ *On input a graph* $G = \langle V, E \rangle$*, a spanning tree* $\mathtt{ST} = \langle \mathtt{root} \in V, T \subset E \rangle$ *over* G*, a mapping* $M := \mathcal{P} \leftrightarrow V$*, a requester participant* $P_i \in \mathcal{P}$*, and a set of corrupted participants* $\mathcal{C} \subset \mathcal{P}$*, output, for each* P *in* $\mathcal{C}$*, the distance and direction (edge to children or parent) from* $M(P)$ *to* $M(P_i)$ *in* $\mathtt{ST}$*, its depth (distance to* $\mathtt{ST}$*'s root node), and the height of each of its children nodes (distance to further leaf on subtree).*

Theorem 1. *Let* $\mathcal{H} = \langle \mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{Add} \rangle$ *be a semantically secure homomorphic public-key encryption scheme. Then, protocol* Π_{ReqResp} *privately realizes functionality* $\mathcal{F}_{\mathsf{ReqResp}}$ *in the* $\mathcal{F}_{\mathsf{network}}$*-restricted model under Leakage 1.*

In the following proof we analyze the case in which the server is not corrupted and there is at least one other honest node (otherwise, the location of the server node is known anyway).

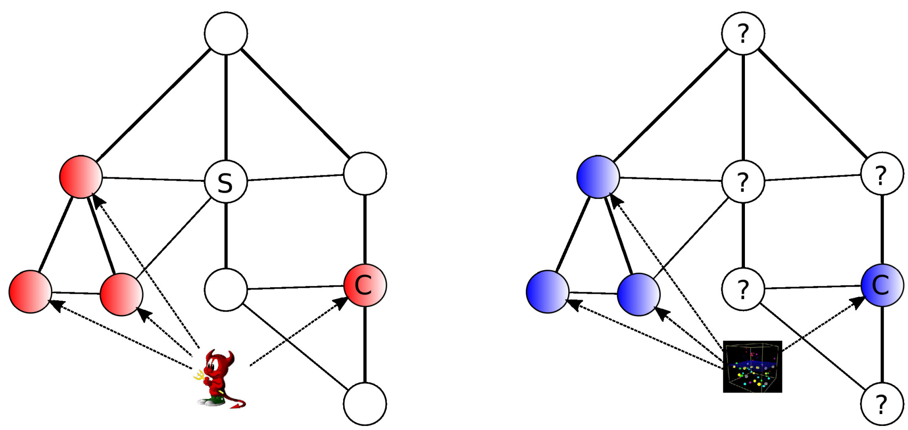

Figure 7: **Real v/s Ideal world:** on the left-hand picture the real-world protocol is executed and the adversary controls a subset of the nodes (in red) that in this example include the client C. The goal of the simulator (right-hand picture) is to reproduce the real-world communication patterns of the real-adversary *without* knowing the location of the server $\mathcal{S}$.

Proof. Let $\mathcal{C}$ be the set of corrupted participants, and $\mathcal{H}$ be the public key encryption scheme as in protocol Π_{ReqResp}. We next specify the behavior of the ideal adversary (simulator) on each of the protocol phases (see Figure 7).
Simulating Setup. In the setup phase, the corrupted participants only receive their key-pairs, the server's public key $\mathsf{pk}_{\mathcal{S}}$.

1. Instantiate network functionality $\mathcal{F}_{\mathsf{network}}$ using graph G for the participant set.

2. Generate server public key $\mathsf{pk}_{\mathcal{S}}$.

3. For each corrupted party, assign its spanning tree edges (to children and parent) and $\mathsf{pk}_{\mathcal{S}}$.

Simulating Req. Let $P_{j^*} = \mathcal{S} \notin \mathcal{C}$ be the server participant. The simulation proceeds as follows:

1. Sample session id sid, key-pair $(\mathsf{sk}_{\mathsf{sid}}, \mathsf{pk}_{\mathsf{sid}})$.

2. If $P_i \in \mathcal{C}$, then upon receiving input $\mathtt{req}$ from P_i, run real adversary on input $\mathtt{req}, P_i$ to obtain possibly updated request $\mathtt{req}'$. Send $\mathtt{req}'$ to P_i as its input and get its output $\mathtt{res}$. Otherwise set $\mathtt{req}'$ and $\mathtt{res}$ to arbitrary value.

3. Using distance and direction from corrupted participants to P_i (obtained from leakage profile), simulate a P_i started multicast protocol on spanning tree with message $\langle \mathtt{request\_to\_server}, \mathsf{sid}, \mathsf{Enc}_{\mathsf{pk}_{\mathcal{S}}}(\mathtt{req}'), \mathsf{pk}_{\mathsf{sid}} \rangle$ where sid and $\mathsf{pk}_{\mathsf{sid}}$ are fresh values. (That is, the corrupted participants get $\langle \mathtt{request\_to\_server}, \mathsf{sid}, \mathsf{Enc}_{\mathsf{pk}_{\mathcal{S}}}(\mathtt{req}'), \mathsf{pk}_i \rangle$ at the "right moment" and through the expected graph edge.)

4. Simulate the $\mathtt{to\_server\_UP}$ messages by assigning a random share share_j to each corrupted participant, and assigning an arbitrary share to the honest children of each corrupted participant. Then, the simulation is done by adding the incoming message $\langle c_i, \mathsf{sid}, \mathtt{to\_server\_UP}, S_{c_i} \rangle$ in the transcript at the right place, meaning children c_i sent his share $S_{c_i} = \mathsf{Enc}_{\mathsf{pk}_{\mathcal{S}}}(\mathsf{share}_{c_i})$.

5. Use corrupted participant depth to simulate the $\mathtt{to\_server\_DOWN}$ message by adding the message $\langle \mathsf{sid}, \mathtt{to\_server\_DOWN}, S \rangle$ to each corrupted participant simulated transcript at the right moment, like in the previous step. If the root of the tree is corrupted, then S must match the homomorphic computed value of the sum of the nodes shares. Otherwise, S can encrypt an arbitrary value.

6. Simulate each participant sending the $\mathtt{to\_requester\_UP}$ message were shares are identical as in step 4, except the honest participants, whose share are updated so that reconstruction yields $\mathtt{res}$.

7. Simulate $\mathtt{to\_requester\_DOWN}$ by adding $\langle \mathsf{sid}, \mathtt{to\_requester\_DOWN}, C \rangle$ to the simulated transcript of each corrupted participants in the path root-to-requester, where $C = \mathsf{Enc}_{\mathsf{pk}_{\mathsf{sid}}}(\mathtt{res})$.

The simulation above is perfect in terms of communication patterns (timing, length and type of messages). This is because the leakage profile contains all the information to "deliver" the messages to the corrupted participants at the right time and through the correct graph edges. Hence, the security of the protocol relies on the ability to simulate the content of the messages seen by the corrupted nodes. We next analyze the content by message type:

- Request multicast. If the request is known to the simulator, it can produce a ciphertext identically distributed to the real message. Otherwise, the simulator produces the encryption of 0^{inlen} (computationally indistinguishable to the real message by the semantic security of the encryption scheme).

- `to_server_UP` and `to_server_DOWN` messages. There is no secret information to simulate. Hence, the simulator produces ciphertexts identically distributed to the real protocol messages.

- `to_requester_UP` and `to_requester_DOWN` messages. Here, the shares corresponding to honest participant are updated so that the reconstruction produces `res`. In the worst case that the adversary controls P_i, then it can decrypt these shares. However, these cannot be correlated with the ones sent to the server in the `to_server_UP`/`to_server_DOWN` messages, since these were encrypted under the server's public key (which is assumed not to be corrupted). In addition, shares are uniformly distributed, $(n-1)$-wise independent, and they reconstruct to the same valid output `res`. Therefore, the simulated shares in plaintext cannot be distinguished from the ones used in the real execution.

A simple hybrid-argument[4] over the security of the encryption scheme proves that the real and simulated views are computationally indistinguishable.

□

6 Handling malicious adversaries with identifiable abort

In this section we informally describe the changes needed for our protocol in order to cope with active adversaries. Our goal is twofold: first, we want to ensure that a malicious adversary will not be able to gain any useful information about the location of the server (nor the request/response in case the adversary does not control the client). And second, we enable the detection and identification of malicious players that abort or send malformed messages. We refer the reader to Appendix B for a formal security definition of this adversarial model.

Our new protocol has to account for the following malicious behaviors:

[4]Changing at each hybrid step the honest participant updated shares in the `to_requester_UP` messages from the ideal distribution to the corresponding ciphertext on the real distribution. Note that the fact we are in the multi-user setting (a message is encrypted under two different public keys) can be reduced to the single-user setting (standard IND-CPA security definition)[3].

- Full or partial aborts (e.g. following a multicast protocol in only a subset of edges).
- Malformed or inconsistent messages.

We will assume that honest parties form a connected subgraph of the entire network graph G[5]. This assumption implies that the adversary is not able to cut off honest nodes from their well-behaving peers. Under this assumption, we can make sure that full aborts are detectable and partial ones are harmless: we replace the "up-and-down" messages on the spanning-tree with multicast invocations on the entire graph. That is, encrypted shares are now sent via a Π_{MCast}. The recipient decrypts each share and combine all of them in plaintext (we do not use homomorphic encryption in this protocol). It is important to note that, in order to keep hiding the location of the server, each participant needs to send its share for the client after it has seen all of the encrypted shares for the server in the previous phase. Consequently, the communication pattern of honest nodes (which includes the server) are identical.

A harder task is to detect malformed or inconsistent messages. These can have the following forms:

1. Client issues different requests through its edges.
2. Participants actively create new requests.
3. Corrupted nodes change the multicast message it receives before forwarding.
4. Participants send unexpected messages.
5. Participants send different or malformed shares during phase 2 (shares to the server) and phase 3 (shares to the client), causing error on the reconstruction of the response.

On case 1 above, the client is corrupted. If the client issues requests with different sids, then this behavior is seen by other participant as different protocol instances, in which on each of these instances the client is partially aborting. Hence, this is not considered a security breach. On the other hand, a corrupted client can use same session ids for different request. In order to handle this, the participants will use the complete request message as the session id. That is, $\mathsf{ssid} = \langle \mathsf{Enc}_{\mathsf{pk}_S}(\mathtt{req}), \mathsf{sid}, \mathsf{pk}_{\mathsf{sid}} \rangle$.

In case 2, we consider the behavior in which corrupted participants can also create new request at any point during the execution of other instances. This is problematic since a corrupted set of participants can try to learn the response that the client would have gotten by just changing the session public key. Although honest participants will see two different requests, they cannot detect which one is valid. We solve this by forcing the client to sign its request.

[5] Otherwise the adversary would be able to perform eclipse attacks [28] on some subset(s) of honest nodes which would yield honest nodes to be tagged as malicious.

In addition, we make the participants in the multicast protocol account for the messages they propagate by signing them as well. This way, the honest nodes have the ability to detect, identify, and prove to others the malicious behavior of a corrupted node. Note that these verifications solve case 3 too.

For case 4 above, we require that each message contains a session identifier of the protocol instance and the phase (`Request`, `to_server`, or `to_client`) they are executing. If the message is unexpected, it can be discarded, and treated as a simple harmless abort (as discussed above).

For case 5 we proceed as follows. First, the encrypted shares the participant submitted in phases 2 and 3 (to server and to requester respectively) need to be accompanied with a zero-knowledge proof that the message is encrypted under the correct key. However, this is not sufficient as dishonest nodes can send different, yet well-formed, shares on phases 2 and 3. Hence, we additionally append a zero-knowledge proof that the two messages encrypt the same message under different public keys. Unfortunately, this is not sufficient either since the server actually has to change its share on phase 3. Hence, this zero-knowledge proof needs to convince that either the ciphertexts encrypt the same message, OR the sender is the server. Nodes that see these messages can verify the proofs and, if one of these fails, then they broadcast the messages as evidence of the malicious behavior of the corrupted participant (since these messages are signed, the proof can be verified by others).

In summary, we modify the protocol described in Figure 6 as follows:

- Requests takes the form $\langle \mathsf{ssid} = (\mathsf{Enc}_{\mathsf{pk}_S}(\mathtt{req}), \mathsf{sid}, i), \sigma\rangle$, where σ is the client's signature of the request message.
- Every message in a multicast protocol is signed by each propagating agent. Any invalid message is disregarded.
- Encrypted shares to the server are send via multicast and take the following form: $\langle(\mathsf{ssid}, \mathsf{to\_server}, S, \phi), \sigma\rangle$, where ϕ is a zero-knowledge proof that $S \in \mathsf{Enc}_{\mathsf{pk}_S}(\mathsf{share})$ and that share belongs to the secret-sharing scheme message space, and σ is the issuer's signature on the message.
- Encrypted shares to the client are sent via multicast and take the following form: $\langle(\mathsf{ssid}, \mathsf{to\_client}, C, \phi, \pi), \sigma\rangle$, where ϕ and σ are as above, π is a zero-knowledge proof that S and C encrypt the same ciphertext, or that the issuer is the server.
- For each single message seen as part of the protocol execution, the participants do the following actions:
  - Check signature of the message. If verification fails, disregard it.
  - $\mathsf{to\_server}$ and $\mathsf{to\_client}$ messages. After verifying the message signatures, verify proof ϕ and, if corresponds, check proof π. If any verification fails, issue a multicast message $\mathsf{malicious} - \mathsf{detected}$ containing the entire message received.

- At the end of the protocol, honest participants output the set of participants for which they have evidence that they have misbehaved and participants that fully aborted (since we assume that honest nodes are connected, these can propagate all correct messages). Hence, if some participant fully aborted, then honest parties will agree on its identity after a reasonable timeout.

More concretely, we can use ElGamal as the encryption scheme so that shares are elements of a DDH group $\mathbb{G}$. In addition, ElGamal allow us to simplify our protocol by removing proofs ϕ above and replace them with a simple checking that the ciphertexts are of the correct form ($\in \mathbb{G} \times \mathbb{G}$) (since an ElGamal ciphertexts can encrypt different message for diverse keys and randomness). The proof π is reduced to a non-interactive zero-knowledge proof of the equality of two discrete logarithms. The detailed construction is available in Appendix A.

Note that due to our use of digital signatures, our new protocol reveals the identity of the client and the distance of each honest node to each corrupted node. Also, given that we replaced spanning-tree up-and-down messages with multicast invocations, the communication complexity increases by a $O(N)$ factor.

7 Conclusion

We have introduced a new protocol that enables to hide a server in a network in the semi-honest model. This protocol has several advantages other previous proposals: it is efficient, asynchronous and collusion-resistant. To the best of our knowledge this is the first solution with these characteristics. In addition, we sketched an extension of our protocol to cope with active adversaries. In this setting, our solution allows honest participants to identify corrupted ones. In fact, dishonest nodes can only force a premature termination of the protocol.

We believe that this work is an important step towards designing practical and provably secure systems that enable to hide relevant meta-data (such as the identity or location of participants) in a controllable way. Future work directions include reducing the communication complexity of the extended protocol for active adversaries, improve the resilience of out solution against termination attempts, and prove our results in stronger security models (such as the UC framework [4] with adaptive corruption).

References

[1] Adi Akavia, Rio LaVigne, and Tal Moran. Topology-hiding computation on all graphs. Cryptology ePrint Archive, Report 2017/296, 2017. `http://eprint.iacr.org/2017/296`.

[2] Adi Akavia and Tal Moran. Topology-hiding computation beyond logarithmic diameter. In *Advances in Cryptology - EUROCRYPT 2017 - 36th*

Annual International Conference on the Theory and Applications of Cryptographic Techniques, Paris, France, April 30 - May 4, 2017, Proceedings, Part III, pages 609–637, 2017.

[3] Mihir Bellare, Alexandra Boldyreva, and Silvio Micali. Public-key encryption in a multi-user setting: Security proofs and improvements. In *International Conference on the Theory and Applications of Cryptographic Techniques*, pages 259–274. Springer, 2000.

[4] R. Canetti. Universally Composable Security: A New Paradigm for Cryptographic Protocols. page 136, oct 2001.

[5] Ran Canetti. Security and composition of multiparty cryptographic protocols. *J. Cryptology*, 13(1):143–202, 2000.

[6] David Chaum. The dining cryptographers problem: Unconditional sender and recipient untraceability. *Journal of cryptology*, 1(1):65–75, 1988.

[7] David L Chaum. Untraceable electronic mail, return addresses, and digital pseudonyms. *Communications of the ACM*, 24(2):84–90, 1981.

[8] Roger Dingledine, Nick Mathewson, and Paul Syverson. Tor: The second-generation onion router. Technical report, Naval Research Lab Washington DC, 2004.

[9] Shlomi Dolev and Rafail Ostrovsky. Xor-trees for efficient anonymous multicast and reception. *ACM Trans. Inf. Syst. Secur.*, 3(2):63–84, 2000.

[10] Taher ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. *IEEE transactions on information theory*, 31(4):469–472, 1985.

[11] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In *Conference on the Theory and Application of Cryptographic Techniques*, pages 186–194. Springer, 1986.

[12] Oded Goldreich. *Foundations of Cryptography: Volume 2, Basic Applications*. Cambridge University Press, New York, NY, USA, 2004.

[13] David Goldschlag, Michael Reed, and Paul Syverson. Onion routing. *Communications of the ACM*, 42(2):39–41, 1999.

[14] Martin Hirt, Ueli Maurer, Daniel Tschudi, and Vassilis Zikas. Network-hiding communication and applications to multi-party protocols. Cryptology ePrint Archive, Report 2016/556, 2016. `http://eprint.iacr.org/2016/556`.

[15] Yuval Ishai, Rafail Ostrovsky, and Vassilis Zikas. Secure multi-party computation with identifiable abort. In *Advances in Cryptology - CRYPTO 2014 - 34th Annual Cryptology Conference, Santa Barbara, CA, USA, August 17-21, 2014, Proceedings, Part II*, pages 369–386, 2014.

[16] Sachin Katti, Dina Katabi, and Katarzyna Puchala. Slicing the onion: Anonymous routing without pki. 2005.

[17] Brian N Levine, Michael K Reiter, Chenxi Wang, and Matthew Wright. Timing attacks in low-latency mix systems. In *International Conference on Financial Cryptography*, pages 251–265. Springer, 2004.

[18] Brian Neil Levine and Clay Shields. Hordes: a multicast based protocol for anonymity1. *Journal of Computer Security*, 10(3):213–240, 2002.

[19] Tal Moran, Ilan Orlov, and Silas Richelson. *Topology-Hiding Computation*, pages 159–181. Springer Berlin Heidelberg, Berlin, Heidelberg, 2015.

[20] Steven J Murdoch and George Danezis. Low-cost traffic analysis of tor. In *Security and Privacy, 2005 IEEE Symposium on*, pages 183–195. IEEE, 2005.

[21] Lasse Overlier and Paul Syverson. Locating hidden servers. In *Security and Privacy, 2006 IEEE Symposium on*, pages 15–pp. IEEE, 2006.

[22] Andreas Pfitzmann, Birgit Pfitzmann, and Michael Waidner. Isdn-mixes: Untraceable communication with very small bandwidth overhead. In *Kommunikation in verteilten Systemen*, pages 451–463. Springer, 1991.

[23] Charles Rackoff and Daniel R Simon. Cryptographic defense against traffic analysis. In *Proceedings of the twenty-fifth annual ACM symposium on Theory of computing*, pages 672–681. ACM, 1993.

[24] Michael K Reiter and Aviel D Rubin. Crowds: Anonymity for web transactions. *ACM transactions on information and system security (TISSEC)*, 1(1):66–92, 1998.

[25] Berry Schoenmakers. *Cryptographic Protocols (2DMI00)*. 2017.

[26] Andrei Serjantov and Peter Sewell. Passive attack analysis for connection-based anonymity systems. *Lecture notes in computer science*, 2808:116–131, 2003.

[27] Vitaly Shmatikov. Probabilistic analysis of anonymity. In *Computer Security Foundations Workshop, 2002. Proceedings. 15th IEEE*, pages 119–128. IEEE, 2002.

[28] Atul Singh et al. Eclipse attacks on overlay networks: Threats and defenses. In *In IEEE INFOCOM*. Citeseer, 2006.

[29] Michael Waidner. Unconditional sender and recipient untraceability in spite of active attacks. In *Workshop on the Theory and Application of of Cryptographic Techniques*, pages 302–319. Springer, 1989.

[30] Matthew K Wright, Micah Adler, Brian Neil Levine, and Clay Shields. An analysis of the degradation of anonymous protocols. In *NDSS*, volume 2, pages 39–50, 2002.

[31] Ye Zhu, Xinwen Fu, Bryan Graham, Riccardo Bettati, and Wei Zhao. On flow correlation attacks and countermeasures in mix networks. In *International Workshop on Privacy Enhancing Technologies*, pages 207–225. Springer, 2004.

A Proving that two ciphertexts encrypt the same plaintext or the participant is the server

In this section we describe the scheme that enables to check the consistency of the encrypted shares. At a conceptual level we need to prevent a malicious participant from sending shares that are not equal to the server and client. At the same time we need to deal with the restriction that the server indeed uses different shares in the protocol. Obviously this fact cannot be revealed to the other participants without leaking the server's location[6]

In order to achieve these two apparently contradictory goals we use the following approach: each participant will prove in zero-knowledge that (1) the encrypted shares correspond to the same plaintext OR (2) the encrypted shares have been produced by the server.

We use the following algorithms: $\mathtt{Prove}(\mathsf{aux}, C^1, C^2, \mathsf{pk}_\mathcal{C}, \mathsf{pk}_\mathcal{S})$ takes as input some auxiliary information aux, ciphertexts C^1, C^2 and public keys $\mathsf{pk}_\mathcal{C}, \mathsf{pk}_\mathcal{S}$ of the client and server respectively. The output is some proof π. Then, given two ciphertexts C^1, C^2, the public key of the client $\mathsf{pk}_\mathcal{C}$, the public key of the server $\mathsf{pk}_\mathcal{S}$ and the proof π computed earlier, it is possible to verify the consistency of the ciphertexts by running $\mathtt{Check}(C^1, C^2, \mathsf{pk}_\mathcal{C}, \mathsf{pk}_\mathcal{S}, \pi)$ that will return `ACCEPT` in case the verification is successful or $\perp$ otherwise.

We instantiate our construction using the Elgamal [10] encryption scheme: Let $\mathbb{G}$ be a cyclic group of prime order p and generator $g \in \mathbb{G}$. Let $\mathsf{sk} \in \mathbb{Z}_p$ be a private key and $\mathsf{pk} := g^{\mathsf{sk}}$ be the corresponding public key. We define the encryption of a plaintext $m \in \mathbb{G}$ using randomness $r \in \mathbb{Z}_p$ as $\mathsf{Enc}_{\mathsf{pk}}(m) := (m \cdot \mathsf{pk}^r, g^r) = (C_1, C_2)$. To decrypt a ciphertext (C_1, C_2) using private key sk, one needs to compute $\mathsf{Dec}_{\mathsf{sk}}((C_1, C_2)) := C_1 \cdot C_2^{-\mathsf{sk}} = m \cdot (g^{\mathsf{sk}})^r \cdot (g^r)^{-\mathsf{sk}} = m$. Let $A := g^s$ be an element of $\mathbb{G}$ for some $s \in \mathbb{Z}_p$. The discrete logarithm of A in base g is denoted $\mathsf{Dlog}_g(A) := s$. Let $R = \{(v; w)\} \subseteq V \times W$ be a NP-relation. We denote by $PK\{(w) : (v, w) \in R)\}$ a protocol where a prover is able to convince a verifier of the knowledge of some witness w that satisfies some relation R, i.e $(v, w) \in R$ for some public value v. For example $PK\{(s) : A = g^s\}$ denotes the proof of knowledge of the discrete logarithm of $A \in \mathbb{G}$ in base g. A $\Sigma-$protocol is a three rounds interaction between the prover and verifier that can be used to prove in zero-knowledge the knowledge of some witness without revealing

[6] Detecting that a specific message comes from the server implies in practice that the server's location will be leaked at some point.

this witness. Σ-protocols can be made non interactive using the Fiat-Shamir heuristic[11].

The idea of our construction is as follows: given two Elgamal ciphertexts $C^1 = (C_1^1, C_2^1) = (m_1 \cdot \mathsf{pk}_{\mathcal{C}}^r, g^r)$ and $C^2 = (C_1^2, C_2^2) = (m_2 \cdot \mathsf{pk}_{\mathcal{S}}^r, g^r)$ encrypted with the same randomness r, the prover will either show that (1) $m_1 = m_2$ by proving the equality of discrete logarithm of $C_1^1 \cdot (C_1^2)^{-1} = (\mathsf{pk}_{\mathcal{C}} \cdot \mathsf{pk}_{\mathcal{S}}^{-1})^r$ and g^r in bases $\mathsf{pk}_{\mathcal{C}} \cdot \mathsf{pk}_{\mathcal{S}}^{-1}$ and g respectively, OR (2) the knowledge of some secret s known to the server (e.g. s such that $D = g^s$, for some public D). Let $\delta = \mathsf{pk}_{\mathcal{C}} \cdot \mathsf{pk}_{\mathcal{S}}^{-1}$.

Construction 1. *[Proof of shares consistency]*
`Prove`$(\mathsf{aux}, C^1, C^2, D, \mathsf{pk}_{\mathcal{C}}, \mathsf{pk}_{\mathcal{S}})$*:*

- *Let* $C^1 = (C_1^1, Y)$ *and* $C^2 = (C_1^2, Y)$*.*
- *Compute* $\Delta := C_1^1 \cdot (C_1^2)^{-1}$ *(*$= \delta^r$*).*
- *If the prover is the server* $\mathcal{S}$ *then* $\mathsf{aux} := s = \mathsf{Dlog}_g(D)$ *otherwise* $\mathsf{aux} := r = \mathsf{Dlog}_\delta(\Delta) = \mathsf{Dlog}_g(Y)$*.*
- *Compute and return*
  π *for* $PK\{(r,s) : [\Delta = \delta^r \wedge Y = g^r] \vee [s : D = g^s]\}$*.*

`Check`$(C^1, C^2, \mathsf{pk}_{\mathcal{C}}, \mathsf{pk}_{\mathcal{S}}, \pi)$*:*

- *Let* $C^1 = (C_1^1, Y_1)$ *and* $C^2 = (C_1^2, Y_2)$*.*
- *Check that* $Y_1 = Y_2$*. If not return* $\perp$*.*
- *Compute* $\Delta := C_1^1 \cdot (C_1^2)^{-1}$*.*
- *Verify* π *using* $\Delta, \mathsf{pk}_{\mathcal{C}}, \mathsf{pk}_{\mathcal{S}}$*. If the verification is successful return* `ACCEPT` *otherwise return* $\perp$*.*

The reader can verify that if, indeed, $g^{\mathsf{Dlog}_\delta(\Delta)} = g^r$, then $\mathsf{Dec}_{\mathsf{sk}_{\mathcal{C}}}((C_1^1, g^r)) = \mathsf{Dec}_{\mathsf{sk}_{\mathcal{S}}}((C_1^2, g^r))$.

The Σ-protocol $PK\{(r,s) : [\Delta = \delta^r \wedge Y = g^r] \vee [s : D = g^s]\}$ that corresponds to the relation $R_{g,\delta} := \{(A, B; r) : A = (\delta)^r \wedge B = g^r\} \cup \{(D; s) : D = g^s\} = R_0 \cup R_1$ is described in Figure 8. In this protocol, values A and B correspond to Δ and g^r respectively, and D to g^s, where s is the server's secret.

The completeness of the protocol can be verified by inspection. Soundness can be shown as follows[7]. Assume that we obtain two conversations $(a_1, a_2, a_3; c; c_0, c_1, r_0, r_1)$ and $(a_1, a_2, a_3, c', {c_0}', {c_1}', {r_0}', {r_1}')$ such that $c \neq c'$. Given that $c = c_0 + c_1$ and $c' = {c_0}' + {c_1}'$ then either $c_0 \neq {c_0}'$ or $c_1 \neq {c_1}'$. Moreover we have that $g^{r_0} = a_2 B^{c_0}, g^{r_1} = a_3 D^{c_1}, \delta^{r_0} = a_1 A^{c_0}, g^{{r_0}'} = a_2 B^{{c_0}'}, g^{{r_1}'} = a_3 D^{{c_1}'}$, and $\delta^{{r_0}'} = a_1 A^{{c_0}'}$. We can deduce that:

- if $c_0 \neq {c_0}'$ then $g^{r_0 - {r_0}'} = B^{c_0 - {c_0}'}$ and $\delta^{r_0 - {r_0}'} = A^{c_0 - {c_0}'}$ which implies that $r = \frac{r_0 - {r_0}'}{c_0 - {c_0}'}$.

[7]See Berry Schoenmakers' lectures notes[25] for a security definition of Σ-protocols.

$$
\begin{array}{c|c c c}
\multicolumn{2}{c}{\textbf{Prover}} & & \textbf{Verifier} \\
R_0 = \{(A,B;r) : A = \delta^{r} \wedge B = g^{r}\} & R_1 = \{(D;s) : D = g^{s}\} & & \\
u_0, r_1, c_1 \in_R \mathbb{Z}_p & u_1, r_0, c_0 \in_R \mathbb{Z}_p & & \\
a_1 := \delta^{u_0} & a_1 := \delta^{r_0} \cdot A^{-c_0} & & \\
a_2 := g^{u_0} & a_2 := g^{r_0} \cdot B^{-c_0} & & \\
a_3 := g^{r_1} \cdot D^{-c_1} & a_3 := g^{u_1} & & \\
 & & \xrightarrow{a_1,a_2,a_3} & \\
 & & \xleftarrow{c} & \\
c_0 := c - c_1 & c_1 := c - c_0 & & \\
r_0 := u_0 + c_0 r & r_1 := u_1 + c_1 s & & \\
 & & \xrightarrow{c_0,c_1,r_0,r_1} & \\
 & & & c_0 + c_1 \stackrel{?}{=} c \\
 & & & \delta^{r_0} \stackrel{?}{=} a_1 \cdot A^{c_0} \\
 & & & g^{r_0} \stackrel{?}{=} a_2 \cdot B^{c_0} \\
 & & & g^{r_1} \stackrel{?}{=} a_3 \cdot D^{c_1}
\end{array}
$$

Figure 8: Σ-protocol for relation $R_{g,\delta} = \{(A, B; r) : A = \delta^r \wedge B = g^r\} \cup \{(D; s) : D = g^s\}$.

- if $c_1 \neq c_1{}'$ then $g^{r_1 - r_1{}'} = D^{c_1 - c_1{}'}$ which implies that $s = \frac{r_1 - r_1{}'}{c_1 - c_1{}'}$.

To show that the protocol is zero-knowledge let us consider a challenge c. Let us assume that the prover knows r. Then the honest-verifier distribution is $\{(a_1, a_2, a_3, c; c_0, c_1, r_0, r_1) : u_0, r_1, c_1 \in_R \mathbb{Z}_p; a_1 = \delta^{u_0}; a_2 = g^{u_0}; c_0 = c - c_1; r_0 = u_0 + c_0 r\}$. The simulated distribution is set as $\{(a_1, a_2, a_3, c; c_0, c_1, r_0, r_1) : c_0, r_0, r_1 \in_R \mathbb{Z}_p; c_1 = c - c_0, a_1 = \delta^{r_0} A^{-c_0}, a_2 = g^{r_0} B^{-c_0}, a_3 = g^{r_1} D^{-c_1}\}$. These distributions are identical. The case where the prover knows s is similar.

B Security definition for active adversaries

In order to capture detectable and identifiable malicious activity, we follow the approach of Ishai et. al. [15] and modify the standard security definition for malicious adversaries [5, 12], and we adapt it for our request response functionality $\mathcal{F}_{\mathsf{ReqResp}}$. Analogously to the semi-honest case described in section 2.4, an external party (similar to an environment of the UC setting [4]) chooses the client participant and its input.

Ideal-world with identifiable aborts. The environment chooses adversaries input $\mathtt{in}_{\mathsf{Sim}}$, client participant P_i, and P_i's input $\mathtt{req}$. The adversary then informs the functionality $\mathcal{F}_{\mathsf{ReqResp}}$ about corrupted participant set $\mathcal{C} \subset \mathcal{P}$. The client P_i forwards its input $\mathtt{req}$ to the functionality. If $P_i \in \mathcal{C}$, then $\mathcal{F}_{\mathsf{ReqResp}}$ request new input $\mathtt{req}$ to the adversary[8]. $\mathcal{F}_{\mathsf{ReqResp}}$ then submits leakage pro-

[8]We do not consider the trivial case in which the adversary abort before submitting its input.

file $\mathcal{L}(G, \texttt{req}, P_i)$ to adversary and waits for its response. If the response is $\texttt{continue}$, then the functionality computes response $\texttt{res}$ and sends it to P_i, other participants output $\perp$. If adversary's response is $\langle \texttt{abort}, \mathcal{K} \rangle$ ($\mathcal{K} \subseteq \mathcal{C}, |\mathcal{K}| \geq 1$)[9], all honest participants get $\langle \texttt{abort}, \mathcal{K} \rangle$. Let $\mathsf{ID} - \mathsf{Ideal}^{\mathcal{F}_{\mathsf{ReqResp}}}_{\mathcal{Z}, \mathsf{Sim}}(\kappa)$ be the random variable denoting the joint output of the honest participants and the adversary Sim.

Real-World. As in the ideal world above, the environment chooses adversary's input, the client and its input. The adversary then chooses the corrupted participant set, and executes an arbitrary (yet polynomially bounded strategy) on behalf of them. The honest participants follow the prescribed protocol. Let $\mathsf{Real}^{\Pi}_{\mathcal{Z}, \mathsf{Adv}}(\kappa)$ be the random variable denoting the joint output of the honest participants and the adversary Adv.

Definition 2. *A protocol* Π securely computes *functionality* $\mathcal{F}_{\mathsf{ReqResp}}$ with identifiable aborts *if for every PPT algorithm* Adv*, there exists a PPT algorithm* Sim *such that for every PPT algorithm* $\mathcal{Z}$ *the random variables* $\mathsf{ID} - \mathsf{Ideal}^{\mathcal{F}_{\mathsf{ReqResp}}}_{\mathcal{Z}, \mathsf{Sim}}(\kappa)$ *and* $\mathsf{Real}^{\Pi}_{\mathcal{Z}, \mathsf{Adv}}(\kappa)$ *are computationally indistinguishable.*

[9]That is, the adversary reveals the identity of at least one dishonest participant.